POLITECNICO DI TORINO

Master's Degree in Cybersecurity



Master's Degree Thesis

Formal Analysis of eBPF Helper Functions and Ring Buffer Synchronization via SPIN Model Checking

Supervisors

Prof. Riccardo SISTO

Dott. Rosario RIZZA

Candidate

Alexandre GAUTIER

September 2026

Knowing how to think empowers you
far beyond those who know only what to think.

— ***Neil deGrasse Tyson***

Abstract

Extended Berkeley Packet Filter (eBPF) has become an increasingly powerful and useful tool for the Linux kernel observability, security and networking. eBPF programs are attached to events happening in the kernel such as system calls or tracepoints and transfer data through eBPF objects called maps. High performance events collection and investigation rely heavily on an eBPF map called ring buffer, a pointer-based communication channel optimized for data transfer. However the kernel and the user space rely on shared-memory synchronized pointers to govern the buffer's layout, consistency and free space. This thesis studies the security and the architectural implications of this synchronization mechanism between concurrent kernel and the user space programs.

To formally analyze these mechanics, I used the Promela language alongside the SPIN model checker, abstracting the behavior of this shared buffer structure focusing on the eBPF-specific helper functions used specifically to interact with it. This formal analysis exposes concurrency hazards, demonstrating how the desynchronization of structural pointers can create behavioral and security violations. While no behavioral invariant has been found violated by the formal model checker, some security specifications has been breached, because of the lack of protections around the consumer pointer, value which specifies to the user-space libraries where to read data in the ring buffer.

To bridge formal verification with real-world implementations, I develop C-based eBPF programs testing the formal results under thread-concurrent conditions. This experiments permits to determine if the results are indeed real flaws in the implementation or only artifacts from the formal abstraction models.

Acknowledgements

First and foremost, I wish to express my deepest gratitude to my academic supervisors, Prof. Riccardo SISTO and Dott. Rosario RIZZA for their expert insight and rigorous feedback throughout this research work. Their profound knowledge of eBPF internals and model checkers steered this project and helped me overcome complex hurdles.
I also want to extend my deepest appreciation to my friends and family, for their constant support, especially when obstacles seemed insurmountable and my faith was faltering.

Table of Contents

List of Tables VII

List of Figures VIII

Acronyms IX

1 Introduction 1

2 Background on eBPF 3
- 2.1 Presentation of eBPF . . . 3
- 2.2 Kernel and User space codes . . . 4
  - 2.2.1 Programs Life Cycle . . . 4
  - 2.2.2 eBPF Maps . . . 5
  - 2.2.3 Helper Function . . . 5
  - 2.2.4 eBPF Skeleton . . . 6
- 2.3 eBPF Verifier . . . 7
- 2.4 eBPF Hooks . . . 8
  - 2.4.1 Networking . . . 8
  - 2.4.2 Tracing . . . 8
  - 2.4.3 Security . . . 9

3 Background on Spin Model Checker 10
- 3.1 Model Checking . . . 10
- 3.2 Spin Model Checker . . . 11
  - 3.2.1 Correctness Properties . . . 12
  - 3.2.2 Monitoring the Model . . . 12
  - 3.2.3 Execution Modes . . . 12

4 Thesis Objectives 14
- 4.1 Motivation . . . 14
- 4.2 Specific Objectives . . . 15

4.3 Scope and Contributions . . . 16
4.3.1 Focus on Ring Buffers . . . 16
4.3.2 Initial C-model Detour . . . 16
4.3.3 Chapters Roadmap . . . 16

5 Preliminary Exploration and Initial C Code Modeling 18
5.1 Useful Functions . . . 18
5.1.1 Ring Buffers . . . 18
5.1.2 Helper Functions . . . 19
5.1.3 User-space Functions . . . 21
5.1.4 Kernel and User-space Programs . . . 22
5.2 Native C modeling . . . 23
5.2.1 Core Translation . . . 23
5.2.2 Execution Environment . . . 23
5.3 eBPF Verifier Model . . . 24
5.3.1 Verifier Checks . . . 24
5.3.2 Verifier Model . . . 25
5.4 Behavioral Invariants . . . 26
5.4.1 Concurrency & Race Conditions . . . 26
5.4.2 Data Integrity & Boundary Quirks . . . 27
5.5 Execution . . . 28
5.5.1 Steps . . . 28
5.5.2 Execution Scripts . . . 29
5.5.3 Issues Encountered . . . 29
5.5.4 Results . . . 30
5.5.5 Final Model . . . 31
5.5.6 Outputs . . . 32
5.5.7 Conclusion . . . 35

6 Abstracted Model and Security Invariants 37
6.1 Abstracted model . . . 37
6.1.1 Helper Functions . . . 38
6.1.2 User-space Functions . . . 38
6.2 Rework of the Verifier Model . . . 39
6.3 Security Invariants . . . 39
6.3.1 Confidentiality . . . 41
6.3.2 Integrity . . . 41
6.3.3 Availability . . . 42

7 Discussion of the Results 44
7.1 Execution Script . . . 44
7.2 Results . . . 44
7.2.1 Out of Order Read . . . 48
7.2.2 TOCTOU . . . 48
7.2.3 Overflow . . . 49
7.2.4 Deadlock . . . 50
7.2.5 Evasion . . . 50
7.3 Replication of the Results . . . 51
7.3.1 Out of Order Read . . . 52
7.3.2 TOCTOU . . . 55
7.3.3 Overflow . . . 56
7.4 Analysis and Comparison of the results . . . 64
7.4.1 Theory vs Observations . . . 64
7.4.2 State Space Complexity . . . 64

8 Real-World Case Studies 69
8.1 Real-World Case Studies . . . 69
8.1.1 Local Program Case Study . . . 69
8.1.2 Falco Case Study . . . 71
8.2 Mitigation . . . 75

9 Conclusion 77
9.1 Summary of Work . . . 77
9.2 Key Findings . . . 77
9.3 Limitations . . . 78
9.4 Future Work . . . 78

A Bash Scripts 79

Bibliography 83

List of Tables

5.1 Naive Model Results 36

6.1 Taxonomy of invariants 40

7.1 Byte-by-byte offset results (1/3) 60
7.2 Byte-by-byte offset results (2/3) 61
7.3 Byte-by-byte offset results (3/3) 62
7.4 Byte-by-byte minus offset results 63
7.5 Models Comparison 67

8.1 Byte-by-byte minus offset results 75

List of Figures

2.1 eBPF and kernel/user space 4
2.2 Schema of a ring buffer with its pointers 6

3.1 Commands to perform the model checking 11

7.1 Results of the SPIN execution over the security Promela models . . 46
7.2 Ring buffer records memory layout 58
7.3 Actual data read by the libbpf consumer 59

A.1 Bash Script for the Naive Approach 80
A.2 Bash Script for the Asbtract Models 82

Acronyms

CLI
Command Line Interface

EDR
Endpoint Detection and Response

FD
File Descriptor

GCC
GNU Compiler Collection

GUI
Graphical User Interface

IDR
Intrusion Detection and Response

LSM
Linux Security Module

MAC
Mandatory Access Control

NIC
Network Interface Card

OS
Operating System

PID

Process Identifier

RNG

Random Number Generator

XDP

eXpress Data Path

Chapter 1

Introduction

eBPF or extended Berkeley Packet Filter, introduced in Linux 3.18 (2014) has revolutionized the kernel programmability, allowing the execution of safe and sandboxed programs for profiling, tracing and monitoring, without changing the kernel source code. It is currently used by a number of large-scale production users such as Meta for balancing the traffic, Google for networking, or Cloudflare for security enforcement. eBPF programs are loaded inside the kernel and can communicate with the user-space through the use of data structures called *maps*. To interact with the system, eBPF programs use *helper functions*. Since the kernel is a highly sensitive area of a system, these helper functions should not contain any flaw that could hinder the confidentiality, the integrity, or the availability of the transiting data. While some research has used formal methods to check individual eBPF programs correctness, applying these techniques to uncover concurrency flaws within multi producer or consumer ring buffer interactions remains unexplored in the literature.
Analyzing every possible concurrency bug and race condition within the while Linux kernel would require more resource than available during this master thesis. As a consequence, this work only focuses on a specific set of eBPF helper functions and set of concurrency issues. For these challenges, we use the SPIN model checker and its specification language, Promela, because this tool is suited for exposing deep interleaving errors and race conditions that other existing software might miss. Rather than directly analyzing the massive C code base of the kernel, we abstract its core behavior into Promela. Once SPIN detects a security invariant violation, it generates the execution trace that led to the flaw. Finally, to verify that the issue detected by the model checker is a true positive, we translate back the trail into a concrete eBPF C code and execute it. If the flaw is replicated, this means we have successfully found a security vulnerability in the helper functions.

The remainder of the thesis is structured as follows. Chapter 2 and Chapter 3 provide technical background on the eBPF architecture including its maps, helper functions and the verifier, but also on model checkers and the SPIN architecture. Chapter 4 outlines the objectives of the thesis, its scope and contributions. Chapter 5 explains a preliminary exploration, a naive model and its complications. Chapter 6 presents a refined model and formal security invariants. Chapter 7 covers the results of the model checking and the replication in real eBPF programs. Chapter 8 focuses on real-world study cases of the discovered flaws. Finally the 9th and last chapter offers a conclusion on the work done and on future research directions.

Chapter 2

Background on eBPF

This section provides a synthesis of the eBPF concepts relevant to this thesis, based primarily on [1].

2.1 Presentation of eBPF

The Linux kernel is an open source C code: this means that everyone can create *modules* to increase its capacities. However, the use of these modules requires the developer to write kernel code, which is very challenging, and to recompile of the whole kernel, which is time-consuming. Moreover, a crash in one module can take down the whole system since the latter is loaded inside the kernel. Finally, loading external modules might introduce some security issues if the developer's main goal is to use it to compromise systems. To counter these drawbacks, the Berkeley Packet Filter (or BPF) has been introduced in 1997 in the Linux kernel 2.1. Its main goal was to allow developers to write custom code that can be loaded in the kernel dynamically (without the need to recompile the whole codebase). It allowed kernel-level packet filtering and later introduced high performance network filtering. In 2014, starting from the version 3.18, the Linux kernel incorporates extended BPF (or eBPF). This came with some improvements such as a 64-bit architecture, the introduction of maps and helper functions and finally an in-kernel verifier preventing programs from crashing the kernel. eBPF programs are very useful in a native cloud approach, where containers are used, to monitor the traffic and activities at the kernel level, without modifying the pods configuration: an eBPF program can monitor globally all the current and future processes that will run on the system.

2.2 Kernel and User space codes

eBPF programs are loaded inside the kernel, triggering hooks and capturing the essential information for monitoring or packet filtering. In addition, applications such as EDRs or IDRs can read this data and perform the checks to detect an issue with the system's behavior. These applications run in the user-space of the system. A simple diagram showing how eBPF programs behave can be found in the **Figure 2.1**.
eBPF code can be written in several programming languages: C, Rust or Python. Each piece of code is translated into bytecode, easily mappable to common CPU architectures, before being loaded inside the kernel.

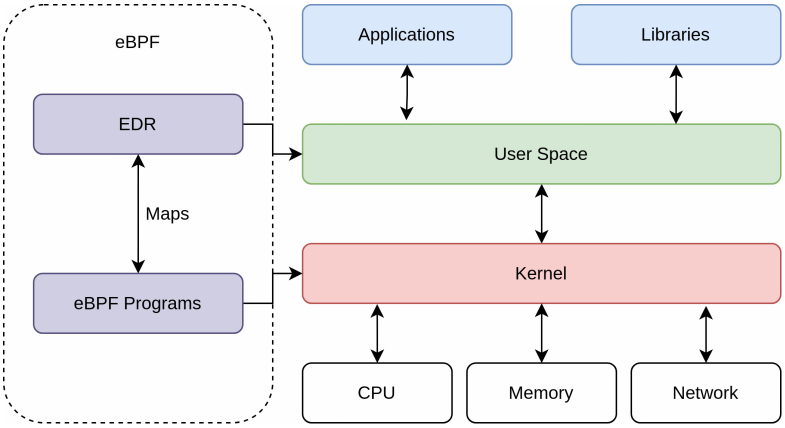


Figure 2.1: eBPF and kernel/user space

2.2.1 Programs Life Cycle

After being written, eBPF programs need to be executed to be effective. When developing eBPF programs in Python, there is a library called BCC (for BPF Compiler Collection) that ease the life cycle management of the program: only a few lines of code are needed. On the other hand, when using C or Rust, all these steps are not that straightforward. Here is a breakdown of eBPF programs life cycle:

- Compilation: the source high level code is compiled into bytecode object files with tools such as LLVM/Clang compilers.
- Load: the eBPF verifier analyses the bytecode through rigorous static analysis, by simulating execution paths to verify that the program is safe, terminates and adheres to kernel security invariants. Any code that does not follow these rules is rejected.

- JIT Compilation: the kernel's Just In Time (or JIT) compiler to translate the eBPF bytecode into pure machine code for high performance.
- Attach: the program needs to be attached to specific hook in the kernel to execute. When the kernel execution flow triggers the hook, the eBPF program runs.

While using Python is a fast and easy way to write eBPF programs, it does not handle the CO-RE (compile once, run everywhere) concept, only available with the other programming languages. This allows developers to write portable code across different platforms. Consequently, we will only use C code in this thesis to write programs.

2.2.2 eBPF Maps

A map is a data structure that can be accessed both from an eBPF program inside the kernel and from the user space. Maps can be used to share data between multiple eBPF programs or to communicate between the kernel and the user-space. The use cases can be the user-space writing a configuration for an eBPF program, or an eBPF program writing data for the user-space to retrieve.
There are many types of maps. Some are optimized for special operations, some are meant to hold information about specific objects and some have per-CPU variants, meaning that there is a different block of memory for each CPU core. The most usual map types are the following:

- Hash Table Map: as its name indicates, this map will hold key-value pairs. This is convenient when the data is naturally a key-value pair.
- **Ring Buffer Maps**: a ring buffer is a piece of memory organized as a ring, with read and write pointers to know where to write and read piece of data. When a read or write is performed on this buffer, the pointers are updated. When the pointers reach each other, this can mean that the buffer is empty (if the read pointer reaches the write one) or full (if the opposite happens) as shown by the **Figure 2.2**.

2.2.3 Helper Function

The kernel side of the eBPF programs uses helper functions. These are predefined functions, shipped with kernel versions, supposed to be stable through the releases: once one is defined in the kernel, it should not change in the future. They are used to safely interact with kernel services and data structures such as maps without risking the system's stability. All the helper functions start with the prefix `bpf_`.

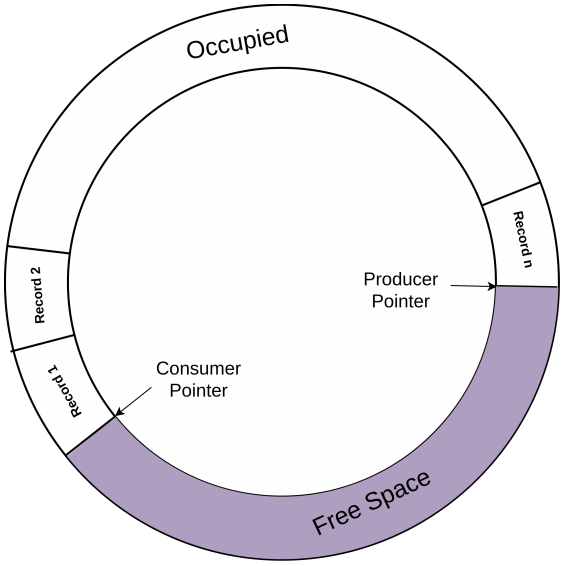


Figure 2.2: Schema of a ring buffer with its pointers

As an example of generic map helper functions, `bpf_map_lookup_elem` can be used to retrieve the value of an element from an eBPF map and `bpf_map_update_elem` allows the modification of a record. Interacting with the system, an eBPF program can retrieve the current PID of the process with the `bpf_get_current_pid_tgid` helper function, or even get the time from system boot with the `bpf_ktime_get_ns` helper function.

2.2.4 eBPF Skeleton

The user-space is responsible for the life cycle of the eBPF programs. When using maps, this introduces even more steps to perform to create the maps and access them. This can be done manually thanks to the *bpftool* command-line tool: one command to load a program inside the kernel, one to attach the eBPF programs to their hooks, one to create a map... this is very verbose. Fortunately, there is a C library called *libbpf* that allows us to handle the eBPF programs' life cycle easier. By using bpftool once, we can generate a *skeleton* from an eBPF bytecode, as a `.h` C header file. This file is used as a header for the user space. It contains structure definitions for the eBPF programs, its maps, as well as several functions that manage the eBPF program and maps life cycle, such as `__open`, `__load`, `__attach`, `__detach` and `__destroy`.

2.3 eBPF Verifier

As explained before, the role of the eBPF verifier is to make sure the eBPF code runs safely inside the kernel. It analyses the eBPF bytecode before loading it inside the kernel.
Rather than executing the instructions, it evaluates them and keeps track of the states of the registers. When it crosses a branch, such as an if/else conditional block, it pushes the registers' state onto the stack and explores all the possible paths. This is computationally expensive so the verifier uses state pruning to avoid re-evaluating equivalent registers states. For each state, the verifies ensures that the value contained in the registers is valid against a set of predefined rules, according to the different functions.
From the verifier C source code [2] and Liz Rice in [1], here are the checks performed by the eBPF verifier when loading a program in the kernel:

- **Helper functions**: each helper function has constraints for its arguments and return values. The verifier checks if the right values are used. Moreover, some helper functions can only be used with a certain hook. The verifier ensures the functions are compatible with the attach point of the eBPF program.
- **License**: each helper function is protected by a license. The eBPF program should contain a compatible license with all the helper functions it uses.
- **Memory access**: eBPF programs should only have access to the memory locations they are supposed to. For example, an eBPF program is not supposed to change the boundaries of a network packet.
- **Pointers**: before being de-referenced, the pointer needs to be non-NULL. The verifier checks if appropriate checks are put around the code parts where de-referencing occurs.
- **Context**: as for the helper functions type checking, the eBPF programs cannot receive a context pointer that is not suited for the running program. For example, a program attached to a writing syscall cannot receive as a context a packet structure.
- **Completion**: an eBPF program needs to run to completion in order to not consume resources indefinitely. There is a limit on the total number of instructions a program can process. This encompasses the cases where the program contains a never-ending loop.
- **Instructions**: the verifier checks that there is no invalid instruction inside the bytecode. However, unless writing eBPF bytecode by hand, there is no reason this exception would trigger.

2.4 eBPF Hooks

To execute, eBPF programs need to be attached to hooks in the kernel. In the C code, this is specified using the `SEC()` macro defined in the BPF library. Depending on the eBPF program's type, the events it can attach to are very different. For every attachment type, the eBPF program has access to a different context containing specific fields tailored to the operational boundaries of the target kernel subsystem. Here is a non-exhaustive list of the attachment types eBPF programs can be hooked to.

2.4.1 Networking

The hooks used for networking programs are located at various layers for the networking stack:

- XDP (`"xdp"`): eBPF programs can be directly attached to the NIC, at the earliest possible point of packet arrival on the system.

- Sockets: the programs attached to the sockets can drop payloads at the socket layer.

2.4.2 Tracing

eBPF programs can also be used for observability or performance analysis and thus need to be attached to parts of the kernel code:

- Kernel probes (`"kprobes"`/`"kretprobes"`): the programs can be attached to almost every kernel functions. Every time this specific piece of code is executed, the eBPF program is triggered. The difference between the kprobes and the kretprobes is that the kprobes are used for the kernel functions entry and kretprobes at the exit.

- User-space probes (`"uprobes"`/`"uretprobes"`): instead of being attached to the kernel code, the eBPF programs can be attached to user-space functions. However, developers need to consider the different versions of the user-space programs and the different libraries that cannot be present on every system.

- Tracepoints (`"tp"`): these are stable marked locations inside the kernel code. The list of tracepoint is available on every system.

2.4.3 Security

Finally, eBPF programs can also be attached to security-sensitive kernel operations such as a file creation, a module loading or a network socket operation to implement MAC. All of these security functions are contained inside LSM framework.

Chapter 3

Background on Spin Model Checker

3.1 Model Checking

Model checking is a formal method for verifying concurrent programs. A model checker verifies a program by analyzing its state space and is opposed to theorem proving where the construction of a mathematical proof is needed to verify a model. The model checker checks all computations of a program and formally proves that they are all correct, or gives a counterexample if the correctness of the program is violated.
According to [3], the way a model checker works is the following. From a given mathematical model of the program that needs to be analyzed, it builds a non-deterministic finite automaton (or NDFA) containing all the possible states of the program. By giving the model checker a correctness claim (what should not happen during the execution of the model), it builds a second NDFA from its negation. The goal of the model checker is to find a path or a computation that satisfies both NDFAs. If no computation is found, then the model verifies the correctness claim, otherwise the model checker gives us that counterexample: the trace of the order in which the states of the model have been reached.
Several approaches can be used to search through the states of the NDFAs such as breadth-first or depth-first algorithms. The latter is preferred because it uses less memory.

Model checkers are used in systems where the actual content of the data is immaterial. For example, we verify that a message is integrally received over a network, not that it does not contain security-breaching payloads. The model used for this verification is typically a high level description of the real system, written in formal

specification languages. As [4] explains, among the design flaws the model checkers can be used to find, we can find:

- Deadlock: two or more concurrent processes are blocked indefinitely, each waiting for a resource held by another.
- Livelock: two or more concurrent processes continually change their internal states in response to each other without making actual progresses.
- Starvation: a process cannot access a critical resource, due to its low priority
- Under specification: discovery of an unexpected behavior of the program.
- Over specification: discovery of dead code, a part of the program that is never accessed.

The correctness properties used to discover these design flaws are either safety properties (to verify that nothing bad for the system never happens), or liveness properties (to verify that an expected behavior will eventually be triggered).

3.2 Spin Model Checker

The Simple Promela Interpreter (or SPIN) is an open source model checker developed in 1989 in the Unix group of the Computing Sciences research Center. It was awarded in 2002 the ACM System Software Award, which is a prize awarded for software systems that has lasting influence. This is one of the leading verification tools used by professionals.
SPIN takes a system specification written in Process Meta Language (or Promela), which has semantics close to the C or the Java programming languages. SPIN does not perform the model checking itself: it generates a highly optimized model checking program in C given a model and its correctness claims. The user needs to compile this code and execute it to actually show the results of the verification. The **Figure 3.1** shows the different steps to execute in order to launch the formal verification of a model.

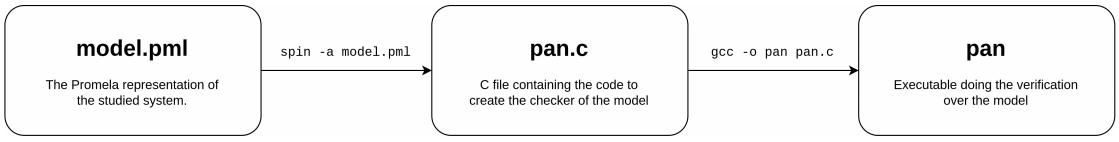


Figure 3.1: Commands to perform the model checking

The Promela language allows the users to determine processes that are executed concurrently, with independence of speed of behavior, alongside channels that

simulate the transmission of data from processes to others. The user can also group statements into an atomic sequence to execute all of them in a single step, mimicking atomic operations used in C code for example.

3.2.1 Correctness Properties

The correctness properties can be of type:

- deadlocks: the model does not end.
- assertions: use the keyword `assert()` to assert that some property needs to be valid at a certain point of the execution.
- unreachable code: the NDFA does not reach a certain state.
- linear temporal logic (or LTL): write logical operators, combining propositional logic and temporal operators such as $\Box$ (`[]` in Promela) for "always", $\diamond$ (`<>` in Promela) for "eventually" and $\mathbf{U}$ (`U` in Promela) for "until". [5] describes more deeply how LTLs work in the context of SPIN.

3.2.2 Monitoring the Model

There are special other ways to observe the system's behavior and to verify if properties are satisfied during the state-space exploration. These are called *monitors* and act as watchdogs that run alongside the system model to verify the defined rules. From [4], there are several forms of monitors in SPIN:

- Monitor Process: this is a Promela proctype that only contains an assertion over the property to check. To avoid adding numerous states to the model execution, we can create a *Guarded Monitor Process*, which executes the assertion only when the property to check is not true: `!P -> assert(P)`.
- Never Claims: it uses the `never` block included in the SPIN documentation. SPIN creates a Buchï automaton and makes it run alongside the model in an asynchronous way. The property written inside the block should never be true.
- LTL Properties: SPIN automatically negates the LTL claims and turn this negation into a never claim.

3.2.3 Execution Modes

SPIN has different execution modes:

- Random simulation: the model checker uses a RNG to resolve non-determinism transitions.
- Interactive simulation: the user chooses the instructions to be executed. This can be useful to simulate scenarios that are very unlikely to occur.
- Verification: in this execution mode, the whole state space is searched. If a violation is found, a trail of the computation can be shown.

Chapter 4

Thesis Objectives

4.1 Motivation

As presented in the background knowledge for the eBPF subsystem, the eBPF verifier ensures that the eBPF program is safe to execute by checking a certain set of rules. With static safety checks, it can ensure the memory bounds of the program, verify the type of the program matches with the helper functions' types or prevent uninitialized reads. However, this verification is only designed for sequential program analysis: while the verifier checks the security of each program individually, it cannot capture concurrency issues that could arise between multiple concurrent programs such as race conditions or dynamic temporal state interactions.
Since eBPF can be used to share information between kernel programs or user-space applications, the fact that the verifier cannot ensure the safety of concurrent program can lead to security issues. As an example, if two programs retrieve and update a global counter without atomic operations, they can overwrite each other's value, resulting in a corrupted tracking variable. In this case, the verifier evaluates each eBPF program independently and sequentially at load time, and since each program contains valid instructions and terminates correctly, the verifier will not detect any issue and will not throw any error, leaving the race condition completely unaddressed.

The primary objective of this thesis is to use formal modeling and the SPIN model checker to analyze and verify the security issues that could arise in the case of the use of helper functions in concurrent eBPF programs. The focus has been made on the helper functions targeting *ring buffers*. The main helper functions used by the kernel-loaded programs are:

- `bpf_ringbuf_reserve`: reserve a memory slot in the ring buffer, to be used by the program to store data inside.

- `bpf_ringbuf_commit` (further used in the `bpf_ringbuf_submit` and `bpf_ringbuf_discard` helper functions): decide what to do with the reserved chunk of data, either we submit it inside the ring buffer, or we discard it, freeing the chunk in the buffer.

Further information and explanations will be given on these functions later in the development of the thesis work.

4.2 Specific Objectives

The primary goal of the work can be broken down into 4 specific objectives:

- Abstraction and Modeling: to apply formal verification to the use of helper functions, a high-level abstraction of their core behavior is needed. The content of the data is not modeled since formal verification methods are not suited for this: only the main mechanics are kept in the model. The more detailed the model is, the bigger the state-space will be, possibly reaching the capacity limits of the software or the machine the model checker is run on. This is why a careful selection is required to create an abstract but realistic model. This abstraction is then translated using the Promela modeling language.

- Properties Specification: as explained in the background section for the model checking, a verification requires a model and one or more correctness claims. In our case, the correctness claims will be security claims, representing a behavior from the model that could lead to the compromise of the CIA (Confidentiality, Integrity, Availability) triad. Once these security invariants are defined, they are integrated inside the Promela model as assertions, LTLs or other structures defined earlier.

- Analysis and Verification: once the model or the models are complete, the SPIN model checker is used to uncover possible security violations. The goal of the work is to use SPIN capabilities to explore all possible execution paths. Two outputs are possible: either the model checker outputs that the claims are correct, or that it has found a way to break them. In the latter case, it generates a trail file, giving in great details all the steps taken by the model to reach the error.

- Validation and Reproduction: because SPIN will explore all the transitions of the created NDFAs, it is possible that the trail it has computed cannot be reproduced in a real-work scenario. Hence, it is crucial to try to replicate the error trail in a contained eBPF program to see if the security violation is a true positive or not. Once an unintended behavior is replicated, it is

interesting to think about crafting an attack on an eBPF commercial tool that uses the ring buffers such as a runtime security tool.

4.3 Scope and Contributions

4.3.1 Focus on Ring Buffers

At the date of the work, more than two hundred helper functions are defined in the kernel. This is why this work has been narrowed down to the helper functions that are specific to the use of ring buffer maps, which reduces the number of target functions to less than ten. As explained before, the verifier acts as a static checker for sequential programs, this is why the scope of the work is to study concurrency between eBPF programs. The model focuses on multiple programs accessing the same ring buffer at the same time: the state-space exploration of the SPIN model checker can spot race conditions between these programs the verifier cannot detect.

4.3.2 Initial C-model Detour

Before designing an abstract but realistic model of a system containing one ring buffer and several eBPF programs accessing it concurrently, I attempted a naive and exhaustive Promela model of the kernel C code of the helper functions and its baseline behavioral invariants. This led to a massive state-space explosion and consequently to a failure in retrieving meaningful security insights about the studied system.
This primary exploration served as a vital methodological lesson: it highlighted why high-level modeling was essential to model concurrency systems. This forced me to shift toward a cleaner and abstracted model, capable of verifying critical security claims in a concurrent environment.

4.3.3 Chapters Roadmap

- Chapter 5 (Preliminary Exploration and Initial C Code Modeling): explanation of the direct translation of the kernel C code into a Promela model, the behavioral invariants used for the runs, and the state-space explosion that emerged from this model.

- Chapter 6 (Abstracted Model and Security Invariants): refined abstract model focusing on the core behavior of the functions, the security invariant used for the verification.

- Chapter 7 (Execution and Verification Framework): analysis of the trail files generated by the SPIN model checker for any error it has found and replication in real eBPF code of the security issues.
- Chapter 8 (Real-World Case Studies): application of pointers de-synchronization over real-world programs and tools to see what the outcomes are.

Chapter 5

Preliminary Exploration and Initial C Code Modeling

Before diving into the subject, here are the versions of the software at the time used for this work:

- Linux: `7.0.0-29-generic`
- Clang: `Ubuntu clang version 18.1.3 (1ubuntu1)`
- GCC: `gcc (Ubuntu 13.3.0-6ubuntu2 24.04.1) 13.3.0`
- SPIN: `Spin Version 6.5.2 - 18 September 2025`
- Libbpf: `1:1.3.0-2build2`

5.1 Useful Functions

5.1.1 Ring Buffers

Ring buffer maps are a special type of map that have some restrictions. For example, lookup, update and delete basic helper functions cannot be used with the ring buffers: there is a set of helper functions specially designed for them.
A ring buffer contains 3 main data sections defined with the `mmap` syscall. The first two are used to represent the useful pointers to the buffer and the last one is the data. The pointers are represented as 64-bit unsigned integers.

- The first pointer is the *consumer* pointer. It indicates the user space which slot of the ring buffer to read and get the data from. Every time the consumer program reads data from the ring buffer, it updates this pointer to read the next data slot. The user-space can read and write in this memory area.

- The second pointer is the `producer` pointer. It is useful for the reserve function: it points to the oldest available slot to give to an eBPF program. This is updated by the reserve function, if the reservation is successful, pointing to the next slot available. For the user-space, this data page is only readable since this is a critical pointer for the ring buffer.

- The producer pointer is not mapped alone: the size argument in the `mmap` call is larger than the pointer size. Indeed, this call also maps the data pages for the actual information held by the ring buffer. In the ring buffer, each data piece is preceded by a header of eight bytes. The first four bytes are used to store the flags (`BPF_RINGBUF_BUSY_BIT` if the slot is not yet committed and `BPF_RINGBUF_DISCARD_BIT` if the slot has been discarded) and the size of the actual data, while the last four are used to store the page offset, used only by the kernel.

When reading the code for the reserve function, a third pointer is present: the *pending* pointer. Its main purpose is to manage concurrent data reservation by tracking the furthest extent of data slots that have been reserved but not yet submitted or discarded.

5.1.2 Helper Functions

Before jumping into the development of the model, we need to choose what helper functions we are going to model. By focusing on maps, we reduce the number of helper functions to about 40. Even by narrowing the study to the use of the ring buffer maps, there are still ten of them. However some of them focus on the use of *dynamic pointers*. For short, dynamic pointers carry metadata about the memory region they bind. Their handling is only done by the use of helper functions, otherwise the verifier would reject them. These metadata can help perform security checks at runtime.
This leaves us with only 5 helper functions used for the handling of ring buffers. Here are their information taken from the kernel C code located in `linux/kernel/bpf/ringbuf.c`:

```
1. void *bpf_ringbuf_reserve
(void *ringbuf, u64 size, u64 flags)
```

- Summary: reserve a slot in a ring buffer map for the program and returns the pointer to the granted data slot in case of success. Otherwise, `NULL` is returned.

- Arguments:

  - `struct bpf_ringbuf * rb`: a pointer of the ring buffer the reservation is made in
  - `u64 size`: the desired size of the data slot to reserve
  - `u64 flags`: flags for the reservation. As of now, the flags for the reservation need to be set to 0.

- Return: `void *`, the pointer to the data slot of the ring buffer.

```
2. void bpf_ringbuf_submit
(void *data, u64 flags)
```

- Summary: submit a previously reserved data slot in a ring buffer so that it can be consumed by user space. This function is a wrapper over the function `bpf_ringbuf_commit`.
- Arguments:
  - `void * data`: a pointer to the data slot returned by a reserve function call
  - `u64 flags`: flags for the submission operation. T A flag can be set to notify the user space that data has been inserted in the ring buffer.
- Return: `void`.

```
3. void bpf_ringbuf_discard
(void *data, u64 flags)
```

- Summary: discard a previously reserved data slot in a ring buffer, freeing it without notifying user space. This function is a wrapper over the function `bpf_ringbuf_commit`.
- Arguments:
  - `void * data`: a pointer to the data slot returned by a reserve function call
  - `u64 flags`: flags for the discard action operation.
- Return: `void`.

```
4. long bpf_ringbuf_output
(void *ringbuf, void *data, u64 size, u64 flags)
```

- Summary: copy data from a source pointer directly into a ring buffer map in a single atomic operation and notify user space. Basically this function combines the reserve and submit operations.

- Arguments:
  - `void * ringbuf`: a pointer to the ring buffer map
  - `void * data`: a pointer to the data to be copied into the ring buffer
  - `u64 size`: the size of the data to copy
  - `u64 flags`: flags for the output operation
- Return: 0 on success, or a negative error code in case of failure.

```
5. u64 bpf_ringbuf_query
(void *ringbuf, u64 flags)
```

- Summary: query metadata or state characteristics of a ring buffer map (such as availability or number of unconsumed data items).
- Arguments:
  - `void * ringbuf`: a pointer to the ring buffer map
  - `u64 flags`: specifies what information to query such as available data or the size of the ring buffer
- Return: a `u64` value representing the queried metric or status.

5.1.3 User-space Functions

For a realistic model of the concurrent use of ring buffers, user-space functions are also needed. These functions are defined in the *libbpf* library, in the `tools/lib/bpf/ringbuf.c` file. There are two main ways to consumer data from the ring buffer

```
1. int ring_buffer__poll
(struct ring_buffer * rb, int timeout_ms)
```

- Summary: polls the underlying ring buffer file descriptors for available records, blocking up to a specified timeout, and automatically triggers the user-defined callback function for each received record.
- Arguments:
  - `struct ring_buffer * rb`: a pointer to the ring buffer.
  - `int timeout_ms`: the maximum time to wait in milliseconds (-1 to block indefinitely, 0 for non-blocking execution)

- Return: the total number of records processed (>0), or a negative error code on failure.

`2. int ring_buffer__consume`
`(struct ring_buffer * rb)`

- Summary: checks the ring buffer memory directly and reads data if some are available in the ring buffer.
- Arguments:
  - `struct ring_buffer * rb`: a pointer to the ring buffer.
- Return: the total number of records processed (≥ 0), or a negative error code on failure.

The main difference between these two functions is that the poll function will sleep until some data is available to read in the ring buffer, while the timeout is not over. The consume function will try to directly read data from the ring buffer. If no data is available, then it returns 0.

5.1.4 Kernel and User-space Programs

To model the actual eBPF and user-space programs, we need to model proctypes that use the helper functions we just modeled. All of the following content has been put in the Promela model called `model.pml` inside the `Code/Spin` sub folder. There are three models present but their presence will be explained later.

For the user-space program, we model an infinite loop that just poll the data from the ring buffer, using a `do::true` loop, which is the Promela equivalent of a `while(1)` in C.

For the kernel side, we use several proctypes that have a different behavior. They still share one particularity: they all run inside a `do::true` loop to simulate a continuous flow of triggering eBPF events.

- The `low_priv_ebpf` proctype is doing basic reserve and submit operations on the ring buffer. The pointer returned by the reserve function is modeled as an offset set to -1 and passed as argument inside the helper function to indicate which slot has been reserved for this program.
- The `delayed_ebpf` proctype is simulating an eBPF program that is launched after a certain time. However this is not really relevant as the SPIN model checker will run by checking the whole state-space, which means that the wait can already be simulated by the verification.

- Finally the `conditioned_ebpf` proctype use non-determinism to choose between a submit and a discard operation on the ring buffer.

5.2 Native C modeling

This section refers to the files called `model_big.pml` and `model.pml`. As mentioned before, the first approach was to translate C code from the kernel source into Promela. For this, I have modeled the ring buffer data structure with its pointers, the spinlock used for the reservation function but also the actual data slots, with their headers and payloads. Every global variables such as the `BPF_RINGBUF_BUSY_BIT` have been kept from the source. The fact that the ring buffer's pointers are increasing indefinitely through the execution of the programs has been modeled as well since the goal of this approach was to be as loyal as possible to the source code. Almost the whole C code of the presented functions has been kept, only the use of the `pg_off` field in the headers has been removed since it is not useful in our case of model checking.

5.2.1 Core Translation

What is represented as atomic functions in the C code such as the reservation of the ring buffer's spinlock is translated in Promela with the use of the `atomic` block. This block is used when a code sequence needs to be executed as one indivisible unit, without being interleaved with other processes.
Since Promela offers a syntax very similar to the C language, the ring buffer structure has been re-used in Promela, including only the essential fields such as the spinlock, the consumer position, the producer position, the pending position and the data array.

5.2.2 Execution Environment

To create running processes in Promela, 2 methods can be used. The first method is to use the keyword `active` before the `proctype` just defined. The other is the use of the keyword `run` inside another process block. When using the active method, the process can be run instantly by the model checker, depending on the mode it is running. In the second case, the start of the process is triggered when the model checker executes the line that contains the run keyword.
In this case, we use the `init` block which is supposed to contain every initialization for the global array variables. Since we want to use the whole state-space exploration by the SPIN model checker, it is possible that some active proctypes are launched before the init block, skipping entirely the initialization process. It is problematic if some variables are initialized with special values such as -1. To patch this issue,

we need to use the second method and use the keyword run for the processes at the end of the init block.

5.3 eBPF Verifier Model

5.3.1 Verifier Checks

Since the goal is to find security issues with eBPF programs that are valid, which means that they pass all the verifier checks, we need to find a way to represent it in the Promela model. From the checks described in the background chapter, only some of them are useful to us:

- Helper Functions Type: Not Needed. The helper functions we use are valid for any type of eBPF programs.
- Helper Function Arguments: Not Needed. The arguments passed inside the helper functions is already controlled in the model and not chosen in a non-deterministic way.
- License: Not Needed. The license has no sense in the Promela code.
- Memory Access: Not Needed. The ring buffer structure's pointers does not represent its actual bounds and since they are indefinitely increasing, there is no rule on them.
- Pointer Referencing: **Needed**. This is a big deal for ring buffer programs in eBPF. There are several rules that an eBPF program using this structure needs to follow.
- Context: Not Needed The context is not represented as the model does not receive any.
- Completion: Not Needed. The model checker already has a completion verification when running a model.
- Invalid Instructions: Not Needed. We are not writing raw eBPF bytecode so these checks does not need to be present.

As we can see, what we can represent as the verifier in the Promela model are the checks done to the pointers' life cycle. The verifier will reject every attempt to:

- Double Release: a reserved slot cannot be submitted or discarded more than once (`R1 type=scalar expected=ringbuf_mem`). Obviously, in this case we can also put the double discard or the discard after submit or lastly the submit after discard.

- Reference Leak: if a slot is reserved at some point in the program, it has to be submitted or discarded later (`BPF_EXIT instruction in main prog would lead to reference leak`).
- Use After Submit: if a pointer is submitted inside a ring buffer, any attempt to modify it after will be caught by the verifier (`R6 invalid mem access 'scalar'`).
- Pointer Offset: if we try to modify the pointer between a reservation and a submission, an error will be thrown (`R1 must have zero offset when passed to release func or trusted arg to kfunc`). This is not present inside the model because I'm working with slot indexes instead of real byte-by-byte pointers, so this is not ideal to be represented inside the Promela model.

Examples of eBPF programs that replicate these errors can be found in the code material under the `Code/Verifier` directory.

5.3.2 Verifier Model

For this naive approach of modeling the C code in Promela, I have chosen to model the verifier by tracking the state of each slot of the ring buffer. For this purpose, we can use the `mtype` data type, which is useful to declare symbolic names for constant values.

```
mtype = { FREE, RESERVED, DISCARDED, SUBMITTED, CONSUMED };
```

All chunks start in the `FREE` state, then when the reserve function is used, they are in the `RESERVED` state. After a submit/discard call is made on that chunk, it passes in the `SUBMITTED/DISCARDED` state. Finally, if the user-space reads the data, it switches in the `CONSUMED` state. The consumed state is equivalent to the free state, which is why in the reserve function, we treat them as the same state.

To model the verifier rejections inside the model, I first chose to use LTLs. The way it works is to declare global boolean variables at the top of the model to false, update the value when a the invariant gets violated and declare the LTL at the end of the model. The LTL at the end of the file looks like this:

```
ltl ltl_name { [] !bool_variable }
```

In terms of LTL logic, this literally means "the `bool_variable` is always evaluated to false". This means that when the boolean value is updated to true, the LTL becomes violated and an error from the model checker is thrown. To catch the behavioral invariants, here is how I proceeded inside the Promela model:

- Double Release: inside the commit helper function, before doing the actual submission or discard of the data slot inside the ring buffer, it checks in what state the slot is: if it is in the submitted or discarded state, then the boolean value is updated.
- Resource Leak: this one is very different from the other LTLs checks because of the syntax of the verifier invariant: if a chunk of the ring buffer is reserved, then it has to be submitted or discarded. This is translated in LTL syntax as

  ```
  slot_reserved -> <> (slot_submitted || slot_discarded)
  ```

  However this has to be done for every slot of the ring buffer, which means that the more slots there are, the more LTLs we have to write. To avoid this constraints, we only check for the first slot of the ring buffer.

5.4 Behavioral Invariants

Now the model contains the helper functions from the kernel side to reserve and commit the data, the user-space functions to poll the data from the ring buffer and the verifier to check that the programs would pass the verifier checks, we need to implement the behavioral invariants that the programs should not break and that the SPIN model checker should catch for us. Here are the behavioral invariants I have found and their implementation inside the model.

5.4.1 Concurrency & Race Conditions

Overlapping Reservations

We need to ensure that after some iterations of the functions two different eBPF programs do not receive the same slot or at least a portion of the same slot for submission. For this, we keep track of the bytes of the ring buffer that are reserved by a program by saving the PID of the program. When a program reserves a chunk of data inside the ring buffer, we make sure that the slots reserved are not already inside this tracking array. When a chunk is submitted, then the reservation is done and we need to free the slots in the tracking array.
As explained before, the catch of these invariants is done by the declaration of a false boolean at the top of the model and a LTL at the end. Whenever the wrong behavior is detected inside the model, the boolean variable is updated and the program throws an error.
To track this issue, we create a `byte_owner` array that will store the PIDs of the program that reserves a chunk of data. When reserving a slot, we loop through all

the length of the data chunk and we verify that the array slot has no owner (PID of -1). If it is free, then we update the value for the next reservations, otherwise we raise the error. As mentioned before, in the commit function we free the entire chunk in the tracking array by setting the PID to -1.

Spinlock Bypass

The reservation helper function of the ring buffer is protected by a spinlock: the program enters the functions and tries to acquire the spinlock. If it is free, then it proceeds with the reservation and releases the spinlock. Otherwise, it returns `NULL`. We must ensure that the use of this spinlock does not block the programs.
This invariant does not need any LTL to be caught because a deadlock in the program will simply be found by SPIN by raising the fact that the model is caught in a never-ending loop.

5.4.2 Data Integrity & Boundary Quirks

Header Corruption

As mentioned in the background chapter on eBPF and the ring buffer structures, every data chunk has a header before its payload, containing length and flags information. Any corruption on this header could break the whole ring buffer.
To catch this, we need to create another array that keeps track of the header values when they are created. Whenever the header is updated or used in some way, we compare the actual value with the one stored in the additional array and if a difference is found, we raise the error. To avoid silent failures, we need to check the whole array of headers when used to ensure no header has been corrupted.
The header array is updated when

- the header is updated with the busy bit inside the reserve function
- the header is updated with the removal of the busy bit and the discard bit inside the commit function

The header array is used to perform the verification once for every function call (reservation, commit and polling). In the verification, we check byte by byte the values stored inside the comparison array and raise the error whenever a difference is found.

Wrap Around

When the ring buffer is reaching the end with the structure pointers such as the producer pointer, it comes back to the beginning of the ring buffer to reserve space.

We need to ensure that this back-to-the-beginning reservation does not hinder active data in the ring buffer.
This catch is done by verifying that the reserved chunks are not in the reserved or submitted state. The only valid states are free, discarded and consumed. This is done by placing a looping check inside the reservation function.

Pointer Ordering

During the whole running of the model, the pointers of the ring buffer need to follow a strict ordering: $cons_pos \leq pend_pos \leq prod_pos$. If this ordering rule is not respected, this might break the checks done by the helper functions to reserve and commit data inside the ring buffer and trigger issues within the data structure. Checking for this error is pretty easy: the pointers' values are put inside the LTL in the right order. Any disorder will trigger this LTL. However there is one catch: since the pending pointer of the ring buffer is specially used inside the reserve function, during the poll operation the consumer pointer can be ahead of the pending pointer. We then define a transient gap allowing this behavior. The final LTL for this issue is

```
!transient_gap -> cons <= pend && pend <= prod
```

5.5 Execution

The model mentioned are located in the `Code/Spin` sub folder.

5.5.1 Steps

Here are the basic steps to run a Promela model with the SPIN model checker.

1. Generate a verification model

   ```
   spin -a model.pml
   ```

   This will generate the C code for the model checker in a C file called `pan.c`.

2. Compile the verifier into an executable

   ```
   gcc -O2 -o pan pan.c
   ```

   This will turn the C code generated by SPIN into an executable to actually perform the checks. The executable is called `pan`. More flags can be added as it is using the `gcc` compiler, depending on special encountered conditions.

3. Run the executable

```
./pan
```

This will execute the model checker on the Promela file given at the beginning. The results will be printed inside the terminal.

There is also GUIs developed by the SPIN creators or by the community that can ease the model checker execution, if using CLI feels uncomfortable. However when running these commands on the model we just created did not generate the wanted results.

5.5.2 Execution Scripts

To automate and avoid introducing errors during the numerous steps of the models' execution, I wrote a bash script that gather all of the crucial steps mentioned in the previous subsection. For this first and naive approach, since every invariant was written as a LTL in a single Promela model, the script did one compilation of the model, then ran it once per property with the right LTL name. It can be found in the directory as `exec.sh` and its content can be found in the Appendix A, **Figure A.1**. The script execution is

1. create the C file from the spin model checker
2. compile the C file into a binary
3. for every LTL, execute the binary file with the flags `-N prop` to check the specified LTL

5.5.3 Issues Encountered

Fairness

Because of the way the state-space exploration works with SPIN, it is possible that it only executes one proctype in a loop and blocks every other one. This is not what we want to verify with the model checker. To prevent this and permit to the other proctypes to be executed as well, we need to add the `-DFAIR` flag to the `gcc` compilation. The executable needs to be given the `-f` flag to run with this. The flag needs to be set to the number of processes we want to enable weak fairness to.

State Explosion

This is the main issue of the model. The model is really big (more than five hundred lines) compared to "classic" models used for model checkers. Moreover, the naive approach introduced a lot of overhead in the model that leads to the creation of a

really huge state space because of the fact that the model checker will try every possible execution path, interleaving the program in all the possible ways:

- a lot of proctypes: adding one proctype will increase significantly the state space
- infinite loops: since the kernel and the user-space programs are executing on infinite loops, this introduce a lot of different states for the model checker to store and check, leading to a significant increase in the running of the model
- nested loops: same issue as the number of proctypes
- non-deterministic branches: this double the state space every time the execution hits this
- increasing pointers: as we mimic the real kernel C code, we represented infinitely increasing pointers. This, as the infinite loop creates a lot more possible states.
- use of LTL: LTL formulas are transformed into never claims and Büchi automaton which require nested search
- variable size: the size of the arrays (driven by the size of the ring buffer) and also the size of the declared variables (`int` or `short`) increase the execution time.

5.5.4 Results

Since SPIN can only check one LTL at a time, we need to execute the model checker several times, once for every LTL formula present in the model. This is done by using the `-N` flag to the `./pan` command with the name of the LTL to verify.

```
gcc -O2 -o pan pan.c
for prop in $SAFETY_PROPERTIES; do
    echo ""
    echo "================================================"
    echo " VERIFYING: $prop"
    echo "================================================"
    ./pan -a -N "$prop"
done;
```

This code is supposed to give us the result for each LTL of the model, stored inside the `SAFETY_PROPERTIES` variable. As explained in the issues encountered before, we have several issues when running the model with this bash script. Running this script will give us the error

```
error: max search depth too small
```

To try to circumvent this, we can use the `-DBITSTATE` flag during the compilation of the model. Initially, when the model checker executes the model, it checks every possible state. For this, it needs to store them to verify that it does not check the same state twice. This might take a lot of space and a lot of execution time. Using this specific flag will change the way the model checker stores the states and massively reduces the memory usage. The model checker uses a bit-vector hashing scheme: each state is hashed into a small number which represent the state's position in a huge state vector. When the model checker visits a state, it sets the bit of the state's position to 1 so it knows that it should not visit it again. The size of the array can be controlled with the `-w` flag when running the created executable, which represents the power of two of bits to use. However using this flag transforms the states exploration probabilistic: the hash function used to represent the states can introduce collisions. If such a collision occurs, then a state that has not been explored yet might be considered as already checked by the model checker.

5.5.5 Final Model

Even with these restrictions on the execution mode, the model is too big to get results: each LTL verification takes too much time to complete. To reduce the model's size, we have to do some modifications according to the reasons why the state-space might explode. As a consequence, I have created another different model called `model_simple.pml`. Here are the main modifications made:

- bound the loops of the producer and consumer proctypes
- use `byte` instead of `int` for the positions of the pointers
- reuse the `chunk_owner` array and remove the `byte_owner` for the overlapping reservations tracking
- reduce the number of proctypes for the execution
- changes the LTLs that can be optimized to reduce drastically the number of states.

Monitors

I have mentioned the monitors in the background chapter on the SPIN model checker. They are these small proctypes that run in parallel of the other proctypes of the model and that catch the same invariants as the LTLs. This is possible where the LTL as the following form:

```
ltl rule { [] !bool_var }
```

The monitor that will be replacing this LTL is a Guarded Monitor Process that will have this form:

```
monitor {
    bool_var; assert(!bool_var);
}
```

This will trigger whenever the boolean predicate `bool_var` will evaluate to false. This syntax is used to block the execution as long as the first line predicate is false. When it evaluates to true, the execution of the block can continue. In this case, we can translate the behavior of the monitor as the following: while `bool_var` is false, do nothing and when it becomes true, execute the assertion. In our case the assertion will automatically execute to false and throw the error because of the value of the boolean variable.
In our case, the rest of the code remains unchanged because we still need to update the variable whenever the bad behavior is detected. Only the catching mechanism changes and with a little of boolean logic, we understand that this monitor will do exactly what the replaced LTL was doing.
Because monitors are processes that are triggered within the `init` block, they cannot be chosen as the LTLs were in the execution script. For this reason, I have created a directory inside `Code/Spin` called `Simple` where I have put the new models and the new execution script.

To keep the same easy way to execute all the verifications within the same simple bash script, I used macros in promela, especially the `#ifdef` and `#endif` pair that permits the use of custom flags during the execution to activate or not some parts of the code. The bash script used to execute the checks can be found in the Appendix A, **Figure A.2**.

5.5.6 Outputs

Here is a sample of the output from the execution of the first LTL "double submit":

```
================================================
 VERIFYING: HEADER_CORRUPTION
================================================
warning: no accept labels are defined,
so option -a has no effect (ignored)
Depth=     205 States=    1e+06 Transitions= 3.53e+06
    Memory=  4103.160 t=     1.46 R=   7e+05
Depth=     205 States=    2e+06 Transitions= 7.26e+06
```

```
    Memory=  4103.160 t=     3.01 R=   7e+05
Depth=     205 States=    3e+06 Transitions=  1.1e+07
    Memory=  4103.160 t=     4.57 R=   7e+05

(Spin Version 6.5.2 -- 18 September 2025)

Bit statespace search for:
 never claim          - (none specified)
 assertion violations +
 acceptance   cycles  - (not selected)
 invalid end states +

State-vector 388 byte, depth reached 205, errors: 0
  3319206 states, stored
  8891759 states, matched
 12210965 transitions (= stored+matched)
   333066 atomic steps

hash factor: 10351.8 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
 1291.500 equivalent memory usage for states
    (stored*(State-vector + overhead))
 4096.000 memory used for hash array (-w35)
    0.763 memory used for bit stack
    6.104 memory used for DFS stack (-m100000)
 4103.160 total actual memory usage

...

pan: elapsed time 5.09 seconds
pan: rate 652103.34 states/second
```

As we can read on the output of the execution commands, several interesting metrics are displayed to the user:

- State Vector (bytes): memory required to store a single system state, made of the different variables of the model or the buffers. It is copied at every branch point.
- Depth (no unit): maximum length of execution steps from the initial state

- States Stored (no unit): breadth of the state space explored, number of distinct path evaluated during the search
- Time Elapsed (seconds): time needed to complete the verification
- Search Rate (states/second): throughput of states evaluated per second

There are more lines than that, explaining which states has not been reached, as for each LTL verification the model checker does not need to reach all the states. One essential line is the one containing the "`errors:0`", which obviously tell us how many errors the model checker have found when testing all the assertions, LTLs and monitor processes. To clean up the outputted results, we can grep the execution to this line:

```
./pan -a -m100000 -w35 | grep errors
```

Which gives us in the end:

```
=========================================
COMPILING MODEL: model_simple.pml
=========================================

================================================
 VERIFYING: DOUBLE_SUBMIT
================================================
State-vector 388 byte, depth reached 205, errors: 0

================================================
 VERIFYING: SUBMIT_AFTER_DISCARD
================================================
State-vector 388 byte, depth reached 205, errors: 0

================================================
 VERIFYING: USE_AFTER_FREE
================================================
State-vector 388 byte, depth reached 205, errors: 0

================================================
 VERIFYING: RESOURCE_LEAK
================================================
ltl rule_no_resource_leak: [] ((! ((chunk_state[1]==RESERVED)))
|| (<> (((chunk_state[1]==SUBMITTED))
|| ((chunk_state[1]==DISCARDED)))))
```

```
State-vector 380 byte, depth reached 404, errors: 0

================================================
 VERIFYING: OVERLAPPING_RESERVATION
================================================
State-vector 388 byte, depth reached 205, errors: 0

================================================
 VERIFYING: HEADER_CORRUPTION
================================================
State-vector 388 byte, depth reached 205, errors: 0

================================================
 VERIFYING: WRAP_AROUND
================================================
State-vector 388 byte, depth reached 205, errors: 0

================================================
 VERIFYING: ORDERING
================================================
ltl rule_ordering: [] ((! (! (transient_gap)))
|| (((rb.consumer_pos<=rb.pending_pos))
&& ((rb.pending_pos<=rb.producer_pos))))
State-vector 388 byte, depth reached 404, errors: 0
```

There is actually a warning line for every invariant that has been transformed into a monitor process because the flag `-a` is only used when checking for acceptance cycles, that only occur when a LTL is used. I removed it for better visibility of the results.
As we can read from the output I pasted before, we have 0 error for each invariant check.

We can however recapitulate the important metrics for each rule inserted inside the model, here is a table (for the units, refer to the previous description of the metrics):

5.5.7 Conclusion

This finally gives us 0 error for every LTL we defined in the model. However, as I explained earlier, this was not security invariants, but only behavioral invariants for the ring buffer. This does not highlight real security issues. Also the model used was very big and it gave me a hard time to actually execute and get real

Property	**State-vector**	**Depth**	**Stored**	**Time**	**Rate**
DOUBLE_SUBMIT	388	205	3.3e+6	4.98	6.7e+5
SUBMIT_AFTER_DISCARD	388	205	3.3e+6	4.95	6.7e+5
USE_AFTER_FREE	388	205	3.3e+6	4.96	6.7e+5
RESOURCE_LEAK	380	404	3.3e+6	5.14	6.5e+5
OVERLAPPING_RESERVATION	388	205	3.3e+6	4.98	6.7e+5
HEADER_CORRUPTION	388	205	3.3e+6	5.09	6.5e+5
WRAP_AROUND	388	205	3.3e+6	5.04	6.6e+5
ORDERING	388	404	3.3e+6	5.25	6.3e+5

Table 5.1: Naive Model Results

results. This is why I had to think all over again to simplify this model by only keeping the abstracted behavior of the functions and using assertions instead of LTLs for the verification of the model checker.
However this model creation and running was not vain: it permitted to understand better how the helper functions and the ring buffer interact, but also how the model checker was actually verifying the claims of the model. This was used as a baseline for the rest of the work as the base functions stays the same.

Chapter 6

Abstracted Model and Security Invariants

By analyzing in greater details the user-space function polling the data from the ring buffer, I discovered that the eBPF architecture places absolute trust in the consumer pointer's integrity and completely lacks of synchronization mechanism at the user-space layer. Because this lack of protection is a design characteristic of the current libbpf implementation, using multiple consumer programs over the same ring buffer can create severe race conditions over the consumer pointer, leading to silent invalidation of the ring buffer state invariants, potentially introducing bigger vulnerabilities in the use of the ring buffer helper functions.
The official kernel documentation as well as articles from eBPF developers such as [6] describe the `BPF_MAP_TYPE_RINGBUF` as a multi-producer single-consumer (MPSC) map.
Consequently, this work shifts to a protocol-level investigation: analyzing the multi-tenancy of the consumer pointer repercussions over the ring buffer's stability. For this, we can leverage the work done in the previous chapter to construct a lightweight formal model in SPIN. With carefully chosen security invariants, we can uncover exact conditions under which concurrent consumptions may violate the ring buffer integrity as a data structure running in the kernel. To extend the work of multiple consumers over the same ring buffer, I will also cover direct modifications over the ring buffer internal pointers using the *mmap* syscall.

6.1 Abstracted model

For this abstracted model, I kept only the core behavior of the helper functions: no array that hold the actual content of the ring buffer nor the corresponding headers and no bit-wise operations. Only the state of the data slots and the pointers

operations have been written in this model.
One other big change from the big model from the first part is that I removed the infinitely increasing ring buffer pointers and I bounded them to the size of the ring buffer. Instead of going all the way to the limit of their data type, they wrap around thanks to modulus operations whenever they get updated.
The abstracted models are in the `Code/Abstracted` directory. All the model files are explained later.

6.1.1 Helper Functions

Reserve

The model of the reserve function is the following: it acquires the spinlock of the ring buffer, updates the pending pos for concurrent reserve, verify that there is free space inside the ring buffer and if so then proceed to the actual reservation of the slot. Since there is no header representation in this model, I used an array that kept track of the state of the slot, if it is busy or not. The array only contained boolean value, true meaning that the slot is busy and false otherwise.

Commit

There is no actual operation done on the pointers of the ring buffer during the submission or the discard of a data slot, which means the commit function only contains useful code for the verifier model.

6.1.2 User-space Functions

Poll from libbpf

The poll function verifies that there is new data in the ring buffer. If there is then we consume the slot and update the pointer, otherwise we break and leave the function.

Raw access

The raw access to the ring buffer is the following: since the ring buffer memory layout is one page for the consumer position, one page for the producer position and then the pages containing the actual data of the ring buffer, we simply modeled the mmap call with the retrieving of the global variables modeling the pointers of the ring buffer. The mmap representation has to be wrapped inside an atomic block to model it otherwise race issues can happen during the retrieval of the pointers' values.
Then if data is present inside the ring buffer, done also by comparing the values

from the producer and the consumer, it updates the value of the ring buffer's consumer. Since this is a raw access, we can also let the model checker choose between jumps from one to three data slots.

6.2 Rework of the Verifier Model

Instead of using the custom type for the slots' state (free, reserved, submitted, discarded, consumed), I used only verifications at the helper function level.
For this, I created a boolean array that kept track of the state of the slots, called `verifier`. Whenever the submit helper function is called on a specific slot, it updates the value inside the array setting the value to true, meaning that this slot is already committed. The work is not over yet because we also need to check before the actual commit function if the slot was already committed or not (accordingly throwing a double commit error from the verifier). However doing this would trigger verifier errors from the model execution, even when the trail showed a benign use of the ring buffer helper functions. To avoid this, I created a wrapper for the submit function for the check of the actual value of the array, instead of putting it inside the function's content.
This allows us to catch the double commit issues within the programs. For the resource leak, I use the array for the busy state: if the first slot once becomes true, it eventually has to become false

```
[] (busy[0] -> <> !busy[0])
```

This will be the only LTL of the model, which will be verified automatically by the model checker every time it runs. With these implementations, we have simplified the model of the verifier.

6.3 Security Invariants

In order to find security invariants to implement in the ring buffer model, we can think of the CIA triad.

- Confidentiality:
  - Out of Order Read: the data read from the consumer programs might be out of order because of the out of sync on the consumer pointer. This could lead to misinterpretation of the data retrieved from the kernel.
- Integrity:

  - Time of Check vs Time of Use (TOCTOU): the consumer program checks that a ring buffer slot is ready to be read. In the meantime, the ring buffer gets full and wraps around and the slot is reserved again. By the time the consumer actually reads the data slot, the header becomes busy again leading to an unintended data read.
  - Overflow: modifying arbitrary the consumer position, instead of using the libbpf built-in functions to consume data could mis-align the pointer that would be now pointing inside other data slots or in the header of a slot.

- Availability:

  - Deadlock: the consumer pointer is used by the kernel to verify the available space inside the ring buffer. An out of synchronization pointer could lead to miscalculation from the helper function and block the ring buffer because it thinks it is full while it is not.
  - Evasion: updating incorrectly the ring buffer's consumer pointer could skip the read of some data slots from the data structure.

To implement these security invariants inside the model, I have chosen to use one model per invariant and use assertions inside the model to catch them. This way we don't suffer from the LTLs' overhead and the models are lighter to run within the model checker. This introduces more time for the execution since each model has to be compiled individually by the GCC compiler.

To systematically categorize and analyze the previous invariants, we can structure them into a hierarchical taxonomy: The category and subtypes numbers will be

Category	**Subtype**	**Invariant**
1	1	Out of Order Read
2	1	TOCTOU
	2	Overflow
3	1	Deadlock
	2	Evasion

Table 6.1: Taxonomy of invariants

used as indexes for naming the files that correspond to it. All the Promela model files are located in the `Code/Abstracted` subdirectory and the indexes above are used to name the files for every invariant: `abstract_model_i_j.pml`.

6.3.1 Confidentiality

Out of Order Reading

The model sits in the file `abstract_model_1_1.pml`. In order to catch an out-of-order reading of the data inside the ring buffer, we need 3 global variables:

- an array of the size of the slots that will hold the sequence number of the packets submitted in the ring buffer (called `slot_seq`)
- an integer that will assign the sequence number to the current slots (called `act_seq` for actual sequence number)
- an integer that will be used to check the sequence number when read (called `exp_seq` for expected sequence number)

When a reservation actually occurs, we update the sequence number for the reserved slot to the actual sequence number and we increase the latter by one. Inside the consumer functions, be it the poll function from the libbpf or the raw access to the ring buffer, when we want to consume the data from the ring buffer, we call a check function which checks that the slot read has a sequence number that is the expected one, with an assertion. If the check is passed successfully, then we can update the expected sequence number.
In this way, if the consumers read a packet that has a sequence number inferior to the expected one, then the out of order read error is triggered.

6.3.2 Integrity

TOCTOU

The model sits in the file `abstract_model_2_1.pml`. To catch this security issue, we use a global array called `written` which holds boolean value for each slot of the ring buffer. This array holds the fact that a slot has ever contained data. The TOCTOU assertion is located just before the actual data consumption: if the consumer checked that the slot was not busy anymore and thus ready to be read, and the value is not the same just before the read, then a TOCTOU race has happened. Using the written array is crucial to avoid false alarms if a slot is discarded instead of being submitted.

Overflow

The model sits in the file `abstract_model_2_2.pml`. To try to find this security issue, we need to represent the ring buffer data array byte by byte instead of one array index per slot. Each header would take eight bytes as in the kernel

representation of the ring buffer (the first four for the size and the busy bit, the last four for the page size) and one record would be of size eight bytes, constantly. To not have to modify too much the model of the helper functions already implemented, I introduced two functions that ease the writing and the reading from this new memory layout representation. Since the last four bits of the header are untouched and we do not actually represent the data inside the array, they only take care of the first four bytes of what they read or write:

- `read_len(ptr, out_val)`: reads the four bytes from the pointer's slot and add them by offsetting them using bit-wise operations.

- `write_len(ptr, in_val)`: in this case, we need to extract the four bytes from the value to insert before actually writing it. For this matter, we do a byte-by-byte extraction using the `NO_OWNER` defined as `255` or `0xFF`, which by using the formula `(value » 8*k) & 255` pull the `k`th byte of the value. With this done, we can write in the four bytes after the wanted pointer the real values.

When using a poll function, if the consumer pointer is not aligned with a data slot, then it could lead to some issued, not representable here in the model, however we need to flag this as an issue. This verification is straightforward and trivial since only the consumer program actually modifies the consumer pointer value, the issue will be raised.

6.3.3 Availability

Deadlock

The model sits in the file `abstract_model_3_1.pml`. If this security issue actually arises, then it would mean that the kernel space cannot reserve data slot inside the ring buffer anymore, while there might be plenty of space remaining. Thus the user-space programs cannot retrieve data and the whole program is stuck. To detect this, we cannot simply wait for the model checker to say that we have run into a loop because the abstraction would not permit to detect this behavior. Instead we need to check for missed reservations while free space was detected in the ring buffer. We need two new variables:

- one that keeps the number of actual free slots inside the ring buffer, called `truly_free_slots`

- one that counts the number of consecutive reserve failures called `consecutive_failures`

Every time a reserve operation fails, we add one to the count of failure and we update the number of free slots in the ring buffer by computing the difference between the producer position and the *verified* consumer position, represented by the variable `verified_consumer_pos`. This is maintained by the model to represent the number of records consumed by benign use, independently from the actual value of the consumer position. When the number of consecutive failures passes the number of slots inside the ring buffer and there is actually free slots in the ring buffer then we raise the error:

```
assert(consecutive_failures < N_SLOTS || truly_free_slots == 0);
```

This assertion fails when all three conditions are false:

- `consecutive_failures` $\geq$ `N_SLOTS`: producer has failed to reserve too many times
- `truly_free_slots` $>$ `0`: slots are available
- `consumer_pos != verified_consumer_pos`: the two values have diverged

The third term in the assertion is necessary to avoid false positives.

Evasion

The model sits in the file `abstract_model_3_2.pml`. When choosing a jump size for the next consumer position, we need to verify that no waiting data is skipped. For this, we keep track of the data freshly inserted inside the ring buffer with an array that contains true if data is committed to this slot. When consumed this boolean value is set to false.
The check is done when choosing the jump size. For every slot between the consumer position and the one updated with the jump size, we verify that no data is skipped with a simple for loop.

Chapter 7

Discussion of the Results

7.1 Execution Script

The script `exec.sh` used to execute the Promela models with SPIN in this section can also be found in the Appendix A, under the **Figure A.2**. The execution path is the following:

1. declare the name of the vulnerabilities to check for and their corresponding indexes
2. since I could use arguments to check individually the security invariants, I put some checks over the indexes
3. extract the right Promela model from the arguments
4. generate, compile and run the right model (without the `-N` flag since only one LTL is used (the resource leak from the verifier)

Same as before, to have a quick overview of the errors found in the models, we can `grep` the output with the string `"errors"`, however this can hide any stopped step in the execution flow.

7.2 Results

Since we have one model per invariant, we need to compile and execute the model checker for each one of them. All the compilations are done with the `-DBITSTATE` flag to avoid big state spaces that cannot be handled by my machine. Using the `exec.sh` bash script will execute all the models and output the results. One model can be compiled and executed by giving as arguments to the script its indexes in the `vulnerabilities` array defined to hold the descriptions of the models.

Using a grep pipe to the execution of the `pan` executable compiled by the GCC compiler and generated by the SPIN model checker, the results are shown in the Figure 7.1.

```
1  ===============================================
2  Rule 1 1: Out of Order
3  ===============================================
4  ltl verifier_no_resource_leak: [] ((! (hdr_busy[0])) ||
       (<> (! (hdr_busy[0]))))
5  State-vector 80 byte, depth reached 521, errors: 1
6
7  ===============================================
8  Rule 1 2: Pointers Misalign
9  ===============================================
10 ltl verifier_no_resource_leak: [] ((! (hdr_busy[0])) ||
       (<> (! (hdr_busy[0]))))
11 State-vector 64 byte, depth reached 343, errors: 1
12
13 ===============================================
14 Rule 2 1: TOCTOU
15 ===============================================
16 ltl verifier_no_resource_leak: [] ((! (hdr_busy[0])) ||
       (<> (! (hdr_busy[0]))))
17 State-vector 72 byte, depth reached 633, errors: 1
18
19 ===============================================
20 Rule 2 2: Overflow
21 ===============================================
22 warning: no accept labels are defined, so option -a has
       no effect (ignored)
23 State-vector 120 byte, depth reached 46, errors: 1
24
25 ===============================================
26 Rule 3 1: Deadlock
27 ===============================================
28 ltl verifier_no_resource_leak: [] ((! (hdr_busy[0])) ||
       (<> (! (hdr_busy[0]))))
29 State-vector 72 byte, depth reached 521, errors: 1
30
31 ===============================================
32 Rule 3 2: Evasion
33 ===============================================
34 ltl verifier_no_resource_leak: [] ((! (hdr_busy[0])) ||
       (<> (! (hdr_busy[0]))))
35 State-vector 76 byte, depth reached 527, errors: 1
36
```

Figure 7.1: Results of the SPIN execution over the security Promela models

What we can see from the output is that for every security invariant we modeled inside the Promela models, SPIN has found an execution path that violates them. When the pan executable generated by SPIN finds a state where an invariant is violated, in addition to the output message that contains the "`errors: 1`" and "`Search not completed`" (truncated in the pasted output for clarity), it generates a trail file called `<model_name>.pml.trail`. This file contains a record of the exact sequence of transition interleaving that leads to the violating state. It contains only indexes, which is not human-readable. To actually understand what is the trail of the invariant violation, we can execute the command

```
spin -t <model_name>.pml
```

This will print a step-by-step trace of every statement executed, which process executed it, and how relevant variables' values have been changed. We can add several flags for easier reading:

- `-p`: print each statement as executed
- `-g`: print value of global variables

Which leaves us with this command to fully understand how the violation has been triggered:

```
spin -t -p -g <model_name>.pml
```

Every trail of the models have been printed inside a file (inside the `Code/Abstracted` directory) I called `trail_i_j` where `i` and `j` are the indexes of the models as described before. When reading, one the lines are written in this way

```
1 <id>: proc <pid> (<proc_name>:<proc_n>) <model_name>:<line> (state <
    state_number>) [<code>]
```

When an assertion or a LTL is violated, it explains which one has triggered the error then gives some statistics about

- step numbers
- spawned processes
- all global variables' values

This human-readable file can help us understand what steps the model took before triggering the error and thus, help us re-create the issue in a real eBPF program.

7.2.1 Out of Order Read

Setup

From what we can understand with the file `trail_1_1`, one producer process is used alongside with two consumers. Each consumer uses the libbpf poll function to extract data from the ring buffer. The consumers are called *C2* and *C3* since they are respectively the processes number two and three.

Trace Summary

1. *C2* gets the consumer pointer from the ring buffer and gets preempted
2. *C3* consumes a record from the ring buffer and updates the sequence tracking variables and the consumer pointer (`exp_seq = 5` and `consumer_pos = 5`)
3. *C2* actually perform its reading over the data and overwrites the variables updated by *C3* with `exp_seq = 4` and `consumer_pos = 4`
4. *C3* reads a new slot of the ring buffer against the corrupted values. It triggers the assert because it sees that the sequence number of this new data is 5 and the `exp_seq` is 4.

Interpretation

This highlights that using two consumers, using a benign access to the ring buffer using the libbpf poll function can break the reading of the data. Since the two consumers advance the consumer pointer independently, a lagging process can retroactively invalidate a faster consumer progress. The consumption stream becomes non-monotonous and the data appears out of order relative to the ring buffer own tracking.

7.2.2 TOCTOU

Setup

From what we can understand with the file `trail_2_1`, also in this case this vulnerability can be triggered by the use of two consumers that use the poll function from the libbpf. Also here the consumers are written *C2* and *C3*.

Trace Summary

1. *C3* wants to read a data slot and checks that the slot is not busy so ready to consume

2. *C3* gets preempted for a long time until the producer fills the ring buffer, the *C2* reads every packet and come back to the slot checked by *C3*

3. the producer reserves the slot and sets the busy bit to true

4. *C3* tries to read it but the slot is not ready in theory: there is a difference between the value at the time of check and the value at the time of use

Interpretation

This vulnerability has been triggered because there is no immediate consumption: the check and the use are separated by an unbounded scheduling gap. While the consumer thinks the slot is safe to be consumed and read, it is actually in the middle of a reservation process for an unrelated record. The race window is extremely narrow in a real-world condition because this would mean the consumer is preempted for a very long time.

7.2.3 Overflow

Setup

From what we can understand with the file `trail_2_2`, this case is particular: only one consumer is needed to trigger the vulnerability. As mentioned before, this one is trivial since we modify the value of the consumer pointer to be misaligned with the bytes of the ring buffer. However in this case, the use of a raw consumer accessing the ring buffer with mmap is needed, instead of a consumer using the libbpf to get the memory pages of the ring buffer.

Trace Summary

The trace produced by the model checker is straightforward:

1. the consumer simulates the consumption of a slot and add only one byte to the consumer pointer

2. during the next round of the consumer polling the data from the ring buffer, it detects that the consumer pointer is not aligned with the eight bytes of the structure and triggers the assertion

Interpretation

This lets us think that using mmap to connect to the ring buffer and misusing the consumer pointer permits us to jump inside the data records or the headers of the ring buffer. Since representing the exact behavior with all the data present inside

the header makes us hit the limit of the state-space allowed by the machine, it is not possible to model what exactly can break inside the ring buffer in this case. The different issues will be described later during the real eBPF programs.

7.2.4 Deadlock

Setup

From what we can understand with the file `trail_3_1`, only one producer and one raw consumer are needed for the issue to come up. The raw consumer can choose between several distances to jump the consumer position, even when that jumps exceeds the number of records available.

Trace Summary

1. The producer pushes data inside the ring buffer, putting the consumer pointer to 1

2. The raw consumer chooses a jump size of 3, overshooting the producer position

3. The next reservation function computes the distances: `(new_prod_pos - cons_pos) % 8 = 7` $\geq$ `4`. The check is false and the kernel cannot reserve a slot

4. The producer program repeats the reserve attempts several times and finish by triggering the assertion because of the cap set in the Promela model

Interpretation

Because the raw consumer connecting directly to the ring buffer memory regions, can update the consumer pointer, allowing it to overpass the producer pointer, can change the view the kernel program has about the ring buffer's occupancy: it thinks it is full while the structure actually contains free slots.

7.2.5 Evasion

Setup

From what we can understand with the file `trail_3_2`, we need at least one raw consumer to catch this one, because we need to have access directly to the consumer pointer.

Trace Summary

In the trace in the folder:

1. `raw_consumer` chooses a jump of 2 and consumes its slot.

2. it gets preempted by a whole reserve and submit cycle on the next slot of the ring buffer

3. because of the jump the slot that just got submitted will never be read because the consumer pointer now points beyond it

Interpretation

When using mmap to connect to the ring buffer data space, if the consumer pointer is not updated properly, this could lead to data skipping: some records will be completely ignored by the consumers. However this can be true only if the consumer pointer does not go beyond the producer pointer, which would simply block the ring buffer from being filled and emptied as seen in the last vulnerability.

7.3 Replication of the Results

Now that we have the exact trace that led to the trigger of the security invariants inserted in the model, the goal is to replicate them in a real eBPF program to see if we have false or true positive. Thanks to the trace, we know in what way the program should behave in order to trigger the issue. All the programs will consist in a kernel side that will reserve and submit data inside the ring buffer since no trace needs a discard function to be replicated, and a user-space program that will contain two or more consumers. For now, the replication of the issues will only be in a controlled environment, with no really sensitive data. The data that will be pushed inside the ring buffer will be sequence number to know in which order the packets have been submitted inside the ring buffer and in what order they are read by the user space.

In all the folders for the vulnerabilities there are several files:

- `i_j.txt`: contains the trace of the model checking, a copy of the `trail_i_j` file sitting in the parent directory
- `makefile`: for the compilation of the user-space and kernel eBPF programs
- `notes.md`: for the notes on the vulnerabilities and their explanations

- `vuln.*`: all C files and dependencies needed for the compilation of the executable
- `vuln`: binary that actually executes the vulnerability

Some folders are different but this will be explained when needed.

7.3.1 Out of Order Read

The Promela model only contains one producer and two libbpf consumers. For the vulnerability to happen, we need that both the polling functions happen at the same time to have a chance of race and de-synchronization.
First I have created a simple eBPF code that just prints the events as they are read by the consumers. Then I moved to a very basic EDR that keeps track of the processes created and terminated within the machine in the folder `1_1/EDR`.

First Program

- Kernel Code: the kernel part of the simple program is just putting a sequence number inside the ring buffer whenever there is a context switch. The program is attached to the tracepoint `sched/sched_switch`. Its role is to reserve a slot, write the sequence number, update it and then submit the data inside the ring buffer.
- User-Space Code: the user-space of the program is firing two consumer programs using the consume function from the libbpf on two separated threads. When one of them reads a slot from the ring buffer, it prints the sequence number. I put a very small time to sleep for the consumers between each consume (10 micro seconds) so it can rapidly fire and reproduce the vulnerability.

When executing this program, as shown inside the `output.txt` file, some events are read in the right order but in the lines 12-15:

```
[Thread 2] Seq 9
[Thread 2] Seq 10
[Thread 1] Seq 9
[Thread 1] Seq 10
```

We can observe that the events are re-read by the consumers and not in order. This is exactly what we were expecting. There is a more relevant example in the file lines 246-255:

```
[Thread 2] Seq 239
[Thread 2] Seq 240
```

```
[Thread 1] Seq 240
[Thread 1] Seq 241
[Thread 1] Seq 242
[Thread 1] Seq 243
[Thread 2] Seq 241
[Thread 2] Seq 242
[Thread 2] Seq 243
[Thread 2] Seq 244
```

In this output, while each consumer read the packets in order, if the same application use these two consumers to read data from the same ring buffer, packet reads are duplicated: each consumer thread reads the packets fine but the application will read them twice. With this, we can show that a simple EDR program that keeps track of the currently running processes can crash because of this issue.

Second Program

- Kernel Code: the ring buffer will contain records of type `event_t`, which hold a PID, a type, and a sequence number. The type will be used by the EDR to decide which action to perform. The ring buffer is small to avoid waiting a long time before seeing the vulnerability happen. The kernel space code contains two programs, each one hooked to a different tracepoint:
  - `handle_fork` attached to the tracepoint `sched/sched_process_exec` (triggered when a new process is executed). When the tracepoint is hit, it reserves a slot, inserts in the event the PID of the program, a type of `1`, and the sequence number of the packet, which is incremented atomically thanks to the `__sync_fetch_and_add` function. After this, it submits the event.
  - `handle_exit` attached to the tracepoint `sched/sched_process_exit` (triggered when a process is terminated). It performs the same operations as the other program, only putting a type 2 in the event. There is one difference, I filter the PID of the triggering program so this does not enter in an uncontrollable loop.
- User-Space Code: the user-space code is re-creating a very simple EDR that keeps track of the process launched and exited. The EDR is an array containing a `Metadata * edr_tracking`. Each element of the array contains the PID of the program and a status string. In this program, two consumers are used, each one on its thread, waiting for each other to be waken up before actually executing. The function that handles the events reading is the following:

- When receiving a type 1 (process created), the program allocates space for the `Metadata` object with `malloc` and prints in the status property "RUNNING".
- When receiving a type 2 (process exited), the program frees the space taken by the running process using the `free` function and setting to `NULL` the element of the array.

The output of the command `./vuln` is in the folder in the file called `edr_output.txt`. When launched, at first the programs behaves correctly:

```
[Thread 2] Sequence: 0 | PID 43319 STARTED
[Thread 1] Freeing PID 43319
[Thread 1] Sequence: 1 | PID 43319 EXITED
[Thread 2] Freeing PID 43319
```

However we can observe that some events are read by both the threads and thus are printed in the terminal. And after some time, the program crashes, stating either

```
free(): unaligned chunk detected in tcache 2
```

Or

```
free(): double free detected in tcache 2
```

This happens because both threads read the same ring buffer packet (the "PID exited" for example), so the same `idx`. What happens is that `[Thread 1]` saves the `edr_tracking` array's corresponding slot into its `metadata` variable. But before it can perform any action on it, `[Thread 2]` also reads the same pointer. Now both threads have their own local copy of the same pointer. After this, both threads call the free function and the crash happens. The reason why two different messages can be shown by the execution of this program is because the `struct Metadata` is relatively small (20 bytes) so it is managed by the *tcache* pool of the heap. If both free calls happen right after one another, the program crashes and throws the error `double free detected`. Otherwise, a new type 1 event is read and this newly freed block is handed out to the heap manager that sees its bookkeeping information overwritten. After this, when the second free happens, since the reservation metadata are completely wrong, the chunk becomes invalid to be freed, forcing the program to show the error `unaligned chunk detected`.
This happens even when putting some NULL-protections around the free of the element, because of the non-atomicity of the operations. If both threads catch the array's slot at the same time, they pass the NULL-check wrapping the free call and still produce the same crash as before.

To conclude, using two libbpf consumers over the same ring buffer, this can cause some tracing tools to crash if the double read of some packets can make programs crash. To avoid this, increasing the time of sleep of the consumers before triggering another consume from the ring buffer can help reading all the packets only once and in the right order. Another fix could be to keep track of the sequence number of the packets that are read and skip the values that are read twice, but this would require a change in the kernel eBPF code to include a sequence number if not already supported.

7.3.2 TOCTOU

This case attempts to expose a TOCTOU vulnerability between the kernel's ring buffer producer and the raw user-space consumer. As seen in the SPIN results earlier in the document, this scenario targets the window between the moment the consumer verifies that a record's busy bit is cleared and the moment it actually reads the payload.

Kernel Code

The kernel code is attached to the `sched/sched_switch` tracepoint and fills the ring buffer with monotonically increasing sequence numbers.

User-space Code

Rather than using directly the consume function from the libbpf, we try to replicate its behavior but changing one important thing: catching the TOCTOU vulnerability if it happens. The threads read consumer and producer positions atomically, check the busy flag of the record and if it is clear, then advance the consumer position by a record size, to reproduce the read of the payload. After this, the header is re-read and if the busy bit is not clear this time, the TOCTOU warning is triggered.

Results

No TOCTOU evidence was found during the execution. In order for this to happen the kernel needs to fill up the 256 slots of the ring buffer, wrapping around and finally using the same slot, between the first busy check and the actual use of the payload by the user space.

7.3.3 Overflow

This case exposes a severe logical flaw that occur when a custom user-space reader perform a manual and unconstrained update to the ring buffer's consumer pointer. Here the counterexample given by the SPIN model checker illustrates a malformed state where the consumer pointer lands inside a record rather than being aligned with the packet's header. The resulting runtime consequences can be multiple: a libbpf consumer use this corrupted consumer pointer and misinterpret raw payload bytes as the control information and block the ring buffer or skip some records.

Kernel Code

The kernel code is attaching the program to the tracepoint `sched/sched_process_exec` and fills the ring buffer with sequence numbers. The program stills uses and updates the sequence number atomically and submits them.

User-space Code

For this section of the program, we need two accesses to the ring buffer: one legit from the libbpf and one raw from the mmap syscall. The access with the libbpf is done as usual with the `ring_buffer__new` function and the FD of the eBPF map. For the raw access, we use the mmap to access solely the consumer position because the rest of the ring buffer is useless for the raw access. The mmap function asks for 6 arguments:

- `void * addr`: the starting address of the mapping. `NULL` in our case.
- `size_t length`: length of the mapping. `page_size` in our case because of the memory layout of the ring buffer.
- `int prot`: the protections on the memory, should not conflict with the protections of the file. In our case, from the user space, the consumer pointer can be read and written so the protections are `PROT_READ` and `PROT_WRITE`.
- `int flags`: if updates are visible to other processes mapping the same region. We use the same value as in the libbpf when it connects to the memory region of the ring buffer which is `MAP_SHARED`.
- `int fd`: this is the object in which the mapping occurs. In our case, we need the FD of the ring buffer created by the eBPF program.
- `off_t offset`: this is the offset of the memory region where the mapping is done. In this case, since the consumer pointer is the first value in the layout of the ring buffer, the offset will be `0`.

The user space programs uses threads to fire the two consumers at the same time. The one using the libbpf consumes the data and prints the sequence numbers. The one using mmap only updates the consumer position with 1 bytes instead of a full record for example.

Results

Here is what happens when executing the program:

```
Generating events...
MMAP Sleeping...
Seq: 0
Seq: 1
Seq: 2
Seq: 3
Seq: 4
Seq: 5
Seq: 6
Seq: 7
...
Seq: 256
Seq: 257
Seq: 258
Seq: 259
Seq: 260
MMAP Triggered !
Seq: 576460752303423489
```

And then it stops. When reading the file `/sys/kernel/debug/tracing/trace_pipe`, which displays all the strings printed with the helper function `bpf_printk`, we can see:

```
...
docker-45832   [001] ...21  4827.147053:
    bpf_trace_printk: [+] Submitted Seq: 260

docker-45843   [008] ...21  4829.177519:
    bpf_trace_printk: [+] Submitted Seq: 261

zsh-45853   [005] ...21  4829.943628:
    bpf_trace_printk: [-] Reserve Failed
```

```
    dircolors-45855   [008] ...21  4829.985334:
        bpf_trace_printk: [-] Reserve Failed

    neofetch-45856   [015] ...21  4829.987118:
        bpf_trace_printk: [-] Reserve Failed
    ...
```

Which means that the reservation of the eBPF program always fail after the mmap program being triggered. Thus the ring buffer is completely blocked after the consumer pointer value's change: no data is submitted inside and therefore no data is read from it.

Normally, the consumer pointer tells the libbpf at which bytes is the next record to read. The pointers points to the eight bytes header of the record which contains four bytes for the size of the record (including the header size) and four bytes for the page offset. The libbpf reads the size value from the header and begins to read the data from the consumer pointer to which is added the size of a header. Since little endian is used and the actual payloads of the ring buffer records are eight bytes, the memory layout is the following:

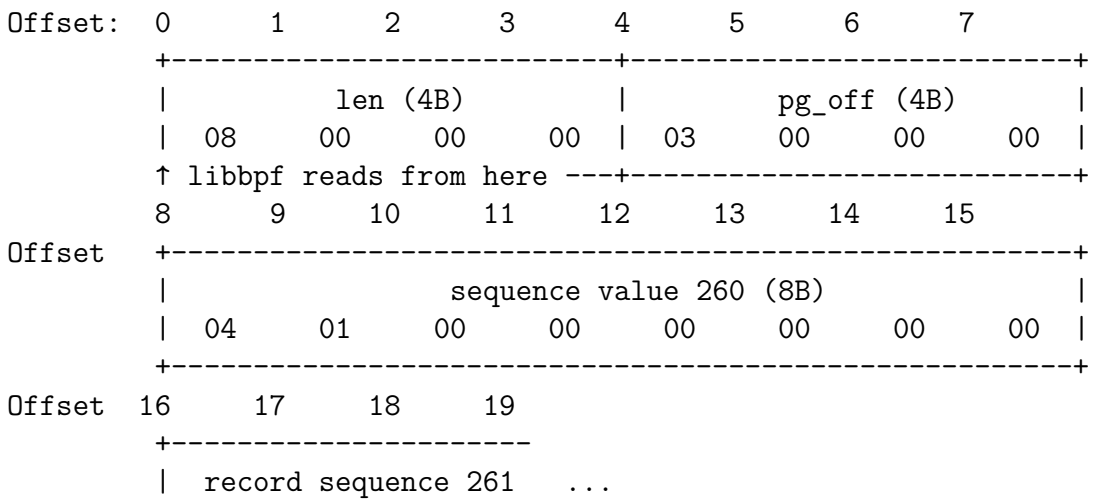


Figure 7.2: Ring buffer records memory layout

The `pg_off` value has been found with the same program by reading the value of the header of a record.
In our special case, we have pushed the consumer position by one byte, which means the header it is reading is the header of the next record advanced by one:

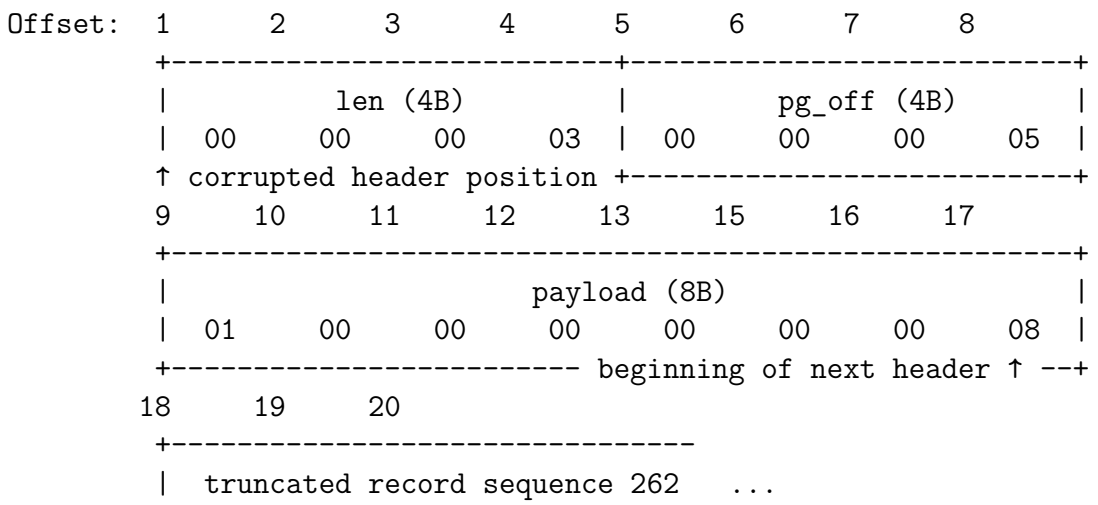


Figure 7.3: Actual data read by the libbpf consumer

This shows that the libbpf consumer thinks the next record is very large (`0x3000000` bytes) and that the payload is containing the beginning of the header of the next record, and missing its first byte. By the figure, the next sequence value to be read by the libbpf consumer is `0x800000000000001`, which corresponds in base 10 to the number 576460752303423489. The fact that we only see the payload of the next packet and not more than 50 megabytes of the ring buffer data is because the libbpf consumer is only reading `u64` (8 bytes) values. It is important to note that this value appears only when the sequence number is already high: the ring buffer has wrapped around and still contains data from the previous round of the ring buffer. Indeed, when repeating the same experiment but with a lower sequence number (like 10), the next sequence number read after the mmap trigger is 0, simply because the next data slots are not initialized and do not contain any data in them. Since the `max_entries` field for the ring buffer defined in the kernel code is set to 4096 (Bytes) and because the sequence numbers use the type `__u64` (`long long unsigned int`, represented by 8 Bytes), the size of a full record is 16 bytes (8 for the header and 8 for the sequence number) the ring buffer wraps around when 256 entries have been added. This is why it shows the high values when the sequence number is 261 and not when it is only 10.

Since we can update the consumer pointer's value as we want, we can try every possible value inside the header or directly to another record. The sections about the Evasion and the Deadlock do not have a specific paragraph because they can be counted as variants of this example. The following tables (**Table 7.1** and **Table 7.2**) contain the results of the program execution for every byte offset.

When a result is not trivial, explanations will be given.

Offset (Bytes)	**Result**	**Output**

2	ringbuf blocked	`...`
		`Seq: 259`
		`Seq: 260`
		`Seq: 261`
		`MMAP Triggered !`
		`Seq: 2251799813685248`

3	ringbuf blocked	`...`
		`Seq: 260`
		`Seq: 261`
		`Seq: 262`
		`MMAP Triggered !`
		`Seq: 8796093022208`

4	ringbuf continues *	`...`
		`Seq: 261`
		`Seq: 262`
		`MMAP Triggered !`
		`Seq: 34359738368`
		`Seq: 34359738368`
		`Seq: 34359738368`
		`...`

Table 7.1: Byte-by-byte offset results (1/3)

When offsetting the consumer pointer by 1, 2 and 3, we observe the same behavior: the ring buffer cannot give space for new reservations and the consumer reads one last value before stopping. This last value can be obtained with the same reasoning than previously with the content of the headers and the sequence payloads. However when advancing the consumer by 4 bytes, which corresponds to half a header, the behavior is slightly different because the consumer program does not stop from printing values from the ring buffer. The value printed from the libbpf consumer function depends if the buffer has wrapped around or not: if not, the value printed is 0, otherwise it is 34359738368 which corresponds to `0x800000000`.

Offset (Bytes)	Result	Output

5	ringbuf blocked	`...`
		`Seq: 259`
		`Seq: 260`
		`Seq: 261`
		`MMAP Triggered !`
		`Seq: 216172782248001536`

6	ringbuf blocked	`...`
		`Seq: 260`
		`Seq: 261`
		`Seq: 262`
		`MMAP Triggered !`
		`Seq: 844424930656256`

7	ringbuf blocked	`...`
		`Seq: 259`
		`Seq: 260`
		`Seq: 261`
		`MMAP Triggered !`
		`Seq: 3298534885376`

Table 7.2: Byte-by-byte offset results (2/3)

When trying with other values greater than 4, the result is pretty much the same: offsetting the consumer position by 5, 6 or 7 will result in the same result as 1, 2 and 3. The ring buffer blocks and prints one last value corresponding to the bytes already present is the ring buffer.
The reason why the ring buffer blocks with values different than 4 is because the new length value that the libbpf reads for the record consumption becomes very high, and since the libbpf updates the consumer pointer's value with the number it has read from the header, the value of the consumer position becomes very high and becomes far ahead of the producer pointer's value. The checks done by the eBPF producer (`new_prod_pos - cons_pos` $\leq$ `rb.size`) wraps around to a huge number and the program thinks the ring buffer is full.
For the rest of the values of the offset, I will only show the interesting results in the **Figure 7.3**: all the offset not shown produce the same result as an already covered value (but most of them will only print a last value and completely block).

Offset (Bytes)	Result	Output
9	ringbuf continues "randomly"	... Seq: 261 Seq: 262 MMAP Triggered ! Seq: 576460752353755136 Seq: 648518346391683072 Seq: 720575940429611008 ...
16	skips one and continues	... Seq: 258 Seq: 259 Seq: 260 MMAP Triggered ! Seq: 262 Seq: 263 ...

Table 7.3: Byte-by-byte offset results (3/3)

As we can see in the table, using the offset of 9 will trigger a new behavior: the ring buffer does not stop but the consumer prints values that seems not having any link. The reason behind this is that the length value read by the consumer is the value from the sequence payload. Since the payload changes for every record, the value printed will be different.
One value that is interesting is when we offset the consumer position by 16. This special value corresponds exactly to a full record (the header and the payload). As we can see from the output of the programs, it skips the record of sequence 261 and continues as nothing happened. This special case highlight the proof that the **2.2 Evasion** is possible. To extend this result, we can try with the values that are multiple of the record's length such as 32 or 48. However in our case the program crashed as with the values 1, 5 or 11. This happens because the check done by the kernel to check for free space is comparing the producer position and the new producer position, which is the old producer position advanced by the length of the reservation. Since with the offset of 16 the values are the same, the check is passed by the kernel. When using bigger values such as 32 or 48, the check is failed and the kernel thinks the ring buffer is full.

The last security invariant we had to cover was the deadlock of the ring buffer, triggered because the consumer position was advanced too much and would break

the verification done by the eBPF program loaded inside the kernel. This behavior is exactly what happens for the big values of the offset that just blocks the ring buffer, while it is not really full.

It is important to note that it is also possible to subtract value from the consumer position. In the **Table 7.4**, I will only show 2 examples, one where we subtract a number that falls inside a payload, and the other where we subtract the value of one full record.

Offset (Bytes)	**Result**	**Output**

-2	ringbuf blocked	`...`
		`Seq: 259`
		`Seq: 260`
		`Seq: 261`
		`MMAP Triggered !`
		`Seq: 393216`

-16	replays one and continues	`...`
		`Seq: 259`
		`Seq: 260`
		`Seq: 261`
		`MMAP Triggered !`
		`Seq: 261`
		`Seq: 262`
		`Seq: 263`
		`...`

Table 7.4: Byte-by-byte minus offset results

As we could expect, the offset of -2 produces the same results as the other values that led directly inside a record. For the case of the full record subtraction, the ring buffer is not blocked but the libbpf consumer will read again the same packet as the previous one. Using this trick could permit some malicious program to repeat the same messages while the new ones are completely different: replay a packet that shows normal activity to an EDR while the system undergoes an attack.

7.4 Analysis and Comparison of the results

7.4.1 Theory vs Observations

The SPIN model checker, through the abstracted Promela models, found some traces for security invariants violation. This is because of the lack of protection around the consumer pointer of the ring buffer, that is not atomically checked and modified, leading to race conditions between multiple consumers.
Regarding the false positives and true positives of the findings, I was not able to replicate the violations only for the TOCTOU case: the rest of the claims were found true by the means of a real eBPF program. The TOCTOU vulnerability was not replicated, mainly because the trail that the SPIN model checker generated contained the preemption of a consumer program for a long enough period of time, which was not highly probable to happen in a real eBPF scenario.

7.4.2 State Space Complexity

To compare the different models and their execution, in term of state space, here are some outputs of the script outcomes. However we should keep in mind that the invariants checked were not the same for both the cases.
Here is the output for the very first naive model:

```
1 ================================================
2  VERIFYING: rule_ordering
3 ================================================
4 pan: ltl formula rule_no_double_submit
5 error: max search depth too small
```

The state space generated by the model was way too big and could not be resolved in a reasonable time. This is why I tried to shrink it by changing the variables types to smaller ones. Here is the output for the same rule:

```
1 ================================================
2  VERIFYING: ORDERING
3 ================================================
4 ltl rule_ordering: [] ((! (! (transient_gap))) || (((rb.
    consumer_pos<=rb.pending_pos)) && ((rb.pending_pos<=
    rb.producer_pos))))
5 Depth=     404 States=    1e+06 Transitions= 3.53e+06
    Memory=  4103.160 t=       1.5 R=   7e+05
```

```
Depth=     404 States=    2e+06 Transitions= 7.26e+06
   Memory=  4103.160 t=      3.1 R=   6e+05
Depth=     404 States=    3e+06 Transitions=  1.1e+07
   Memory=  4103.160 t=     4.71 R=   6e+05

(Spin Version 6.5.2 -- 18 September 2025)

Bit statespace search for:
    never claim             + (rule_ordering)
    assertion violations    + (if within scope of claim)
    acceptance   cycles     + (fairness disabled)
    invalid end states  - (disabled by never claim)

State-vector 388 byte, depth reached 404, errors: 0
  3319207 states, stored
  8891760 states, matched
 12210967 transitions (= stored+matched)
   333066 atomic steps

hash factor: 10351.8 (best if > 100.)

bits set per state: 3 (-k3)

Stats on memory usage (in Megabytes):
 1291.501   equivalent memory usage for states (stored*(
   State-vector + overhead))
 4096.000   memory used for hash array (-w35)
    0.763   memory used for bit stack
    6.104   memory used for DFS stack (-m100000)
 4103.160   total actual memory usage


 ...

pan: elapsed time 5.25 seconds
pan: rate  632229.9 states/second
```

Finally, here is the memory usage output for the more abstracted model. For this run, I used the flags `-c0` for the execution of the `pan` binary to avoid the model to stop for the first error and let it run entirely to compete with the other models. This is why it shows a very high number of errors, because it counted every time it

encountered the assertion error of the model.

```
1  ===============================================
2  Rule 1 1: Out of Order
3  ===============================================
4  ltl verifier_no_resource_leak: [] ((! (hdr_busy[0])) ||
       (<> (! (hdr_busy[0]))))
5  pan:1: assertion violated ((slot_seq[slot]!=255)&&(
       slot_seq[slot]==exp_seq)) (at depth 167)
6  Depth=     609 States=    1e+06 Transitions= 1.65e+06
       Memory=  4102.299 t=     0.57 R=   2e+06
7  Depth=     609 States=    2e+06 Transitions=  3.4e+06
       Memory=  4102.299 t=     1.17 R=   2e+06
8  Depth=     609 States=    3e+06 Transitions= 5.09e+06
       Memory=  4102.299 t=     1.77 R=   2e+06
9  Depth=     609 States=    4e+06 Transitions= 6.78e+06
       Memory=  4102.299 t=     2.37 R=   2e+06
10
11 ...
12
13 Depth=     609 States=  6.1e+07 Transitions= 1.06e+08
       Memory=  4102.299 t=     31.7 R=   2e+06
14 Depth=     609 States=  6.2e+07 Transitions= 1.07e+08
       Memory=  4102.299 t=     32.2 R=   2e+06
15
16 (Spin Version 6.5.2 -- 18 September 2025)
17     + Partial Order Reduction
18
19 Bit statespace search for:
20     never claim             + (verifier_no_resource_leak
       )
21     assertion violations    + (if within scope of claim)
22     acceptance   cycles     + (fairness enabled)
23     invalid end states  - (disabled by never claim)
24
25 State-vector 80 byte, depth reached 609, errors: 4783774
26  51637993 states, stored (6.24106e+07 visited)
27  45709812 states, matched
28 1.0812039e+08 transitions (= visited+matched)
29         0 atomic steps
```

```
30
31 hash factor: 550.543 (best if > 100.)
32
33 bits set per state: 3 (-k3)
34
35 Stats on memory usage (in Megabytes):
36  5318.549   equivalent memory usage for states (stored*(
      State-vector + overhead))
37  4096.000   memory used for hash array (-w35)
38     0.763   memory used for bit stack
39     5.341   memory used for DFS stack (-m100000)
40  4102.299   total actual memory usage
41
42 ...
43
44 pan: elapsed time 32.4 seconds
45 pan: rate 1928037.8 states/second
```

To recap the numbers in a table:

	Simplified Model	**Abstract Model**
State-vector	388B	80B
Depth	404	609
States stored	3,319,207	51,637,993
Time elapsed	5.25	32.4
Rate	632,229.90 states/s	1,928,037.8 states/s

Table 7.5: Models Comparison

The most significant figure is the state-vector size. It was about 388B in the first naive model and only 80B in the abstracted version, which is around a 4.85x reduction. Since the states are used for every node of the verification, a smaller state-vector means that every operation made by the model checker is cheaper. The goal of the abstracted model has been fulfilled: it successfully reduced the state representation to the essential variables needed to demonstrate if a property holds or not. This improvement can be easily seen using the states per second metrics: the abstract model processes 1,928,037 states per seconds while the naive simplified model only handled 632,229. The depth and the number of states stored show that the abstract model caught execution paths that required more interleaved to reach. The execution time is a misleading metric because the naive model finishes sooner because it has done less than the abstract model and its numerous states.

Metrics about memory usage have not been included inside the comparison because these values has been controlled during the execution by the flags

- `-DMEMLIM` during the compilation of the C code to limit the use of the RAM during the execution of the program
- `-m` during the execution of the verification to maximize the stack depth limit
- `-w` (coupled with the `-DBITSTATE` flag of the compilation) during the execution of the verification to set the number of entries inside the hash table for the hashed states.

So in the end the programs used almost the same memory during the execution: 4,103.160B for the naive model and 4,157.231B for the abstract model. (redo the execution with same -w)

Chapter 8

Real-World Case Studies

8.1 Real-World Case Studies

To interact with eBPF programs, the user (or the program) triggering the loading and other eBPF-relevant operations must have the `CAP_BPF` or `CAP_SYS_ADMIN` permissions. The second capability permits to perform a lot of powerful operations such as bpf and ioctl operations or use ptrace. However this capability contained too much diverse permissions, and this is why they created the `CAP_BPF` ability, to make a clear distinction between the BPF operations and the rest. This is why every program that replicate vulnerabilities are executed with `sudo`, which is a proxy for these abilities. Of course, `CAP_BPF` can be granted independently of full root.
Now that we have proven that some of the vulnerabilities found thanks to the SPIN model checker were actually replicable in a real eBPF program, we can try to mount an "attack" with an external program on an already running eBPF instance: link the malicious program to the eBPF ring buffer maps used by the benign program and try to reproduce the cases.

8.1.1 Local Program Case Study

Code Presentation

In this case, I tried to replicate an attack on a very simple program running on my machine. The simple eBPF program is the following. It is attached to the tracepoint `sched/sched_process_exec` and every time it is triggered, it updates and pushes a sequence number inside the ring buffer.
The user space program uses the libbpf function `ring_buffer__consume` to read the sequence numbers and prints them in the terminal. This is basically the same user space as the programs before, only one thing changes: depending on the way

to connect to the map, I use the libbpf `bpf_map__pin` or not, to pin the ring buffer map to the filesystem of the machine. There is a good reason to actually pin the map to the filesystem: maps summoned by a program normally live as long as the process holds their file descriptor. This means that if the program crashes or is restarted, the state and the content of the map is completely lost. In order to resume this, the program can pin the map to the filesystem and give it a permanent fd. Also, when sharing a map with different eBPF program, it is essential to pin it, allowing any process within the filesystem to access the map independently without coordination of the loader program.

After this, we need to craft an attacker program located in the file `Code/Abstracted/eBPF/2_2attack/attack.c`, which connects to the map and disrupt its pointers. The connection process is very similar to the previous user space programs that used the mmap syscall to interact with the ring buffers:

- map the consumer pointer
- map the producer pointer
- map the actual data of the ring buffer
- change the value of the pointers and read the data of the ring buffer according to the pointers' value

The program essentially use the same functions as the other user-space programs however one thing is missing: to use the mmap function, we need the fd of the map we are aiming to connect to. For this, two ways can be used in the attacker program:

- If the user-space program pins the map in the filesystem of the machine and the attacker knows where it is: we use the eBPF function `bpf_obj_get` which gets the file descriptor of a BPF object that has been pinned, thanks to is path string. To access the map, the attacker needs to have enough rights to: previously mentioned capabilities are not necessarily needed.
- Otherwise, the ring buffer if not pinned exists inside the kernel's internal map table. To retrieve the file descriptor of the map in this case, we need to loop through all the maps inside the kernel with the function `bpf_map_get_fd_by_id`, which as its name indicates, retrieves the file descriptors of the maps from their ID. However this will return all the maps loaded inside the kernel, also the ones from other programs and of other types such as hash table maps. We use the function `bpf_obj_get_info_by_fd` which gives us information about the retrieved map such as (from the `linux/include/linux/bpf.h` file) the type, the max entries or even its name. Thanks to this, we can filter out the

maps that are not ring buffers. Since I tested this program on my machine that is not running other eBPF programs than the one I wrote, any ring buffer that the attack program will find has been loaded from my eBPF program. This is why I use without further checks the de-synchronization over every of these ring buffers. I then get the consumer pointer and updates it.

For the recall, when using the method where we actually know the fd of the ring buffer in the filesystem of the machine, we need it to be pinned and the right permissions for the directory where it is pinned. As for the second method, the capabilities needed to access the map are `CAP_BPF` or `CAP_SYS_ADMIN`. However the pin cannot be done anywhere on the standard filesystem of the machine. It must be pinned in a portion of the filesystem that reside on a mounted BPF virtual filesystem. For this, the command is

```
mount -t bpf bpf /path/to/directory
```

After this, the map can be pinned and accessed by every other programs with enough capabilities and privileges.

Results

The results are relatively the same as when we were using the overflow eBPF programs:

- When making the consumer pointer jump in the middle of a record, it blocks and read a huge value or a 0, depending on if the ring buffer has already wrapped around or not
- When making it jump of one full record (header and payload), it skips one sequence number and continues as if nothing happened
- When making it jump backwards of a full record, it replays the same sequence number and continues

Only this time, the program that compromises the value of the pointer can be launched independently of the eBPF kernel and user-space programs, making it easier to control the behavior of the read records.

8.1.2 Falco Case Study

Now that we have shown that it was possible to attach an external program to an already running eBPF ring buffer map running inside the kernel and modify the value of the consumer pointer to change its behavior, we can try to replicate this on an existing open-source tool. For this experiment, I have chosen to use the

docker version of the Falco tool, initially developed by *Sysdig*.
From their website (`https://falco.org/about/`), Falco is a "network of security cameras" for the system where it is installed. It basically collects data happening in real time and alerts if something strange happens. It can be used to detect security threats such as "crypto mining, file extraction [and] privilege escalation". These events can be detected thanks to a set of rules that collect kernel syscalls, kubernetes audit logs and cloud events. It is nowadays a highly used software as it has been downloaded more than 100 million times. Two version of the Falco security tool exist: one being an eBPF probe, working the same way as the programs we have been using since the beginning of the experiment, or as a kernel module.

To perform this experiment, I have obviously chosen to use the eBPF version of the Falco tool, using it in a docker container. The version of the Docker at the time of this experiment was `Docker version 29.7.2, build a7dcaa6` and the image for the Falco container used was the `falcosecurity/falco@sha256:d0cfe422d6ac0e0f20857798f46c7d7273210e1b064b22821e4e6e7f843cde6b`.

Before launching the malicious program, several steps need to be executed.

1. Firstly, launch the Falco container with the command:

```
docker run --rm --name falco --privileged \
  -v /sys/fs/bpf:/sys/fs/bpf \
  -v /sys/kernel/tracing:/sys/kernel/tracing:ro \
  -v /var/run/docker.sock:/host/var/run/docker.sock \
  -v /proc:/host/proc:ro \
  -v /etc:/host/etc:ro \
  falcosecurity/falco:latest
```

   The `-privileged` flag is important because Falco interacts directly with the host's kernel to perform tracing and monitoring. This adds the same privileges to the container as if it had `sudo` capabilities. We can use `-cap-add=BPF` to simulate the `CAP_BPF` capabilities the user needs to launch a user-space program that loads eBPF programs inside the kernel. The `-v` flags with two arguments (`host_path:container_path`) is used in order to mount a special directory of the container in the filesystem of the host machine. The four last lines containing the `-v` flag are the ones written on the official Falco documentation. The first one (`-v /sys/fs/bpf:/sys/fs/bp`) has been added by me to pin the eBPF maps from the container to the host's filesystem. This can be useful for the same reasons as stated as when I explained why pinning

a map to the filesystem is recommended: when the container is stopped, the map loses all of its state.

2. From here, as stated before, 2 cases are possible: or the program does not have the `CAP_BPF` privileges and there is a huge misconfiguration of the map pinning of the Falco ring buffers and other maps, or the program has the capabilities.

The file `Code/Abstracted/eBPF/2_2attack/Falco/attack.c` is a slightly modified version of the previously introduced attack file. Firstly, Falco uses separate ring buffer maps for every single CPU core on the host machine. Since my tests have been done on a virtual machine running `Ubuntu 26.04.1 LTS`, I only gave 2 cores to run so it spawned only 2 ring buffers on my test machine. This means that for the `known_fd` function to be correctly executed, we need to mmap both ring buffers. The location of the ring buffers in the `bpf_object_get` function are determined by the misconfiguration I will explain later. Other than the acknowledgment of the two ring buffers, there is no other important change in the "attack" file.
The offset chosen is defined at the top of the file, in the line `#define OFFSET`. To compile this file into an executable binary file, we can use the `makefile` sitting in the same directory that compile the file with GCC and the usual flags.

Before diving into the execution of the de-synchronization binary, we need to see what will actually permit us to see what is happening to the ring buffers: trigger the Falco detection rules that will permit the filling of the ring buffer and the printing of the security notifications. From the rules I read from the official documentation, these two commands trigger an event.
The first one tries to read the content of a sensitive file, triggering the message "Warning Sensitive file opened for reading by non-trusted program":

```
cat /etc/shadow > /dev/null
```

The second tries to find a file under `/etc` and trigger the message "Warning Grep private keys or passwords activities found":

```
find /home -name id_rsa 2>/dev/null
```

These commands are gathered in the bash file called `exec.sh`. Executing it will execute these two commands separated by 1 second each to populate the ring buffer.

Known FD Case

This is the case of the misconfiguration. It simulates a system administrator that created a shortcut to the eBPF maps, bypassing restrictive permissions over the highly secured access to the ring buffers. Here are the steps to re-create this:

1. Load the Falco container in docker using the previous command
2. Look for the IDs of the ring buffer created:

   ```
   sudo bpftool map list | grep ringbuf
   ```

   With only two CPU cores, this should give us a result like this:

   ```
   296: array_of_maps  name ringbuf_maps  flags 0x0
   303: ringbuf  flags 0x0
   304: ringbuf  flags 0x0
   ```

   This is made possible by the bridge made between the filesystem of the container and the machine's one with the `-v` flag.
3. Pin the maps again in a more "convenient" part of the filesystem:

   ```
   sudo bpftool map pin id [id] [path]/[name]
   ```

   The ID is the one previously displayed by the map list command and the path is chosen arbitrarily. This will create a persistent file system reference to the active selected map to a virtual path on a BPF filesystem. This has to be done for every ring buffer map generated by the Falco container.
4. Then the most critical command, bypassing the high security over the BPF programs loading and maps access:

   ```
   sudo chmod a+rwx [path]/[directory]
   ```

   Where the path leads to the directory where the ring buffer are pinned. This gives all permissions (`r`: read, `w`: write and `x`: execute) for every user (`a`: all). The same command has to be executed to the ring buffers inside the directory as they also come pinned with privileged permissions.

With these commands replicated, the misconfiguration is ready to be used to de-synchronize the consumer pointers of the ring buffer maps.

To execute the program using the `known_fd` function, either we launch the program

with no argument or the argument 0.
Here are the results of the tests:

Offset (Bytes)	Result
2	ringbuf continues as normal
8	ringbuf continues as normal
10M	ringbuf stopped

Table 8.1: Byte-by-byte minus offset results

There has been no test to see if one record could be skipped simply because the size of the payload inserted in the ring buffer may vary according to the type of event that has been triggered by Falco.

Unknown FD Case

This case assumes that the program is executed with the right capabilities to access the ring buffers. When executing the program, we need to be aware that this will affect all the maps of type ring buffer that are loaded directly inside the kernel of the machine, even the ones that are used by other programs. However the use of this method produces the same results as using the `known_fd` function. The only difference as I said before is the fact that there is no need to pin the maps in a privileged-deprived subdirectory of the machine.

8.2 Mitigation

To sum up the result we got on the Falco case, when updating the consumer position by a little size such as 2, 8 or even 100 (I also tested), the events get still printed the same way by the Falco container. There is no skip, no replay nor stopping of the ring buffers. The only exception is when I updated the ring buffer with an incredibly large value.
The Falcosecurity teams host their code on Github. The Falco application is under the `falcosecurity/falco` repository and the custom libraries they use are under the `falcosecurity/libs` repository. When investigating why I was getting these results, I realized we do not get the same output as when tested locally simply because Falco does not use the libbpf to read the packets out of their ring buffers. Indeed, the team rewrote the entire library so it could be easier for them to control.

To read the events from the ring buffers, the code sits in the `libs/userspace/libpman/src/ringbuffer.c` C file. In here, there is the function called `ringbuf_`

`_get_first_ring_event` which purpose is to read the payloads of the ring buffers' records. A line caught my intention (285):

```
g_state.cons_pos[pos] += roundup_len(len);
```

This line seems to update the consumer position of the ring buffer, since the next line (286) is

```
smp_store_release(r->consumer_pos, g_state.cons_pos[pos]);
```

There is the same function used to update the consumer position that sits in the memory of the kernel. While this function is used to update the global value, there is nowhere in this file where the consumer position called `g_state.cons_pos[pos]` is updated from the kernel side, with the function. `smp_load_acquire`. This is because Falco only relies on an internal consumer position, not visible from the outside of the library program. This means that an arbitrary change like I tried will not be taken into account by the user space program of the Falco tool.
To explain why the ring buffer stops when updating the consumer position with a very large value, it is because the bpf helper functions used to reserve and submit data in the ring buffers still use the kernel values. This means that the consumer pointer's value used by the kernel within the reserve function is the one that we corrupt and not the local one used by the user-space. This means that the checks done by the kernel to verify that the ring buffer is not full fail because of the huge value of the consumer position.
One last thing to mention is that if we corrupt the kernel's consumer pointer and the user space program updates it from its local value, the value will become again the right one and the ring buffer will continue their behavior as if nothing happened. I have been able to recreate this only because there is a 1 second delay between the events that trigger the Falco tool, which means that it was very unlikely that this savior update happens. In a context where Falco is flooded with alerts and notifications, this might be harder to obtain.

Chapter 9

Conclusion

9.1 Summary of Work

This thesis had for goal to investigate the concurrency and synchronization of the eBPF ring buffer maps, moving from the kernel C code of the eBPF subsystem and its libraries, to a rigorous formal analysis using the SPIN model checker. I have begun by translating the C code into the Promela language and defined behavioral invariants the ring buffer should not violate. However this approach was very fastidious as it introduced way too much states for the model checker to analyze, and the behavioral invariants turned out to be untouched, which seemed an expected result since the eBPF subsystem is part of the kernel, and is consequently meticulously inspected and written. The lack of protection around the consumer pointer in the libbpf library made me shift toward the investigation of the possible security issues that could arise because of the race conditions over this value. In this second approach, I only modeled the core behavior of the helper functions and the user-space libraries that interact with the ring buffer and defined some other security invariants that should not occur in this case. After the model checker had done its part of the verification, I tried to replicate these security issues in real eBPF programs and even try them out on an open source monitoring tool that uses eBPF ring buffer to report any suspicious activity.

9.2 Key Findings

During the execution of the model checker on the abstracted model of the ring buffer environment, I have found that several security invariant could be violated: packets replay and evasion, TOCTOU or even arbitrary jumps in the ring buffer's structure, which could stop entirely the map. When trying to replay these findings in real eBPF programs, I could not replicate the TOCTOU. When testing the

programs on the Falco monitoring tool to see if these issues could be replicated in real-world open source tools, only the complete stop of the ring buffer was possible. This was because the Falcosecurity team was not using the official eBPF user-space library and had developed a way for those attacks to not occur by keeping a local tracking variable of the consumer pointer.

9.3 Limitations

In all of the security invariant that could be violated by the model checker and by its reproduction in a real eBPF program, most of them were user-space vulnerabilities. Indeed, we have shown that replaying some packets could completely crash an application and that it was possible to make the user-space program miss some packets, possibly leading to the hiding of an attack if the application running is a monitoring tool. The only vulnerability that reached the kernel space is the one that makes the kernel believe that the ring buffer is completely full while it is not, creating a Denial of Service on this map and all other programs that rely on it. Moreover, the replication of the deadlock on the Falco tool was only possible with a huge misconfiguration of the system by the administrator.

9.4 Future Work

Building on the limitations identified in the thesis, several directions remain future research into eBPF security and formal verification. For example, extend the formal verification to other eBPF maps or shared user-kernel data structures, such as hash maps or perf event arrays, which are commonly used in modern eBPF programs. Another point of the research would be to find mitigation to avoid tampering with the consumer pointer from other programs accessing this portion of the memory thanks to the mmap syscall, with for example adding atomic boundary validations or cryptographic sequence checks.

Appendix A

Bash Scripts

```
1
2  #!/bin/bash
3
4  MODEL=${1:-model.pml}
5
6  echo "========================================="
7  echo " COMPILING MODEL: $MODEL"
8  echo "========================================="
9
10 spin -a "$MODEL"
11 gcc -O2 -DBITSTATE -DMEMLIM=8192 -o pan pan.c
12
13 SAFETY_PROPERTIES="rule_no_double_submit \
14 rule_no_submit_after_discard \
15 rule_no_resource_leak \
16 rule_no_overlapping_reservation \
17 rule_no_multi_consumer_race \
18 rule_no_header_corruption \
19 rule_no_wrap_around \
20 rule_ordering"
21
22 for prop in $SAFETY_PROPERTIES; do
23     echo ""
24     echo
    "================================================"
25     echo " VERIFYING: $prop"
```

```
26      echo
    "================================================"
27      ./pan -a -m100000 -w35 -N "$prop"
28  done;
```

Figure A.1: Bash Script for the Naive Approach

```
#!/bin/bash

declare -A vulnerabilities

vulnerabilities["1,1"]="Out of Order"
vulnerabilities["2,1"]="TOCTOU"
vulnerabilities["2,2"]="Overflow"
vulnerabilities["3,1"]="Deadlock"
vulnerabilities["3,2"]="Evasion"

run() {
    local i=$1
    local j=$2
    if [ $i -eq 0 ]; then
        echo ""
        echo "================================================"
        echo "Default model"
        echo "================================================"
        spin -a abstract_model.pml
        gcc -O2 -DBITSTATE -DMEMLIM=8192 -o pan pan.c
        ./pan -a -f -m100000 -w35
    else
        local filename="abstract_model_${i}_${j}.pml"
        if [ -f "./${filename}" ]; then
            description="Rule $i $j: ${vulnerabilities["$i,$j"]}"
            echo ""
            echo "================================================"
            echo "${description}"
            echo "================================================"
            spin -a "${filename}"
            gcc -O2 -DBITSTATE -DMEMLIM=8192 -o pan pan.c
            ./pan -a -f -m100000 -w35
        fi
    fi
```

```
35 }
36
37 if [ "$#" -gt 1 ]; then
38     run "$1" "$2"
39 else
40     for i in {1..3}; do
41         for j in {1..2}; do
42             run "$i" "$j"
43         done
44     done
45 fi
```

Figure A.2: Bash Script for the Asbtract Models

Bibliography

[1] Liz Rice. *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security*. O'Reilly Media, 2023 (cit. on pp. 3, 7).

[2] Linux Kernel Community. *Linux Kernel Source Tree, kernel/bpf/verifier.c*. https://elixir.bootlin.com/linux/v6.19/source/kernel/bpf/verifier.c (cit. on p. 7).

[3] Mordechai (Moti) Ben-Ari. «A Primer on Model Checking». In: *ACM Inroads* (2010). https://spinroot.com/spin/Doc/p40-ben-ari.pdf, pp. 40–47 (cit. on p. 10).

[4] Theo C. Ruys. *SPIN Beginners' Tutorial*. Online tutorial slides, https://spinroot.com/spin/Doc/SpinTutorial.pdf. 2002 (cit. on pp. 11, 12).

[5] Hao Zheng. *A Tutorial on Model Checker SPIN*. https://cse.usf.edu/~haozheng/teach/cda5416/slides/spin-tutorial.pdf. 2002 (cit. on p. 12).

[6] Andrii Nakryiko. *BPF Ring Buffer*. Andrii Nakryiko's Blog. Oct. 2020. URL: https://nakryiko.com/posts/bpf-ringbuf (cit. on p. 37).

[7] eBPF Foundation. *eBPF Documentation*. https://ebpf.io/what-is-ebpf/.

[8] Linux Kernel Community. *Linux Kernel Source Tree, kernel/bpf/ringbuf.c*. https://elixir.bootlin.com/linux/v6.19/source/kernel/bpf/ringbuf.c.

[9] Linux Kernel Community. *Linux Kernel Source Tree, tools/lib/bpf/ringbuf.c*. https://elixir.bootlin.com/linux/v6.19/source/tools/lib/bpf/ringbuf.c.

[10] Linux Kernel Community. *bpf-helpers(7) — Linux manual page*. https://www.man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[11] Linux Kernel Community. *libbpf Overview*. https://libbpf.readthedocs.io/en/latest/libbpf_overview.html.