POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Design and implementation of a verifier for compiling eBPF code

Supervisors

Prof. Riccardo SISTO

Rosario RIZZA

Candidate

Oscar MARZO

March 2026

Summary

Extended Berkeley Packet Filter (eBPF) is a powerful Linux kernel technology that enables the safe execution of user-defined programs inside kernel space. To guarantee kernel stability and security, every eBPF program must pass a strict static verification process performed by the eBPF verifier before execution. Although effective, the verifier operates on compiled bytecode and often produces error messages that are difficult for developers to interpret and trace back to the original source code. As a consequence, debugging verifier rejections represents a significant challenge in the eBPF development workflow.

This thesis addresses this problem by proposing a source-level static analysis approach aimed at anticipating eBPF verifier errors directly on C code. The objective is to provide earlier and more understandable feedback to developers, shifting error detection to an earlier stage of the development pipeline.

The proposed solution is based on the Clang Static Analyzer framework from the LLVM ecosystem. Custom static analysis checkers were designed and implemented to model a subset of the safety rules enforced by the eBPF verifier. In particular, the developed checkers target the following verifier error categories: missing or incompatible license declarations, incorrect helper function parameter types, out-of-bounds accesses on global arrays, and unsafe accesses to packet memory, including prohibited pointer arithmetic on packet boundary pointers.

The checker architecture is modular and integrates naturally into the Clang analysis pipeline, enabling the analysis of eBPF C programs directly at source level, without requiring compilation to bytecode or program execution. By operating directly on the source code, the proposed framework is able to generate precise and contextualized diagnostics that are easier to understand and correct compared to the messages produced by the kernel verifier.

Acknowledgements

ACKNOWLEDGMENTS

Table of Contents

List of Tables

List of Figures

Acronyms

eBPF
: Extended Berkeley Packet Filter

LLVM
: Low Level Virtual Machine

CSA
: Clang Static Analyzer

CFG
: Control Flow Graph

IR
: Intermediate Representation

AST
: Abstract Syntax Tree

JIT
: Just-In-Time

Introduction

eBPF, or extended Berkeley Packet Filter, is a Linux kernel technology that enables the safe execution of user defined programs directly inside kernel space. Due to the privileged execution environment, every eBPF program must pass a strict static verification phase before being loaded and executed. This verification is performed by the kernel eBPF verifier, which symbolically analyzes the compiled eBPF bytecode to ensure memory safety, correct pointer usage, valid helper function invocations, bounded control, proper program termination, and license compliance. These checks are essential to preserve kernel stability and security. While the eBPF verifier is fundamental to guarantee kernel safety, it also represents one of the main obstacles in the eBPF development workflow. Verifier diagnostics are generated at bytecode level and expose internal states, abstract register types, and symbolic execution paths that are often difficult to interpret and hard to correlate with the original source code. As a result, debugging verifier rejections frequently becomes an iterative and time-consuming process, even in the presence of relatively simple programming errors. This mismatch between source-level programming and bytecode-level verification has a significant impact on developer productivity.

The goal of this thesis is the definition of a source-level static analysis approach aimed at improving the eBPF development workflow by anticipating verifier rejections before program compilation and loading. Unlike the Linux eBPF verifier, which operates exclusively on compiled bytecode and produces low-level diagnostics, the proposed approach shifts part of the verification effort to the source-code level, targeting eBPF programs written in C, enabling earlier, clearer, and more actionable feedback for developers. This work introduces a complementary verification layer that operates at source level and models a representative subset of the safety rules enforced by the eBPF verifier. Rather than replacing the kernel verifier, the proposed approach aims to anticipate its most common rejection causes, allowing developers to identify potential verification failures earlier in the development pipeline, before compilation to eBPF bytecode. Two custom static analysis checkers have been designed and implemented, selecting and modeling four error classes

that frequently lead to program rejection and that can be effectively reasoned about using static analysis on source code: license-related violations, misuse of helper functions, out-of-bounds accesses to global arrays, and unsafe packet memory accesses. The first category addresses license compatibility violations arising from the use of GPL-restricted helper functions. The second concerns incorrect helper function usage: eBPF helper functions are tightly constrained by program type and argument signatures, and violations of these constraints represent a common source of verifier errors. The third category covers unsafe accesses to global arrays. By reasoning about both constant and symbolic indices, the framework is able to detect potential out-of-bounds accesses in cases where index bounds are not sufficiently constrained by program logic. Finally, the fourth category targets packet memory accesses: since packet data requires explicit bounds checks to satisfy the verifier, the proposed approach tracks packet-derived pointers and verifies that memory accesses are performed only within regions that have been previously validated, directly addressing a frequent and difficult-to-debug class of verifier errors in networking and XDP programs.

Chapter 1 provides the necessary background knowledge on the eBPF technology. It introduces the main concepts behind eBPF, its architecture, and its role within the Linux kernel.
Chapter 2 focuses on the eBPF verifier. It describes the verification process, explaining the safety checks that are applied and the main limitations developers face when interacting with the verifier. The content presented in this two chapters is based on Liz Rice,Learning eBPF [1], eBPF Official Documentation [2], the eBPF Verifier Documentation [3] and the eBPF Helper Function Documentation [4]
Chapter 3 presents the tools and technologies used in this work, the Clang and LLVM infrastructures and on the principles of static analysis. The chapter also introduces the Clang Static Analyzer framework, which provides the foundation for the analysis techniques adopted in this thesis. This chapter use the informations found in Bruno Cardoso Lopes and Rafael Auler, Getting Started with LLVM: Core Libraries [5], the official LLVM Documentation [6] and the official Clang Doxygen documentation of the Clang Static Analyzer framework [7].
Chapter 4 describes the implementation developed in this work. In particular, it presents the designed checkers and explains the classes of errors they are intended to detect when analyzing eBPF programs.
Chapter 5 presents the experimental evaluation of the proposed approach. It describes the test cases used to evaluate the effectiveness of the implemented checkers and discusses the obtained results.
Chapter 6 concludes the thesis by summarizing the main contributions of this work and discussing possible directions for future improvements and research.

Chapter 1

Background on eBPF

This chapter provides the necessary background knowledge on eBPF technology. The content presented here is based on the following references: Liz Rice,Learning eBPF [1], eBPF Official Documentation [2], eBPF Verifier [3]

1.1 eBPF History

The origins of eBPF (extended Berkeley Packet Filter) trace back to its predecessor, the Berkeley Packet Filter (BPF). BPF was originally designed to provide a high-performance, user-programmable mechanism for filtering network packets, particularly for real-time packet capture and monitoring. eBPF extends this functionality beyond simple packet filtering: it allows users to load and execute custom programs directly within the operating system kernel, thereby extending and enhancing the kernel's behavior.

Traditionally, observing the behavior of an application requires modifying its source code, for instance by adding log statements. eBPF, however, enables this kind of monitoring without altering the target application's source code. More importantly, it does so without making permanent modifications to the kernel itself, which can be difficult due to approval processes and stability concerns. eBPF programs are loaded into the kernel in a safe manner, can be attached to and detached from events dynamically, and are triggered in response to specific kernel or application events.

1.2 Linux Kernel

The Linux kernel is the core software layer between applications and the underlying hardware. eBPF programs run inside the Linux kernel. When an eBPF program is loaded, the eBPF verifier performs a series of steps to ensure that the program it is safe to execute. This includes checking for memory safety, ensuring proper termination, and validating access to kernel data structures. Once verified, the program can be attached to specific kernel hooks, such as networking events, tracepoints, kprobes, or cgroups.

This design allows eBPF to extend kernel functionality without modifying kernel source code. Programs can be dynamically loaded, attached, and detached at runtime, providing powerful capabilities for observability, performance monitoring, and security enforcement. Because the verifier enforces strict safety checks, eBPF programs cannot compromise kernel stability, making it possible to safely run user-defined code in kernel space.

1.3 Compilation Pipeline

eBPF programs are typically written in C or Rust and then compiled using the LLVM Clang compiler. Initially, Clang parses the source code into an Abstract Syntax Tree (AST), which captures the structure and syntax of the program. The AST is then converted into the LLVM Intermediate Representation (IR), which serves as a platform-independent, low-level representation of the program. The LLVM IR is further compiled into eBPF bytecode, which is a 64-bit instruction set designed for the eBPF virtual machine.

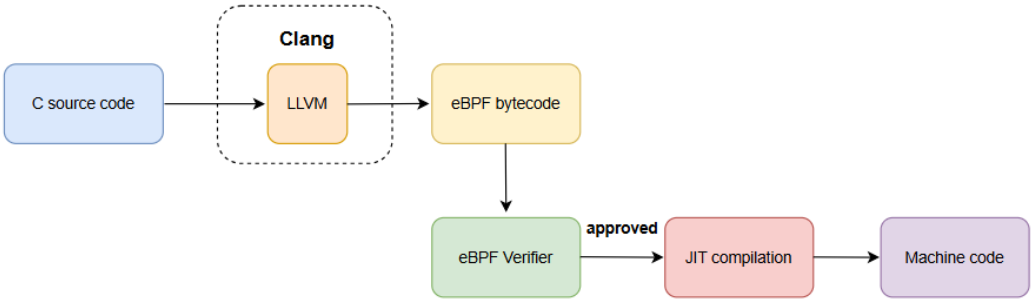

Figure 1.1: Compiling pipeline.

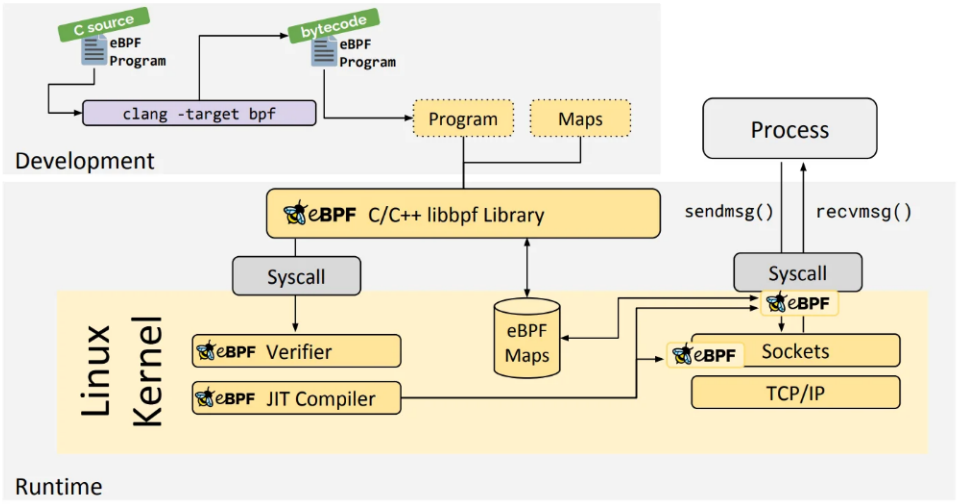

Figure 1.2: Overview of the eBPF program compilation, loading, verification, and execution pipeline. From the official eBPF documentation [2].

Each instruction consists of an opcode, source and destination registers, a signed offset, and an immediate value, represented in the following structure:

```
struct bpf_insn {
    __u8 code;        /* opcode */
    __u8 dst_reg:4;   /* destination register */
    __u8 src_reg:4;   /* source register */
    __s16 off;        /* signed offset */
    __s32 imm;        /* signed immediate constant */
};
```

The compiled bytecode is then loaded into the Linux kernel. During this loading phase, the eBPF verifier performs a thorough static analysis to ensure that the program is safe to execute within kernel space. It checks for issues such as invalid memory accesses, incorrect pointer dereferencing, program termination, unbounded loops, controls on helper functions [4]. Only programs that successfully pass the verification stage can be attached and executed.

Once verified, the eBPF bytecode is executed by the eBPF virtual machine. The kernel applies Just-In-Time (JIT) compilation to translate the bytecode into native machine code. Also, `bpftool` is a command-line utility in Linux for interacting with

eBPF programs and maps. It allows loading and attaching programs, manipulating maps, retrieving information about eBPF objects and list currently loaded programs.

Example using bpftool

A concrete example of loading an eBPF program:

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

int counter = 0;

SEC("xdp")
int hello(struct xdp_md *ctx) {
    bpf_printk("Hello World %d", counter);
    counter++;
    return XDP_PASS;
}

char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

1. **Compile:**

   ```
   clang -O2 -target bpf -g -c hello_world.bpf.c -o hello_world.bpf.o
   ```

2. **Inspect the bytecode:**

   ```
   llvm-objdump -S hello_world.bpf.o

   hello_world.bpf.o: file format elf64-bpf

   Disassembly of section xdp:

   0000000000000000 <hello>:
   0:       7b 1a f8 ff 00 00 00 00 *(u64 *)(r10 - 0x8) = r1
   1:       18 01 00 00 00 00 00 00 r1 = 0x0 ll
   3:       79 14 00 00 00 00 00 00 r4 = *(u64 *)(r1 + 0x0)
   4:       18 01 00 00 00 00 00 00 r1 = 0x0 ll
   6:       7b 1a e8 ff 00 00 00 00 *(u64 *)(r10 - 0x18) = r1
   ```

```
7:       61 13 00 00 00 00 00 00 w3 = *(u32 *)(r1 + 0x0)
8:       18 01 00 00 00 00 00 00 r1 = 0x0 ll
10:       b4 02 00 00 0f 00 00 00 w2 = 0xf
11:       8d 04 00 00 00 00 00 00 callx r4
12:       79 a2 e8 ff 00 00 00 00 r2 = *(u64 *)(r10 - 0x18)
13:       7b 0a f0 ff 00 00 00 00 *(u64 *)(r10 - 0x10) = r0
14:       61 21 00 00 00 00 00 00 w1 = *(u32 *)(r2 + 0x0)
15:       04 01 00 00 01 00 00 00 w1 += 0x1
16:       63 12 00 00 00 00 00 00 *(u32 *)(r2 + 0x0) = w1
17:       b4 00 00 00 02 00 00 00 w0 = 0x2
18:       95 00 00 00 00 00 00 00 exit
```

3. **Loading into the kernel:**

```
bpftool prog load hello_world.bpf.o /sys/fs/bpf/hello_world
```

4. **Attaching the program:**

```
bpftool net attach xdp pinned /sys/fs/bpf/hello_world dev enp0s3 sec hello
```

5. **Listing loaded programs:**

```
bpftool prog show
```

```
2: tracing  name hid_tail_call  tag 7cc47bbf07148bfe  gpl
        loaded_at 2025-12-20T02:54:51+0000  uid 0
        xlated 56B  jited 133B  memlock 4096B  map_ids 2 btfid 6
5: cgroup_device  name sd_devices  tag e3dbd137be8d6168  gpl
        loaded_at 2025-12-20T02:55:14+0000  uid 0
        xlated 504B  jited 311B  memlock 4096B
...
3544: xdp  name hello  tag 94468b3a849e5748  gpl
        loaded_at 2025-12-20T12:28:15+0000  uid 0
        xlated 96B  jited 64B  memlock 4096B  map_ids 767,768
        btf_id 6381
```

6. **Detaching / removing the program:**

```
rm -f /sys/fs/bpf/hello_world
```

Chapter 2

The eBPF Verifier

The eBPF verifier represents a fundamental component of the Linux kernel's eBPF infrastructure, serving as the primary safety mechanism that ensures eBPF programs can execute safely within kernel space. Given that eBPF programs operate with elevated privileges, even minor implementation flaws could result in catastrophic consequences such as system crashes, memory corruption, or exploitable security vulnerabilities. To prevent such scenarios, the verifier performs a comprehensive static analysis at program load time, rejecting any program that cannot be proven safe. This chapter examines the eBPF verifier in detail, focusing on its internal design, the verification process, and the safety guarantees it enforces.

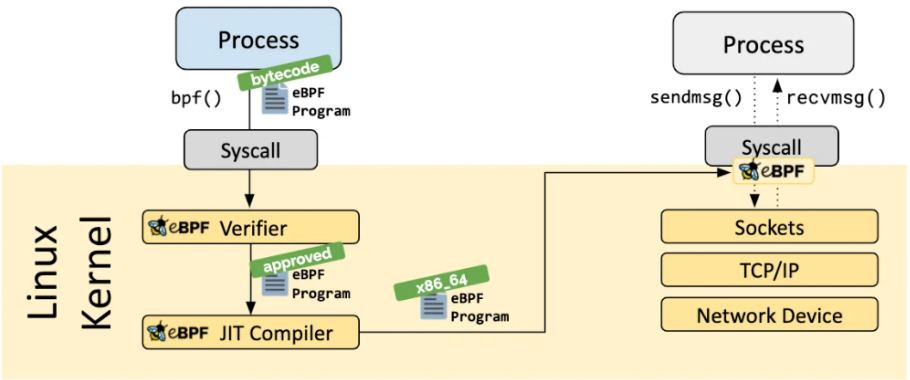

Figure 2.1: Overview of the eBPF verification pipeline. From the official eBPF documentation [2].

The content presented here is based on the following references: Liz Rice,Learning eBPF [1], eBPF Official Documentation [2], eBPF Verifier [3], eBPF Helper Functions Documentation[4]

2.1 The Verification Process

The eBPF verifier analyzes a program exploring all its possible execution paths. Rather than executing the program, the verifier symbolically evaluates each instruction in sequence, maintaining an abstract representation of the program state. Verification is performed entirely at load time and must successfully complete before the program is allowed to run in kernel space.

2.1.1 Register State and Range Tracking

The verifier processes the eBPF bytecode one instruction at a time, symbolically simulating the effects of each instruction on the program state rather than executing it concretely. During this analysis, it maintains a precise abstraction of the state of every register using the `bpf_reg_state` structure. Each instruction is evaluated to determine whether it may lead to unsafe behavior, such as invalid memory accesses, illegal pointer arithmetic, or violation of kernel safety constraints.

The state of each register is represented by an instance of `struct bpf_reg_state`, which captures both the type of the value stored in the register and the range of values that the register may hold at a given program point.

Each register is associated with a `bpf_reg_type`, which describes the meaning and allowed usage of the value stored in the register. The main categories include:

- `NOT_INIT`: indicating that the register has not been written
- `SCALAR_VALUE`: representing a scalar value
- `PTR_TO_CTX`: a pointer to the `bpf_context`
- `CONST_PTR_TO_MAP`: a pointer to a `struct bpf_map`, arithmetic on this pointer is forbidden.
- `PTR_TO_MAP_VALUE`: a pointer to the value stored in a map element.

- `PTR_TO_MAP_VALUE_OR_NULL`: representing either a valid pointer to a map value or `NULL`
- `PTR_TO_STACK`: pointer to frame pointer
- `PTR_TO_PACKET`: pointer to the beginning of the packet data (`skb->data`)
- `PTR_TO_PACKET_END`: pointer to the end of the packet data (`skb>data`$_e nd), arithmeticonthis$
- `PTR_TO_SOCKET`: pointer to (`struct bpf`$_s ock_o ps), implicitlyref counted.$
- `PTR_TO_SOCKET_OR_NULL`: representing either a valid socket pointer or `NULL`

However, a pointer may be offset from this base (as a result of pointer arithmetic), and this is tracked in two parts: the 'fixed offset' and 'variable offset'. The former is used when an exactly-known value (e.g. an immediate operand) is added to a pointer, while the latter is used for values which are not exactly known. The variable offset is also used in SCALAR$_V ALUEs, totracktherangeofpossiblevaluesintheregister.$

The verifier's knowledge about the variable offset consists of:

minimum and maximum values as unsigned

minimum and maximum values as signed

knowledge of the values of individual bits, in the form of a 'tnum': a u64 'mask' and a u64 'value'. 1s in the mask represent bits whose value is unknown; 1s in the value represent bits known to be 1. Bits known to be 0 have 0 in both mask and value; no bit should ever be 1 in both. For example, if a byte is read into a register from memory, the register's top 56 bits are known zero, while the low 8 are unknown - which is represented as the tnum (0x0; 0xff). If we then OR this with 0x40, we get (0x40; 0xbf), then if we add 1 we get (0x0; 0x1ff), because of potential carries.

2.1.2 Control-Flow Exploration and Branching

To ensure the safety of an eBPF program, the verifier must consider all possible execution paths that may arise due to conditional branches and jumps. Rather than executing the program concretely, the verifier symbolically explores the control flow by simulating the effects of each instruction on the program state.

Whenever the verifier encounters a branch instruction, it creates a copy of the current state of all registers and saves it on an internal stack. One of the possible

execution paths is then explored, while the alternative paths are deferred for later analysis. The verifier continues evaluating instructions along the chosen path until it reaches a program termination point, such as a return instruction, or until it reaches the maximum number of instructions it is allowed to process, which is currently limited to one million instructions.

Once a path has been fully analyzed, the verifier pops a previously saved state from the stack and resumes verification from the corresponding branch point. This process is repeated until all reachable execution paths have been explored. If, during this analysis, the verifier encounters any instruction that could potentially result in an unsafe operation, such as invalid memory access or illegal pointer usage, the entire program is rejected and verification fails.

In addition conditional branches also refine the verifier's knowledge of register values. When a branch condition involves a comparison on a `SCALAR_VALUE`, the verifier updates the value bounds differently along each branch. For example, if a register is compared against a constant using an unsigned comparison (e.g., `> 8`), the verifier infers that the register must be at least 9 along the true branch, while it must be at most 8 along the false branch, so updating the signed minimum or maximum bounds of the register.

2.1.3 State Pruning and Optimization

Exploring all possible execution paths can be computationally expensive. To mitigate this issue, the verifier employs an optimization technique known as *state pruning*. During verification, the verifier records the register states observed at specific instructions. If the verifier later reaches the same instruction with an equivalent register state, it can safely skip further analysis of that path, as its safety has already been established. This optimization significantly reduces verification time while preserving correctness.

2.2 Verifier Control

2.2.1 License Verification

As part of the verification process, the eBPF verifier control license on programs that invoke kernel helper functions. In particular, if an eBPF program makes use

of a helper function that is marked as GPL-restricted, the program itself must declare a GPL-compatible license. Failure to satisfy this requirement causes the verifier to reject the program.

Each helper function is associated with a boolean attribute,`gpl_only`, which indicates whether the helper is restricted to GPL-licensed programs.

Also, if the license string provided by the eBPF program does not match any of the accepted values, and the program attempts to invoke a GPL-only helper, the verification fails. The set of GPL-compatible license identifiers is defined in the Linux documentation [8].

2.2.2 Validating BPF Helper Functions

eBPF programs can invoke a predefined set of helper functions, which allow them to interact with the kernel and its internal data structures, as well as to retrieve information. However, the use of these helper functions is subject to strict verification.

First, the verifier ensures that each helper function is permitted for the specific program type, as defined by the program section (`SEC`). If a helper function is not allowed for that program type, the program is immediately rejected.

If this initial check succeeds, the verifier then validates the arguments passed to the helper function. Each parameter must respect the type defined by the helper's prototype, and if there is a mismatch in parameter type the verifier reject the program.

2.2.3 Invalid Memory Access

Whenever an eBPF program accesses a map or a global array, the verifier checks that the index used for the access is within the bounds of the underlying data structure. Specifically, the verifier must be able to prove that the index cannot exceed the declared size of the map value or array.

For example, if a program defines a global array of size 10, any access using an index smaller than 0 or greater than or equal to 10 is considered invalid. In such cases, the verifier rejects the program, as an out-of-bounds access to undefined behavior or memory corruption.

To perform these checks, the verifier associates at the register a verified range of possible values. In this example, the valid range is $[0, 9]$, and the verifier ensures that the index used for the access is in this range.

2.2.4 Packet Bounds Checking

A similar reasoning applies when an eBPF program attempts to access packet data through the `ctx->data` (start of packet) and `ctx->data_end` (the end of the packet) pointers. For each packet pointer, the verifier tracks a verified access range, ensuring that any memory access remains within the bounds of the packet buffer. Any direct dereference of `data_end`, as well as any pointer arithmetic performed on it, is always rejected by the verifier,

Before accessing packet data through `ctx->data`, the program must perform an explicit bounds check against `ctx->data_end`. This check refines the verifier's knowledge of the accessible range and guarantees that packet accesses are safe.

```
int test_bounds_rejected(struct xdp_md *ctx) {
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  s[0] = *((unsigned char *)data);
  bpf_printk("%s", s);

  return XDP_PASS;
}
```

The program above is rejected by the verifier because it attempts to read packet data without first performing a bounds check. Since the verifier cannot prove that the access is within the packet boundaries, the access is considered unsafe.

```
int test_bounds_accepted(struct xdp_md *ctx) {
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  if (data + 20 > data_end) {
```

```
    return XDP_DROP;
  }

  s[0] = *((unsigned char *)(data + 12));
  bpf_printk("%s", s);

  return XDP_PASS;
}
```

In this case, the program is accepted by the verifier. The explicit bounds check guarantees that the packet is at least 20 bytes long, and the access at offset 12 is therefore proven to be safe ($12 < 20$).

2.2.5 Pointer Dereferencing and Null Checks

Dereferencing a null pointer would result in a kernel crash. For this reason, every pointer must be explicitly checked for validity before it is dereferenced. The eBPF verifier reject any program that attempts to access memory through a pointer whose was not performed null check

For example, the following code is accepted by the verifier because it performs an explicit null check:

```
if (p != 0) {
 char a = p->message[0];
 bpf_printk("%d", cc);
}
```

In addition some helper functions, such as `bpf_probe_read_kernel()`, are designed to safely handle potentially null pointers by taking unsafe pointers as arguments and performing necessary checks internally

2.2.6 Program Termination

The verifier must statically guarantee that the program it analyzes can be executed from start to end, ensuring that its execution always terminates. To enforce this

property, the verifier imposes an upper bound on the total number of instructions it is allowed to process during analysis, currently set to one million instructions.

If the verifier does not reach a valid program termination before this limit is exceeded, the program is rejected. This situation can easily arise in programs that do not terminate, for example due to the presence of infinite loops.

2.2.7 Checking Return Value

In every eBPF program, the return code must be placed in register R0. When this register is never explicitly initialized, the verifier rejects the program.This issue can be resolved by setting a return value or by invoking a helper function, `bpf_printk()`, which initializes R0.

2.2.8 Invalid Bytecode Instructions

The eBPF verifier detects and rejects bytecode instructions that are considered invalid or unsupported, as their execution could compromise kernel stability. These errors may ccur when eBPF bytecode is written manually or when atomic operations are used on kernels that do not yet support them. Anyway, modern compilers generally do not emit invalid instructions.

2.2.9 Unreachable Instructions

The verifier also rejects programs that have unreachable instructions. Oftentimes, these will get optimized out by the compiler anyway.

2.2.10 Program Size Limit

The eBPF verifier imposes a upper bound on program size. Any program whose bytecode exceeds one million instructions is rejected. Although most eBPF programs remain well below this threshold, overly complex implementations or improperly written code—such as programs with unrolled loops—may easily exceed the limit. During verification, the verifier explicitly counts every instruction, and once the maximum limit number is surpassed, the program is rejected

Chapter 3

Static Analysis with Clang

Static analysis is a powerful technique for detecting program errors without executing the code. In the context of eBPF, static analysis can provide early feedback on potential verifier errors, offering developers a chance to correct mistakes before attempting to load programs into the kernel. Clang, part of the LLVM project, provides a robust framework for implementing static analyzers that operate on C and C++ code, making it an ideal choice for eBPF analysis.

The content of this chapter is based on the reference book [5], the official LLVM Documentation [6] and on the official Clang Doxygen documentation of the Clang Static Analyzer framework [7].

3.1 LLVM Overview

LLVM is a collection of modular and reusable compiler and toolchain technologies designed to support the construction of modern compilation frameworks. The project originated as a research initiative at the University of Illinois, with the objective of developing an SSA-based compilation strategy capable of enabling both static and dynamic compilation for a wide range of programming languages.

The LLVM ecosystem is composed of several major subprojects. In the context of this thesis, the most relevant ones are:

- **LLVM Core:** The LLVM Core provide a source- and target-independent optimization framework together with code generation support for many processor architectures. These libraries are built around the LLVM Intermediate Representation (LLVM IR), which serves as a well-defined and flexible program representation.
- **Clang:** Clang represents the LLVM-native frontend for C, C++, and Objective-C. In addition to offering fast compilation and high-quality diagnostic messages, Clang provides tools for source-level analysis tools such as Clang tidy and the Clang Static Analyzer

The informations presented in this section are derived from the official LLVM project documentation [6].

3.2 LLVM Compilation Infrastructure

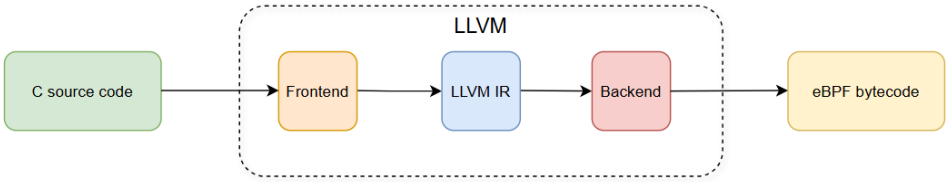

Figure 3.1: Overview of LLVM infrastructure

The LLVM infrastructure is organized into three main components:

- **Frontend:** This phase translates high-level programming languages such as C, C++, and Objective-C into the LLVM Intermediate Representation. It includes lexical analysis, syntactic parsing, semantic analysis, and the generation of LLVM IR code.
- **Intermediate Representation (IR):** LLVM IR is available in both human-readable and binary formats. Dedicated tools and libraries provide support for

IR generation, assembly, and disassembly. Most of the compiler optimizations are applied at this level by the LLVM optimization framework.

- **Backend:** This stage is responsible for final code generation. It transforms LLVM IR into target-specific assembly or object code. Backend tasks include register allocation, loop optimizations, peephole optimizations, and other architecture-dependent transformations.

3.2.1 Clang Frontend

Clang is adopted as the frontend in this work because eBPF programs are usually written in a C language. In LLVM, each programming language requires a dedicated frontend to translate source code into LLVM IR.

To transform a source code program into LLVM IR bitcode, there are a few intermediate steps the source code must pass through.

Figure 3.2: Clang frontend stages, from the reference book [5].

- **Lexical analysis:** analyzes the textual source code and separates language constructs into a stream of words and tokens, while discarding elements such as comments, whitespace, and tab characters. Each produced token must conform to the language grammar, and reserved keywords are mapped to corresponding internal compiler representations.
- **Syntactic analysis:** This phase groups the input tokens to form expressions, statements, and function bodies. The parser takes a stream of tokens as input and produces an Abstract Syntax Tree (AST), which represents declarations, statements, and types in a hierarchical structure. In Clang, AST nodes are organized around three core classes: `Decl`, `Stmt`, and `Type`. All other AST node classes are derived from these three fundamental base classes.
- **Semantic analysis:** Semantic analysis verifies that the program does not violate the language type system by using a symbol table. This table stores, among other information, the associations between identifiers and their corresponding types. A common approach consists in performing type checking

by traversing the AST after parsing. Clang, however, follows a different strategy: instead of performing a separate traversal, it executes type checking during AST node construction, integrating semantic validation directly into the parsing process.

- **Generating LLVM IR code:** After parsing and semantic analysis, the Clang frontend produces a complete Abstract Syntax Tree representing the entire translation unit. At this point, the frontend invokes the LLVM code generation phase, which traverses the AST and translates each declaration and statement into equivalent LLVM Intermediate Representation (LLVM IR) instructions.

  At the end of the frontend pipeline, the source code provided as input is fully translated into LLVM Intermediate Representation and the frontend stage is complete. The compilation process can then proceed with IR-level optimizations, followed by target-specific code generation performed by the backend.

3.2.2 LLVM Intermediate Representation

LLVM Intermediate Representation (IR) is the backbone that connects frontend and backend, allowing LLVM to support multiple programming languages and multiple target architectures. It is at this stage that most target-independent optimizations are applied.

LLVM IR is a normalized and language-independent program representation used by the compiler to analyze, optimize, and transform source code. Within the LLVM framework, it provides a low-level, strongly typed, and target-independent representation based on the Single Static Assignment (SSA) form, which facilitates precise analysis and efficient optimization.

3.2.3 Backend

The backend consists of a sequence of analysis and transformation passes responsible for converting the LLVM Intermediate Representation (IR) into target-specific assembly or object code. LLVM provides backend support for a wide range of architectures, including ARM, AArch64, Hexagon, MSP430, MIPS, NVIDIA PTX, PowerPC, R600, SPARC, SystemZ, x86, and XCore. The backend pipeline is

organized into multiple stages, which ultimately lead to the emission of machine-level code.

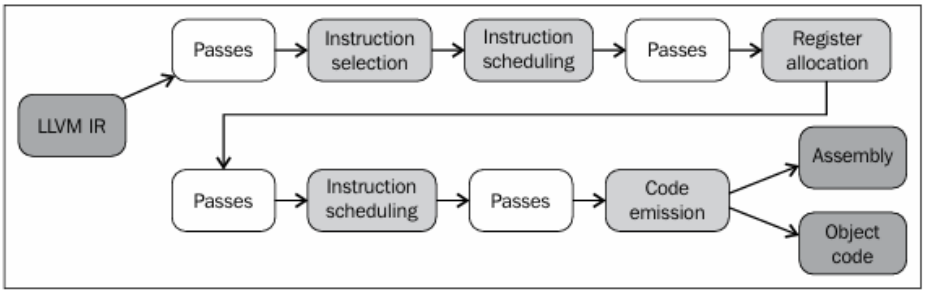

Figure 3.3: Backend stages, from the reference book [5].

In the context of eBPF, after all optimization passes have been applied, the LLVM IR is translated into eBPF bytecode. It is important to note that Just-In-Time compilation is performed at execution time: once the eBPF bytecode successfully passes the kernel verifier, the Linux kernel itself is responsible for JIT-compiling it into native machine code.

3.3 Clang Static Analyzer

The Clang Static Analyzer (CSA) is an automated tool designed to perform static analysis on source code without executing the program. Its main objective is to detect a wide range of common programming errors in C, C++, and Objective-C applications at an early stage of development, before the code is compiled or executed.

Although it is tightly integrated into the Clang ecosystem, the Clang Static Analyzer is not part of the standard compilation pipeline. Instead, it is a separate analysis framework that exploits the data structures produced by the Clang frontend. In particular, it operates on the Abstract Syntax Tree (AST), the Control Flow Graph (CFG), and symbol information generated after parsing and semantic analysis, but before the LLVM IR generation phase.

The analysis is performed through a collection of checkers, where each checker is responsible for detecting a specific class of programming errors. These checkers generate detailed diagnostic reports, similar to compiler warnings, but with deeper

semantic information. In this way, the static analyzer supports developers in identifying defects early in the development process, reducing the need to postpone bug detection to runtime testing. In addition to the predefined checkers provided by Clang, it is possible for developers to implement custom checkers

3.3.1 Symbolic Execution

The Clang Static Analyzer is a path-sensitive and interprocedural analysis framework. It explores multiple execution paths of a program by symbolically evaluating program statements, allowing it to detect potential bugs such as null pointer dereferences, memory leaks, and incorrect API usage. By analyzing different paths independently, CSA is able to reason about program behavior under multiple execution conditions.

During the analysis, CSA builds a graph of reachable program states, which represents all possible execution paths that may be taken by the program without actually executing it. Each node in this graph corresponds to a symbolic program state, while edges represent transitions caused by program statements.

At the beginning of the analysis, the engine assigns symbolic initial states to program variables while traversing the code in execution order. When a control-flow construct is encountered, such as a conditional branch, the analysis splits into multiple paths. Each path is then explored independently, and the corresponding program states are added to the reachable program states graph.

CSA performs symbolic execution according to the following principles:

- **Path-sensitive analysis:** Each possible execution path is analyzed independently.
- **Symbolic values:** Variables are represented by symbolic values instead of concrete ones, enabling the modeling of multiple states.
- **Constraint tracking:** Conditional expressions refine the symbolic values along different paths.
- **Bug detection:** Violations such as null-pointer dereferences, uninitialized reads, and type mismatches are reported.

In the context of eBPF, CSA can be extended to detect source-level patterns that

would be rejected by the kernel verifier, providing early diagnostics directly at the C code level.

3.3.2 Checker Infrastructure

The Clang Static Analyzer provides an extensible infrastructure for implementing custom checkers. Each checker encapsulates the logic required to detect a specific class of programming errors or domain-specific violations.

This infrastructure offers several key components:

- **Predefined checker APIs:** Interfaces that allow checkers to hook into analysis points such as function entry, variable access, and control-flow transitions.
- **Custom bug reporters:** Mechanisms for generating user-friendly diagnostics, including source locations and detailed descriptions.
- **Shared state management:** Facilities to track and propagate program states across different execution paths and function boundaries.
- **Integration with Clang compilation:** Checkers can be executed automatically during compilation, producing warnings and errors directly in the compiler output.

This infrastructure enables the development of domain-specific checkers, such as those for eBPF license validation, map access verification, helper function usage, and packet boundary safety.

3.3.3 Checker Callbacks

Checkers in the Clang Static Analyzer react to specific analysis events through a mechanism known as callbacks. These callbacks allow a checker to observe and analyze different stages of program execution.

Common callback methods include:

- `checkPreStmt`: Invoked before a statement is symbolically executed.
- `checkPostStmt`: Invoked after a statement has been symbolically executed.
- `checkPreCall` and `checkPostCall`: Invoked respectively before and after a function call.
- `checkDeadSymbols`: Invoked when symbols become unreachable or go out of scope.

By registering these callbacks, a checker can monitor variable usage, function arguments, and memory accesses, enabling the detection of patterns that violate safety or correctness constraints, as described in the official Clang Doxygen documentation [7].

Chapter 4

Design and Implementation of eBPF Checkers

4.1 Motivation and Objectives

The checkers presented in this chapter aim to identify a class of programming errors that are typically detected later by the eBPF verifier. These issues include license-related violations, out-of-bounds accesses to global arrays, incorrect usage of eBPF helper functions, and unsafe packet data accesses.

Unlike the eBPF verifier, which operates on the generated bytecode, the proposed checkers perform static analysis directly on the eBPF C source code. Their primary objective is to detect potential verification failures at an earlier stage of the development process, providing developers with immediate and more comprehensible feedback.

In particular, the license checker verifies compliance with GPL requirements imposed by specific helper functions. The array out-of-bounds checker analyzes global array accesses to detect invalid index usage. The helper function type checker ensures that the arguments passed to eBPF helper functions match the types required by their respective prototypes. Finally, the packet checker validates that any pointer used to access packet data refers to a memory region that lies within the bounds explicitly verified by the program.

By anticipating errors that would otherwise be rejected by the eBPF verifier, these

static checkers improve code safety, reduce debugging time, and enhance the overall development experience for eBPF programs.

4.2 Files and Checker Registration

The implementation of the proposed eBPF static analysis framework is organized around two main components, each corresponding to a different class of verification tasks performed by the Linux eBPF verifier.

The first component, referred to as the Verifier Checker. These include GPL license compliance, correct usage of helper functions, and safe accesses to global arrays. Its internal structure groups together the logic for analyzing license declarations, validating helper prototypes, and detecting out-of-bounds array accesses, thus providing a unified representation of the verifier-related checks performed by the framework.

The second component, called the Packet Checker, is dedicated to the analysis of memory safety with respect to packet data. In particular, it tracks pointers derived from packet buffers and verifies that all memory accesses are performed within regions that have been previously validated by explicit bounds checks.

Both checkers are implemented as extensions of the Clang Static Analyzer and are registered under a dedicated analysis package.he checkers are grouped within the alpha.ebpf package. Within this package, the Verifier Checker and the Packet Checker are exposed as two independent analysis modules, enabling users to run either or both depending on the type of verification they require.

4.3 License Checker

Problem Description

Certain eBPF helper functions are restricted to programs that declare a GPL-compatible license. When an eBPF program is loaded, the verifier checks whether any of the helper functions used by the program are marked as *GPL-only*, in particular each helper functions has a boolean parameter *GplOnly* setted to true o false. If this condition is true, the program is required to explicitly declare a GPL-compatible license; otherwise, the program is rejected during the loading

phase.

An example of the bpf_map_update_elem helper function prototype is reported in the Linux eBPF helper documentation, which reflects the definitions provided by the Linux kernel [4]. In this case, the gpl_only field is set to false, meaning that the function can be used by programs that do not declare a GPLcompatible license.

```
const struct bpf_func_proto bpf_map_update_elem_proto = {
    .func        = bpf_map_update_elem,
    .gpl_only    = false,
    .pkt_access  = true,
    .ret_type    = RET_INTEGER,
    .arg1_type   = ARG_CONST_MAP_PTR,
    .arg2_type   = ARG_PTR_TO_MAP_KEY,
    .arg3_type   = ARG_PTR_TO_MAP_VALUE,
    .arg4_type   = ARG_ANYTHING,
};
```

Conversely, if an eBPF program does not invoke any GPL-only helper functions, no license declaration is strictly required. This distinction allows some eBPF programs to be loaded without enforcing GPL compatibility, depending on the set of helper functions they are using.

The set of GPL-compatible license identifiers accepted by the Linux kernel is defined in the kernel source code and is used by the eBPF verifier to validate program licensing [8].

```
/* SPDX-License-Identifier: GPL-2.0-only */
#ifndef __LICENSE_H
#define __LICENSE_H

static inline int license_is_gpl_compatible(const char *license)
{
    return (strcmp(license, "GPL") == 0
        || strcmp(license, "GPL v2") == 0
        || strcmp(license, "GPL and additional rights") == 0
        || strcmp(license, "Dual BSD/GPL") == 0
        || strcmp(license, "Dual MIT/GPL") == 0
        || strcmp(license, "Dual MPL/GPL") == 0);
}

#endif
```

Failing to satisfy these licensing constraints results in a rejection by the eBPF verifier at load time. License-related issues represent a of verification failures that could be detected earlier during development through static analysis at the C source code level.

What the Checker Does

License Extraction: The checker extracts the declared license from the program by analyzing the ‘LICENSE‘ variable:

- ‘VarDecl::getName()‘ → retrieves the variable name (searches for "LICENSE")
- ‘VarDecl::attrs()‘ → iterates over variable attributes
- ‘dyn_cast<SectionAttr>()‘ → checks if an attribute represents a section
- ‘SectionAttr::getName()‘ → verifies the section is named "license"
- ‘VarDecl::getInit()‘ → obtains the initialization expression
- ‘Expr::IgnoreImpCasts()‘ → removes implicit casts
- ‘dyn_cast<StringLiteral>()‘ → confirms the value is a string literal
- ‘StringLiteral::getString()‘ → extracts the license string (e.g., "GPL", "Dual MIT/GPL")

GPL-only Helper Function Detection: The checker identifies calls to GPL-only eBPF helper functions:

- ‘CallExpr::getDirectCallee()‘ → obtains the called function
- ‘CallExpr::getBeginLoc()‘ → retrieves the call location in the source
- ‘SourceLocation::isMacroID()‘ → checks if the call originates from a macro
- ‘SourceManager::getSpellingLoc()‘ → maps to the original source location
- ‘SourceManager::getCharacterData()‘ → accesses the source buffer
- ‘Lexer::getImmediateMacroName()‘ → extracts the macro name

Warning Reporting: The checker reports potential license violations:

- ‘ExplodedGraph::nodes()‘ → iterates over analysis graph nodes
- ‘ExplodedNode::getLocationContext()‘ → gets the node’s location context
- ‘LocationContext::getDecl()‘ → retrieves the function declaration
- ‘PathDiagnosticLocation::createBegin()‘ → creates a diagnostic location
- ‘BugReporter::emitReport()‘ → emits a warning or error

Checker Logic: The checker evaluates each program according to three possible cases:

1. **No GPL-only helper functions found:** the program is accepted; no warning is issued.
2. **GPL-only functions present, but license not declared:** a warning is generated:

   ```
   cannot call GPL-restricted function from non-GPL compatible
   program
   ```

3. **GPL-only functions present, license declared but not GPL-compatible:** a warning is generated:

   ```
   cannot call GPL-restricted function from non-GPL compatible
   program
   ```

4.3.1 Warning Explanation

The warning indicates that the program attempts to use a GPL function without having an appropriate license declaration.

```
warning: cannot call GPL-restricted function from non-GPL
compatible program [alpha.ebpf.EBPFVerifierChecker]
```

4.3.2 Algorithm & Data Structures

LicenseInfo

```
// Structure to store license information
struct LicenseInfo {
    bool foundDeclaredLicense = false;   // LICENSE declared in code
    std::string declaredLicense = "";    // license string
    bool isGplCompatible = false;        // GPL-compatible?
};
```

Main Algorithm

Initialize `LicenseInfo` structure:
  `foundDeclaredLicense = false`
  `declaredLicense = ""`
  `isGplCompatible = false`

for each variable declaration VD in the program **do**
  if VD name is `"LICENSE"` **then**
    for each attribute Attr in VD **do**
      if Attr is a `SectionAttr` and section name is `"license"` **then**
        `foundDeclaredLicense = true`
        if VD has an initializer expression and initializer is a `StringLiteral` **then**
          `declaredLicense = initializer string value`
          if `declaredLicense` is one of **then**
  {`"GPL"`, `"GPL v2"`,
  `"Dual BSD/GPL"`,
  `"Dual MIT/GPL"`,
  `"Dual MPL/GPL"`}
            `isGplCompatible = true`
          end if
        end if
      end if
    end for
  end if
end for
for each eBPF helper function call **do**
  if helper.`GplOnly` == true
  and (`foundDeclaredLicense == false`
  or `isGplCompatible == false`) **then**

22: Report warning:
"cannot call GPL-restricted function from non-GPL compatible program"|
23: **end if**
24: **end for**

4.4 Array Out Of Bounds Checker

Problem Description

This checker verifies accesses to global arrays (eBPF maps). It handles both compile-time known indices (e.g., `array[5]`, `array[x]`) and runtime symbolic indices (`array[i]`). It relies on the Clang Static Analyzer constraints to validate variable indices and detect out-of-bounds accesses.

Array and memory region identification:

- `dyn_cast<ArraySubscriptExpr>()` → identifies array access `array[index]`
- `CheckerContext::getSVal(ASE)` → obtains the symbolic value of the element
- `SVal::getAsRegion()` → converts SVal to a memory region (MemRegion)
- `dyn_cast<ElementRegion>()` → checks that it is an array element
- `ElementRegion::getSuperRegion()` → retrieves the full array from the element
- `dyn_cast<TypedValueRegion>()` → gets the typed region
- `dyn_cast<VarRegion>()` → gets the variable region
- `VarRegion::getDecl()` → gets the variable declaration
- `VarDecl::hasGlobalStorage()` → filters only global arrays (eBPF maps)

Runtime variable index case:

- `NonLoc::getAsSymbol()` → gets the symbolic index
- `CheckerContext::getConstraintManager()` → accesses the ConstraintManager

- `ConstraintManager::getSymMinVal(State, Symbol)` → gets minimum possible value
- `ConstraintManager::getSymMaxVal(State, Symbol)` → gets maximum possible value
- `APSInt::getSExtValue()` → converts min/max to `int64_t`

Clang CFG and ProgramState

Clang builds a Control Flow Graph (CFG) for each analyzed program. Each CFG node is associated with a `ProgramState`, which contains:

- **Store**: maps each variable (MemRegion) to its value (SVal), either concrete (e.g., 5) or symbolic (Symbol S)
- **Environment**: maps temporary expressions to their values
- **Constraints**: logical constraints on symbols, represented as ranges (e.g., S [0, 10])

When code branches (e.g., `if`, `while`, `for`), the `ConstraintManager`:

- Adds constraints to `ProgramState` for each branch
- Computes resulting ranges for each involved symbol
- Creates new `ProgramState` objects per path (true/false branches)

What the Checker Does

The checker detects out-of-bounds array accesses by analyzing both concrete and symbolic indices: **Array access detection (checkPostStmt()):**

- `dyn_cast<ArraySubscriptExpr>(Stmt)` → identifies `array[index]`
- Calls `checkArrayBounds(ASE, C)` to extract the MemRegion and array information
- Validates that the array is global (`hasGlobalStorage()`) and obtains the array size from `ConstantArrayType`

Concrete index analysis:

- `ElementRegion::getIndex()` → obtains the index
- Checks if concrete: `getAs<nonloc::ConcreteInt>()`
- Extracts value: `getValue().getExtValue()`
- Bounds check:
  - index < 0 → Warning: negative index
  - index $\geq$ array size → Warning: out-of-bounds

Symbolic index analysis:

- `getAsSymbol()` → gets the symbolic index
- Accesses `ConstraintManager` to get min/max constraints (`getSymMinVal()`, `getSymMaxVal()`)
- Case A: no constraints (MinVal or MaxVal is nullptr) → Warning: "unchecked symbolic index"
- Case B: constraints present
  - min < 0 → Warning: "unbounded min value"
  - max $\geq$ array size → Warning: out-of-bounds

For compile-time known indices or fully constrained symbolic indices, Clang can precisely detect out-of-bounds accesses. For runtime unknown indices, constraints are used to infer possible violations, although less accurately.

4.4.1 Warning Explanation

The checker emits three main types of warnings:

1. **Invalid Map Value Access**
   This warning occurs when si prova ad accedere fuori dalla dimensione dell array

   ```
   warning: invalid access to map value, value_size=16 off=24 size=4
   [alpha.ebpf.EBPFVerifierChecker]
   ```

2. **Invalid Map Value Access with Unbounded min value**
   This warning occurs when si prova ad accedere fuori dalla dimensione dell array e il limite minore dell'array non è verificato.

   ```
   warning: math between map_value pointer and register with unbounded min
   value is not allowed [alpha.ebpf.EBPFVerifierChecker]
   ```

3. **Unchecked Symbolic Index on Global Array**
   This warning indicates that a global array (e.g., an eBPF map) is accessed using a symbolic index whose bounds have not been verified, potentially causing out-of-bounds access.

```
warning: global array 'array name' accessed with unchecked
symbolic index
```

4.4.2 Algorithm & Data Structures

Input: Array A with size n (known or symbolic), index i (constant or symbolic)
Output: Error report if the index is out of bounds
if i is constant **then**
    if $i < 0$ **then**
        Report error: "Negative index"
        return
    end if
    if size n is known and $i \geq n$ **then**
        Report error: "Index out of array bounds"
        return
    end if
    if size n is symbolic and $i \geq$ symbolic maximum value of n **then**
        Report error: "Index may exceed array bounds"
        return
    end if
else if i is symbolic **then**
    if symbolic minimum index < 0 **then**
        Report error: "Index may be negative"
        return
    end if
    if size n is known and symbolic maximum index $\geq n$ **then**
        Report error: "Index may exceed array bounds"
        return
    end if
    if size n is symbolic and symbolic maximum index $\geq$ symbolic minimum array size **then**
        Report error: "Index may exceed variable array bounds"
        return
    end if
end if
return

4.5 Helper Function Type Checker

Problem Description

eBPF programs interact with the kernel through a set of predefined helper functions. Each helper function is associated with:

a specific set of eBPF program types in which it is allowed to be used, and

a well-defined function signature specifying the expected types of its arguments.

Violations of these constraints—such as invoking a helper function from an incompatible program type or passing arguments with incorrect types—cause the eBPF verifier to reject the program at load time.

These errors are often non-trivial to detect during development, especially when argument types depend on map definitions, pointer semantics, or implicit casts in C code. Since the eBPF verifier operates on bytecode rather than on the original C source, error messages may be difficult to trace back to the exact source-level cause.

For this reason, a static analysis performed at the C source code level can identify helper function misuse earlier in the development cycle, improving programmability and reducing debugging effort

The specification of eBPF helper functions, including their allowed program types and argument signatures, is derived from two main sources: the Linux kernel helper definitions and the helper type metadata used by the BRF verifier [9, 10].

These sources provide, for each helper function, information about its availability, GPL restrictions, supported eBPF program types, and expected argument types.

What the Checker Does

The Helper Function Type Checker statically analyzes eBPF programs written in C to validate the correctness of helper function invocations with respect to both **program type compatibility** and **argument type correctness**.

The checker operates in the following stages.

Step 1: Program Section Identification The checker analyzes function attributes to identify the program section specified through the `SEC("section_name")` macro. From the extracted section name (e.g., `tracepoint/syscalls/sys_enter_execve`), the corresponding eBPF program type is derived and stored as the *current program type*. This information is later used to determine whether specific helper functions are allowed in the given execution context.

Step 2: Map Declaration Extraction Global variables annotated with `SEC(".maps")` are identified as eBPF map declarations. For each detected map, the checker extracts structural information, including:

- map name,
- map type,
- key size,
- value size,
- maximum number of entries.

This information is stored using an internal representation (e.g., `EBPFMapInfo`) and is subsequently used during helper argument validation, particularly for helpers that operate on map pointers, keys, or values.

Step 3: Helper Function Identification and Program Type Validation The checker scans the program to identify calls to known eBPF helper functions. The list of supported helpers, together with their allowed program types and expected argument signatures, is derived from authoritative kernel specifications and helper definitions.

For each helper function call, the checker:

- verifies whether the helper function is permitted for the current eBPF program type, and
- reports an error if the helper is not valid in the detected context.

Only if the helper function is compatible with the current program type does the checker proceed to argument validation.

Step 4: Argument Type Validation For each argument passed to a helper function, the checker:

- extracts the effective type of the argument as specified in the C source code,
- retrieves the expected argument type from the helper function signature, and
- compares the actual and expected types.

If the types are compatible, the argument is considered valid. Otherwise, the checker reports a type mismatch error, indicating an invalid helper function invocation.

By performing these checks at the C source code level, the checker anticipates potential verifier rejections related to helper misuse and provides precise, source-level diagnostics before eBPF bytecode generation.

4.5.1 Warning Explanation

The checker emits 2 main types of warnings:

1. **Helper Function Not allowed in the context of the program**
   This warning occurs when a helper function is not allowed in the context of the program.

   ```
   warning: unknown func 'helper function name', not allowed
   in section 'sec name' [alpha.ebpf.EBPFVerifierChecker]
   ```

2. **Helper Function Parameter Mismatch**
   This warning occurs when a helper function is allowed in the program context, but the parameters passed do not match the expected types defined in the helper's signature.

   ```
   warning: call 'helper function name' - parameter '<expression>' does
   not match expected type 'expected type name' (actual: 'parameter used')
   [alpha.ebpf.EBPFVerifierChecker]
   ```

4.5.2 Algorithm & Data Structures

The Helper Function Type Checker relies on a set of custom data structures used to model eBPF helper functions, program types, and map metadata. These structures are derived from [9, 10].

BpfProgTypeEnum

```
// BPF Program Type Enum
enum BpfProgTypeEnum {
    BPF_PROG_TYPE_UNSPEC = 0,
    BPF_PROG_TYPE_SOCKET_FILTER = 1,
    BPF_PROG_TYPE_KPROBE = 2,
    BPF_PROG_TYPE_SCHED_CLS = 3,
    BPF_PROG_TYPE_SCHED_ACT = 4,
    BPF_PROG_TYPE_TRACEPOINT = 5,
    BPF_PROG_TYPE_XDP = 6,
    BPF_PROG_TYPE_PERF_EVENT = 7,
    BPF_PROG_TYPE_CGROUP_SKB = 8,
    BPF_PROG_TYPE_CGROUP_SOCK = 9,
    BPF_PROG_TYPE_LWT_IN = 10,
    BPF_PROG_TYPE_LWT_OUT = 11,
    BPF_PROG_TYPE_LWT_XMIT = 12,
    BPF_PROG_TYPE_SOCK_OPS = 13,
    BPF_PROG_TYPE_SK_SKB = 14,
    BPF_PROG_TYPE_CGROUP_DEVICE = 15,
    BPF_PROG_TYPE_SK_MSG = 16,
    BPF_PROG_TYPE_RAW_TRACEPOINT = 17,
    BPF_PROG_TYPE_CGROUP_SOCK_ADDR = 18,
    BPF_PROG_TYPE_LWT_SEG6LOCAL = 19,
    BPF_PROG_TYPE_LIRC_MODE2 = 20,
    BPF_PROG_TYPE_SK_REUSEPORT = 21,
    BPF_PROG_TYPE_FLOW_DISSECTOR = 22,
    BPF_PROG_TYPE_CGROUP_SYSCTL = 23,
    BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE = 24,
    BPF_PROG_TYPE_CGROUP_SOCKOPT = 25,
    BPF_PROG_TYPE_TRACING = 26,
    BPF_PROG_TYPE_STRUCT_OPS = 27,
    BPF_PROG_TYPE_EXT = 28,
    BPF_PROG_TYPE_LSM = 29,
    BPF_PROG_TYPE_SK_LOOKUP = 30,
    BPF_PROG_TYPE_SYSCALL = 31,
    BPF_PROG_TYPE_NETFILTER = 32,
    BPF_PROG_TYPE_MAX = 33
};
```

BpfHelperEnum

```
// eBPF Helper Functions Enum (partial)
enum BpfHelperEnum {
    BPF_FUNC_unspec = 0,
    BPF_FUNC_map_lookup_elem = 1,
    BPF_FUNC_map_update_elem = 2,
    BPF_FUNC_map_delete_elem = 3,
    BPF_FUNC_probe_read = 4,
    BPF_FUNC_ktime_get_ns = 5,
    BPF_FUNC_trace_printk = 6,
    BPF_FUNC_get_prandom_u32 = 7,
    BPF_FUNC_tail_call = 12,
    BPF_FUNC_probe_read_user = 112,
    BPF_FUNC_probe_read_kernel = 113,
    BPF_FUNC_probe_read_user_str = 114,
    BPF_FUNC_probe_read_kernel_str = 115
};
```

BpfHelper

```
// Struct BpfHelper
struct BpfHelper {
    std::string Uname;
    BpfHelperEnum HelperId;
    std::string ImplFunc;
    std::string ProtoFunc;
    std::vector<std::string> Args;
    std::string Ret;
    bool PktAccess = false;
    bool GplOnly = false;

    BpfHelper(std::string uname, BpfHelperEnum helperId,
              std::string implFunc, std::string protoFunc,
              std::vector<std::string> args, std::string ret,
              bool pktAccess = false, bool gplOnly = false)
        : Uname(std::move(uname)), HelperId(helperId),
          ImplFunc(std::move(implFunc)), ProtoFunc(std::move(
    protoFunc)),
          Args(std::move(args)), Ret(std::move(ret)),
          PktAccess(pktAccess), GplOnly(gplOnly) {}
};
```

BpfSecDef

```
// Struct BpfSecDef
```

```
struct BpfSecDef {
    std::string Sname;
    bool enabled;
};
```

BpfProgType

```
// Struct BpfProgType
struct BpfProgType {
    std::string name;
    std::string user;
    std::string kern;
    BpfProgTypeEnum progTypeId;
    std::vector<BpfSecDef> secDefs;
    std::vector<std::string> funcProtos;

    BpfProgType(std::string name,
                std::string user,
                std::string kern,
                BpfProgTypeEnum progTypeId,
                std::vector<BpfSecDef> secDefs,
                std::vector<std::string> funcProtos)
        : name(std::move(name)), user(std::move(user)),
          kern(std::move(kern)), progTypeId(progTypeId),
          secDefs(std::move(secDefs)), funcProtos(std::move(
    funcProtos)) {}
};
```

EBPFMapInfo

```
// Struct EBPFMapInfo
struct EBPFMapInfo {
    std::string name = "not_known";
    std::string type = "not_known";
    std::string key_size = "not_known";
    std::string value_size = "not_known";
    int max_entries = 0;

    EBPFMapInfo() = default;

    EBPFMapInfo(std::string name,
                std::string mapType,
                std::string keyType,
                std::string valueType,
                int maxEntries = 0)
        : name(std::move(name)), type(std::move(mapType)),
```

```
            key_size(std::move(keyType)), value_size(std::move(
    valueType)),
            max_entries(maxEntries) {}
};
```

ProgTypesMap

```
// Example: ProgTypesMap
inline const std::map<BpfProgTypeEnum, BpfProgType> ProgTypesMap = {
    {BPF_PROG_TYPE_SOCK_OPS, BpfProgType{
        "sock_ops", "struct bpf_sock_ops", "struct bpf_sock_ops_kern"
    ,
        BPF_PROG_TYPE_SOCK_OPS,
        {{"sockops", false}},
        {"bpf_sock_ops_setsockopt", "bpf_sock_ops_getsockopt",
         "bpf_sock_ops_cb_flags_set", "bpf_sock_map_update"}
    }},
    {BPF_PROG_TYPE_LIRC_MODE2, BpfProgType{
        "lirc_mode2", "__u32", "u32", BPF_PROG_TYPE_LIRC_MODE2,
        {{"lirc_mode2", false}},
        {"rc_repeat", "rc_keydown", "rc_pointer_rel", "
    bpf_map_lookup_elem"}
    }}
};
```

HelperFuncsMap

```
// Example: HelperFuncsMap
inline const std::map<std::string, BpfHelper> HelperFuncsMap = {
    {"bpf_map_lookup_elem", BpfHelper{
        "bpf_map_lookup_elem", BPF_FUNC_map_lookup_elem,
        "bpf_map_lookup_elem", "bpf_map_lookup_elem",
        {"ARG_CONST_MAP_PTR", "ARG_PTR_TO_MAP_KEY"},
        "RET_PTR_TO_MAP_VALUE_OR_NULL", false
    }},
    {"bpf_map_update_elem", BpfHelper{
        "bpf_map_update_elem", BPF_FUNC_map_update_elem,
        "bpf_map_update_elem", "bpf_map_update_elem",
        {"ARG_CONST_MAP_PTR", "ARG_PTR_TO_MAP_KEY", "
    ARG_PTR_TO_MAP_VALUE", "ARG_ANYTHING"},
        "RET_INTEGER", false
    }}
};
```

Main Algorithm

```
Input:
  eBPF C source code
  Helper function definitions (HelperFuncsMap)
  Program type definitions (ProgTypesMap)
Output:
  Diagnostic reports for invalid helper usage or argument type mismatches

Step 1: Program Section Identification
for each function declaration F do
   if F has attribute SEC("section_name") then
      if E thenxtract section_name
         if D thenetermine current eBPF program type
         end if


         Step 2:  Map Declaration Extraction
         for each global variable declaration V do
            if V has attribute SEC(".maps") then
               Extract map metadata
               Store data in an EBPFMapInfo structure
            end if
         end for

         Step 3:  Helper Function Identification
         for each function call expression C do
            if C matches a helper in HelperFuncsMap then
               Retrieve corresponding BpfHelper
               if helper not allowed for current program type
then
                  Report error:  invalid helper for program
type
                  continue
               end if
               Proceed to argument validation
            end if
         end for

         Step 4:  Argument Type Validation
         for each argument a_i do
            Extract actual C type
            Retrieve expected helper argument type
            if types are incompatible then
               Report error:  helper argument type mismatch
            end if
         end for
         return Helper function validation complete
```

4.6 Packet Checker

Problem Description

In XDP programs, packet data is accessed through pointers such as `data` and `data_end`. Improper handling of these pointers can lead to invalid memory accesses, including reading or writing outside the bounds of the packet. Such errors may cause program crashes, undefined behavior, or security vulnerabilities. The problem is exacerbated when pointer arithmetic involves symbolic or dynamic offsets that are not known at compile time. The Packet Checker addresses this class of errors by statically analyzing the program to detect unsafe memory accesses.

What the Checker Does

The Packet Checker ensures safe packet memory access by performing the following tasks:

- Detecting packet memory accesses that are not protected by explicit bounds checks.
- Identifying invalid arithmetic operations on `data_end`.
- Handling offsets that are either concrete (e.g., `data + 15`, `data + i` where `i` is known at compile time) or symbolic (e.g., `data + var_dyn` where `var_dyn` is unknown at compile time).
- Analyzing all possible execution paths in the Control Flow Graph (CFG).

Key Clang components used in the analysis:

- **Control Flow Graph (CFG)**: a graphical representation of program execution, with nodes as statements and edges as transitions.
- **Exploded Graph**: an expansion of the CFG including symbolic program states, enabling path-sensitive analysis.
- **Program State**: snapshots of variable values, symbolic values, and constraints at a given program point.

- **Constraint Manager**: tracks constraints on symbolic variables, supporting operations such as `getSymbolRange()` to determine the possible value range of variables unknown at compile time.

Main analysis functions:

- `checkBind`: captures pointer assignments, updating `PacketPtrMap` entries.
- `checkPostStmt`: intercepts arithmetic operations on packet pointers.
- `analyzeIfInPath`: examines bounds-checking conditions in `if` statements to extract valid ranges.
- `analyzeAccessInPath`: validates each memory access against the pointer's allowed range.
- `collectPaths`: reconstructs all possible execution paths through the CFG.

Pointer range propagation occurs only between pointers derived from the same concrete base without unknown symbolic components, ensuring memory accesses remain safe.

4.6.1 Warning Explanation

The checker emits three main types of warnings to detect violations in eBPF programs related to safe packet access.

1. **Arithmetic on `data_end`**
   Unsafe pointer arithmetic performed on the end-of-packet pointer.

   ```
   warning: 'data_end' ('name of the end-of-packet pointer')
   pointer arithmetic on pkt_end prohibited
   [alpha.ebpf.EBPFPacketChecker]
   ```

2. **Dereference of `data_end`**
   Attempt to directly dereference the end-of-packet pointer.

```
warning: 'data_end' ('name of the end-of-packet pointer')
invalid mem access 'pkt_end'
[alpha.ebpf.EBPFPacketChecker]
```

3. **Out-of-Bounds Access**
   Access to packet data that exceeds the validated bounds.

```
warning: Invalid packet access (offset=X, bound=Y), ptr 'variable_name'
[alpha.ebpf.EBPFPacketChecker]
```

4.6.2 Algorithm & Data Structures

PacketPointerInfo Structure tracking packet pointer information:

```
struct PacketPointerInfo {
  std::string varName;            // Variable name
  std::string regionStr;          // Memory region identifier
  bool isPtrToEnd;                // true if points to data_end
  int offsetPkt;                  // Verified bound (accessible bytes)
  int basePkt;                    // Base offset from root (bytes)
  bool hasSymbolicBase;           // true if base contains symbolic
    components
};
```

BoundSnapshot Snapshot for temporary scope management:

```
struct BoundSnapshot {
  const IfStmt *ifStmt;
  std::map<const MemRegion*, PacketPointerInfo> savedPtrMap;
};
```

Program State Maps

- `PacketPtrMap`: map `MemRegion` → `PacketPointerInfo` in program state

- `localPtrMap`: local map for bound tracking along execution path
- `boundStack`: stack for restoring bounds after temporary branches

Main Algorithm

Algorithm 1: Packet Checker - Complete Flow

```
// PHASE 1: Path Collection
allPaths ← []
for each endpoint e ∈ G do
    paths ← CollectPathsRecursive(e, [], allPaths)
end for

// PHASE 2: Per-Path Analysis
for each path ∈ allPaths do
    localPtrMap ← ∅                          ▷ Tracks bounds per pointer
    boundStack ← []                          ▷ Stack for temporary scopes
    for N ∈ path in execution order do
        S ← statement at node N
        St ← program state at node N
        if S = ∅ then
            continue
        end if
        // STEP 2.1: Update Pointer Base Information
        for (region, info) ∈ St.PacketPtrMap do
            if region ∉ localPtrMap then
                localPtrMap[region] ← info
                localPtrMap[region].offsetPkt ← 0      ▷ Bound managed by checker
            else
                localPtrMap[region].basePkt ← info.basePkt
                localPtrMap[region].hasSymbolicBase ← info.hasSymbolicBase
                localPtrMap[region].isPtrToEnd ← info.isPtrToEnd
            end if
        end for

        // STEP 2.2: Manage Bound Stack (restore after scope)
        while boundStack ≠ [] do
            topIfStmt ← boundStack.top().ifStmt
```

```
32:             stillInside ← (S inside topIfStmt.then or S inside topIfStmt.else)
33:             if not stillInside and S ≠ topIfStmt then
34:                 localPtrMap ← boundStack.pop().savedPtrMap        ▷ Restore bounds
35:             else
36:                 break
37:             end if
38:         end while
39:
40:         // STEP 2.3: Analyze Bounds-Checking Conditions
41:         if S is IfStmt then
42:             BO ← S.condition (binary operator)
43:             if BO.op ∈ {<, ≤, >, ≥} then
44:                 lhsVars ← ExtractVariables(BO.LHS)
45:                 rhsVars ← ExtractVariables(BO.RHS)
46:                 // Check pattern: LHS = non-end pointers, RHS = data_end
47:                     validPattern ← (all v ∈ lhsVars are non-end) and (all v ∈ rhsVars are end)
48:                     if validPattern then
49:                         // Determine which branch was taken
50:                         tookThen ← DetermineBranch(path, S, N)
51:                         hasReturnThen ← HasReturnStmt(S.then)
52:                         hasReturnElse ← HasReturnStmt(S.else)
53:                         // Classify guard pattern
54:                         isGuard ← false
55:                         shouldUpdate ← false
56:                         needsPush ← false
57:                         if BO.op ∈ {≥, >} then
58:                             if hasReturnThen and not tookThen then
59:                                 isGuard ← true, shouldUpdate ← true
60:                             else if hasReturnElse and tookThen then
61:                                 continue                    ▷ Unsafe branch, skip
62:                             else if hasReturnElse and not tookThen then
63:                                 shouldUpdate ← true, needsPush ← true
64:                             else if not tookThen and S.else ≠ ∅ then
65:                                 shouldUpdate ← true
66:                             end if
67:                         else if BO.op ∈ {<, ≤} then
68:                             if hasReturnThen and tookThen then
69:                                 shouldUpdate ← true, needsPush ← true
70:                             else if hasReturnThen and not tookThen then
```

```
71:                         continue                    ▷ Unsafe branch, skip
72:                     else if hasReturnElse and tookThen then
73:                         shouldUpdate ← true
74:                     else if tookThen then
75:                         shouldUpdate ← true
76:                     end if
77:                 end if
78:                 if shouldUpdate then
79:                     // Compute offset from LHS expression
80:                     hasVoidCast ← DetectVoidCast(BO.LHS)
81:                     totalOffset ← 0
82:                     region ← St.getSVal(BO.LHS).getAsRegion()
83:                     while region is ElementRegion do
84:                         index ← region.index
85:                         offset ← EvaluateOffset(index, St)
86:                         elemSize ← (hasVoidCast ? 1 : region.elementType.size)
87:                         totalOffset ← totalOffset + offset × elemSize
88:                         region ← region.superRegion
89:                     end while
90:                     baseRegion ← ExtractBaseRegion(BO.LHS)
91:                     if baseRegion ∈ localPtrMap then
92:                         totalOffset ← totalOffset + localPtrMap[baseRegion].basePkt
93:                     end if
94:                     targetRegion ← ExtractTargetRegion(BO.LHS)
95:                     localPtrMap[targetRegion].offsetPkt ← max(current, totalOffset)
96:                     if isGuard then
97:                         // Propagate bound to all non-symbolic pointers
98:                         for r ∈ localPtrMap where not localPtrMap[r].hasSymbolicBase do
99:                             localPtrMap[r].offsetPkt ← max(localPtrMap[r].offsetPkt, totalOffset)
100:                        end for
101:                    end if
102:                    if needsPush then
103:                        boundStack.push({S, localPtrMap})   ▷ Save for later restore
104:                    end if
```

```
105:                     end if
106:                 end if
107:             end if
108:         end if
109:
110:         // STEP 2.4: Verify Memory Accesses
111:         if S is UnaryOperator with opcode = UO_Deref then
112:             ptrExpr ← S.subExpr
113:             baseVar ← ExtractBaseVariable(ptrExpr)
114:             baseRegion ← GetVarRegion(baseVar, St)
115:             if baseRegion ∉ localPtrMap then
116:                 continue                    ▷ Not a tracked packet pointer
117:             end if
118:             ptrInfo ← localPtrMap[baseRegion]
119:             if ptrInfo.isPtrToEnd then
120:                 ReportBug("Dereference of data_end", S)
121:                 continue
122:             end if
123:             // Compute access offset
124:             hasVoidCast ← DetectVoidCast(ptrExpr)
125:             totalOffset ← 0
126:             offsetCalculated ← false
127:             if not hasVoidCast then
128:                 // METHOD 1: MemRegion traversal
129:                 derefRegion ← St.getSVal(S).getAsRegion()
130:                 if derefRegion is ElementRegion then
131:                     currentRegion ← derefRegion
132:                     while currentRegion is ElementRegion do
133:                         index ← currentRegion.index
134:                         offset ← EvaluateOffset(index, St)
135:                         elemSize ← currentRegion.elementType.size
136:                         totalOffset ← totalOffset + offset × elemSize
137:                         currentRegion ← currentRegion.superRegion
138:                     end while
139:                     if currentRegion is VarRegion with symbolic base then
140:                         totalOffset ← totalOffset + localPtrMap[currentRegion].basePkt
141:                     end if
142:                     offsetCalculated ← true
143:                 end if
144:             end if
```

```
145:                if not offsetCalculated then
146:                    // METHOD 2: Expression-based evaluation
147:                    expr ← ptrExpr
148:                    while expr is BinaryOperator do
149:                        if expr.op = Add then
150:                            rhsVal ← St.getSVal(expr.RHS)
151:                            offset ← EvaluateSymbolic(rhsVal, St)
152:                            totalOffset ← totalOffset + offset
153:                        else if expr.op = Sub then
154:                            rhsVal ← St.getSVal(expr.RHS)
155:                            offset ← EvaluateSymbolic(rhsVal, St)
156:                            totalOffset ← totalOffset − offset
157:                        end if
158:                        expr ← expr.LHS
159:                    end while
160:                    totalOffset ← totalOffset + ptrInfo.basePkt
161:                end if
162:                // Compare against bound
163:                if totalOffset ≥ ptrInfo.offsetPkt then
164:                    msg ← "Out of bounds: offset=totalOffset,
    bound=ptrInfo.offsetPkt"
165:                    ReportBug(msg, S)
166:                end if
167:            end if
168:        end for
169:    end for
170:
171:    // HELPER: Evaluate Offset (concrete or symbolic)
172:    function EvaluateOffset(index, St)
173:        if index is ConcreteInt then
174:            return index.value
175:        else if index is Symbol then
176:            return EvaluateSymbolic(index, St)
177:        end if
178:        return 0
179:    end function
180:
181:    function EvaluateSymbolic(sym, St)
182:        if sym is SymIntExpr then                    ▷ symbol OP constant
183:            lhs ← EvaluateSymbolic(sym.LHS, St)
184:            rhs ← sym.RHS.value
```

```
185:            return ApplyOp(sym.opcode, lhs, rhs)
186:        else if sym is IntSymExpr then                ▷ constant OP symbol
187:            lhs ← sym.LHS.value
188:            rhs ← EvaluateSymbolic(sym.RHS, St)
189:            return ApplyOp(sym.opcode, lhs, rhs)
190:        else if sym is SymSymExpr then                ▷ symbol OP symbol
191:            lhs ← EvaluateSymbolic(sym.LHS, St)
192:            rhs ← EvaluateSymbolic(sym.RHS, St)
193:            return ApplyOp(sym.opcode, lhs, rhs)
194:        else                                          ▷ Base symbol: extract range
195:            (min, max) ← GetSymbolRange(sym, St)
196:            return max                                ▷ Worst-case analysis
197:        end if
198:    end function=0
```

Range Propagation Policy Bound propagation occurs only between pointers sharing the same concrete base. If `hasSymbolicBase = true`, the pointer is isolated and does not inherit or propagate bounds. Guard patterns (`if(ptr >= end) return;`) propagate bounds globally to all non-symbolic pointers along the execution path.

Chapter 5

Static Analyzer Checker Tests

5.1 Clang Version

The version of the Clang compiler used to build and execute the static analyzer checkers was based on LLVM/Clang [11].
The commit used for compilation is

`df570dadcb93a32c308d31c2ab54d2f46c8ae0c0.`

```
clang version 21.0.0git
Target: unknown
Thread model: posix
InstalledDir: /usr/local/bin
```

5.2 Linux Version

The operating system and kernel on which the tests were executed correspond to Linux kernel version 6.8 [12], running on Ubuntu 6.8.0-87-generic.

```
Linux 6.8.0-87-generic #88-Ubuntu SMP PREEMPT_DYNAMIC
Sat Oct 11 09:28:41
UTC 2025 x86_64 x86_64 x86_64 GNU/Linux
```

5.3 Test Structure

Each analyzed test case consists of intentionally incorrect C code designed to be rejected by the eBPF verifier. The goal of these tests is to determine whether the

developed static analyzer checkers can detect the errors and generate warnings that are consistent with those reported by the verifier.

5.4 Test Suites

The checkers developed in this work were evaluated using two main sources of test cases:

1. **Internal test cases:** A set of programs maintained in private institutional repositories of the Politecnico di Torino. These include standard test programs as well as deliberately crafted cases designed to exercise corner cases of the verifier.

2. **Fuzzer-generated programs:** Additional test cases were produced using a semantic fuzzer based on the BRT framework, which represents the state-of-the-art in semantic fuzzers for eBPF. The fuzzer was inverted to generate programs intentionally rejected by the verifier, enabling the evaluation of the *pretty-verifier* tool, which aims to improve feedback provided by the verifier.

5.5 Test Categories

5.5.1 License

These tests focus on verifying the correct enforcement of eBPF helper function licensing rules. Programs using GPL-only helper functions without a proper license declaration should be rejected by the verifier, while programs using non-GPL helpers should be accepted.

gpl_only_true.bpf.c

```c
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __uint(key_size, sizeof(__u32));
    __uint(value_size, sizeof(__u32));
} events SEC(".maps");

SEC("tracepoint/syscalls/sys_enter_openat")
int test_gpl_required(void *ctx) {
    __u64 data = 42;

    // bpf_perf_event_output has gpl_only = true
    bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, &data,
    sizeof(data));

    return 0;
}
```

Verifier Rejection

```
cannot call GPL-restricted function from non-GPL compatible program
libbpf: prog 'test_gpl_required': failed to load: -22
libbpf: failed to load object 'gpl_only_true.bpf.o'
```

Checker Behaviour

```
warning: cannot call GPL-restricted function from non-GPL compatible program
[alpha.ebpf.EBPFVerifierChecker]
  14 | SEC("tracepoint/syscalls/sys_enter_openat")
```

gpl_only_false.bpf.c

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

//No license declared
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 1);
```

```
    __type(key, __u32);
    __type(value, __u64);
} cgroup_map SEC(".maps");

SEC("tracepoint/syscalls/sys_enter_openat")
int gpl_only_false(void *ctx) {
    __u64 cgroup_id = bpf_get_current_cgroup_id();
    __u32 key = 0;
    bpf_map_update_elem(&cgroup_map, &key, &cgroup_id, BPF_ANY);

    return 0;
}
```

Verifier

```
The verifier accepts the code
```

Checker Behaviour No warning generated from the checker

```
oscar@oscar:~/test-supervisor-clean/llvm-project/build$ /home/oscar/scripts/run
```

5.5.2 Array Out Of Bounds

These tests examine unsafe accesses to global arrays, where the index may be derived at runtime or exceed array bounds.The eBPF verifier must ensure memory safety, rejecting programs that attempt out-of-bounds access, and the checker should generate corresponding warnings

test_01

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

int arr[4] = {10, 20, 30, 40};

SEC("xdp")
int test_01(struct xdp_md *ctx)
{
    int index = ctx->ingress_ifindex;
    int val = arr[index];
    bpf_printk("val: %d\n", val);

    return XDP_PASS;
```

```
}

char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

Verifier Rejection

```
math between map_value pointer and register with unbounded min value
is not allowed
libbpf: prog 'test_01': failed to load: -22
libbpf: failed to load object 'test01.bpf.o'
```

Checker Behaviour

```
 warning: math between map_value pointer and register with unbounded
 min value is not allowed [alpha.ebpf.EBPFVerifierChecker]
   10 |     int val = arr[index];
      |               ^~~~~~~~~~
1 warning generated.
```

test_02

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

int arr[4] = {10, 20, 30, 40};

SEC("xdp")
int test_02(struct xdp_md *ctx)
{
    int index = ctx->ingress_ifindex;
    int val= 0;
    if(index < 0 || index > 6)
    {
        return XDP_PASS;
    }
    val = arr[index];
    bpf_printk("val:  %d\n", val);

    return XDP_PASS;

}

char LICENSE[] SEC("license") = "Dual BSD/GPL"
```

Verifier Rejection

```
invalid access to map value, value_size=16 off=24 size=4
R2 max value is outside of the allowed memory range
libbpf: prog 'test_02': failed to load: -13
libbpf: failed to load object 'test02.bpf.o'
```

Checker Behaviour

```
test02.bpf.c:15:11: warning: invalid access to map value,
value_size=16 off=24 size=4 [alpha.ebpf.EBPFVerifierChecker]
   15 |     val = arr[index];
      |           ^~~~~~~~~~
1 warning generated.
```

test_03

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

int arr[4] = {10, 20, 30, 40};

SEC("xdp")
int test_03(struct xdp_md *ctx)
{
    int index = ctx->ingress_ifindex;
    int val= 0;
    if(index < 10)
    {
        val = arr[index];
    }
    bpf_printk("val:  %d\n", val);

    return XDP_PASS;

}

char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

Verifier Rejection

```
math between map_value pointer and register with unbounded min value
is not allowed
```

```
libbpf: prog 'test_03': failed to load: -22
libbpf: failed to load object 'test03.bpf.o'
```

Checker Behaviour

```
/home/oscar/scripts/outOfBounds/test03.bpf.c:16:15: warning: invalid
access to map value, value_size=16 off=36 size=4 [alpha.ebpf.EBPFVerifierChecke
   16 |         val = arr[index];
      |               ^~~~~~~~~~
1 warning generated.
```

The discrepancy between the verifier error and the checker warning arises from their different constraint models: the Constraint Manager infers a symbolic range [0, 9], whereas the eBPF verifier only validates the upper bound (index 9), leaving the lower bound unbounded.

test_05

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

int arr[4] = {10, 20, 30, 40};

SEC("xdp")
int test_05(struct xdp_md *ctx)
{
    int index = 10;
    int val = arr[index];
    bpf_printk("val: %d\n", val);

    return XDP_PASS;

}

char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

Verifier Rejection

```
invalid access to map value, value_size=16 off=40 size=4
R1 min value is outside of the allowed memory range
libbpf: prog 'test_05': failed to load: -13
libbpf: failed to load object 'test05.bpf.o'
```

Checker Behaviour

```
warning: invalid access to map value, value_size=16 off=40 size=4
[alpha.ebpf.EBPFVerifierChecker]
   10 |     int val = arr[index];
      |               ^~~~~~~~~~
1 warning generated.
```

test_08

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

int arr[4] = {10, 20, 30, 40};

SEC("xdp")
int test_08(struct xdp_md *ctx)
{
    int index = -2;
    int val= 0;
    val = arr[index];
    bpf_printk("val:  %d\n", val);

    return XDP_PASS;

}

char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

Verifier Rejection

```
R1 min value is outside of the allowed memory range
libbpf: prog 'test_08': failed to load: -13
libbpf: failed to load object 'test08.bpf.o'
```

Checker Behaviour

```
warning: invalid access to map value, value_size=16 off=-8 size=4
[alpha.ebpf.EBPFVerifierChecker]
   12 |     val = arr[index];
      |           ^~~~~~~~~~
1 warning generated.
```

5.5.3 Helper Functions

These tests illustrate the use of helper functions that are either restricted to certain program types or disallowed in specific contexts. The verifier rejects improper uses, and the static checker should identify and warn about these violations

test_02

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

SEC("xdp")
int xdp_wrong_helper(struct xdp_md *ctx)
{
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    bpf_printk("PID: %d\n", pid);

    return XDP_PASS;
}

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

Verifier Rejection

```
unknown func bpf_get_current_pid_tgid#14
libbpf: prog 'xdp_wrong_helper': failed to load: -22
libbpf: failed to load object 'test02.bpf.o'
```

Checker Behaviour

```
./test02.bpf.c:9:15: warning: unknown func 'bpf_get_current_pid_tgid'
, not allowed in section 'xdp' [alpha.ebpf.EBPFVerifierChecker]
    9 |     u32 pid = bpf_get_current_pid_tgid() >> 32;
      |               ^~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
All done.
```

test_06

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
```

```c
struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __uint(max_entries, 10);
    __type(key, u32);
    __type(value, u64);
} null_test SEC(".maps");

SEC("kprobe/sys_write")
int null_pointer_test(struct pt_regs *ctx)
{
    u32 key = 0;
    u64 *val = bpf_map_lookup_elem(&null_test, NULL);
    return 0;
}

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

Verifier Rejection

```
R2 type=scalar expected=fp, pkt, pkt_meta, map_key, map_value, mem,
ringbuf_mem, buf, trusted_ptr_
libbpf: prog 'null_pointer_test': failed to load: -13
libbpf: failed to load object 'test06.bpf.o'
```

Checker Behaviour

```
./test06.bpf.c:18:16: warning: call bpf_map_lookup_elem - parameter
'<expression>' does not match expected type ARG_PTR_TO_MAP_KEY
(actual: const void *) [alpha.ebpf.EBPFVerifierChecker]
   18 |     u64 *val = bpf_map_lookup_elem(&null_test, NULL);
      |                ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

test_12

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

SEC("xdp")
int wrong_section_type(struct xdp_md *ctx)
{
    // bpf_msg_apply_bytes is only for SK_MSG programs
    bpf_msg_apply_bytes(ctx, 100);
    return XDP_PASS;
}

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

Verifier Rejection

```
unknown func bpf_msg_apply_bytes#61
libbpf: prog 'wrong_section_type': failed to load: -22
libbpf: failed to load object 'test12.bpf.o'
```

Checker Behaviour

```
./test12.bpf.c:10:5: warning: unknown func 'bpf_msg_apply_bytes',
not allowed in section 'xdp' [alpha.ebpf.EBPFVerifierChecker]
   10 |     bpf_msg_apply_bytes(ctx, 100);
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

test_18

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 64 * 1024);
} ringbuf SEC(".maps");

SEC("kprobe/sys_getpid")
int alloc_mem_test(struct pt_regs *ctx)
{
```

```
    void *data = bpf_ringbuf_reserve(&ringbuf, 64, 0);
    if (!data)
        return 0;

    u64 stack_var = 42;
    bpf_ringbuf_submit(&stack_var, 0);

    return 0;
}

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

Verifier Rejection

```
R1 type=fp expected=ringbuf_mem
libbpf: prog 'alloc_mem_test': failed to load: -13
libbpf: failed to load object 'test18.bpf.o'
```

Checker Behaviour

```
./test18.bpf.c:20:5: warning: call bpf_ringbuf_submit - parameter 'stack_var'
does not match expected type ARG_PTR_TO_ALLOC_MEM (actual: u64)
[alpha.ebpf.EBPFVerifierChecker]
   20 |     bpf_ringbuf_submit(&stack_var, 0);
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

test_19

```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

SEC("sk_lookup")
int sk_lookup_prog(struct bpf_sk_lookup *ctx)
{
    u32 dport = ctx->local_port;

    if (dport == 8080) {
        struct bpf_sock *sk = bpf_sk_lookup_tcp(ctx, NULL, 0, 0, 0);
        if (sk) {
            bpf_sk_assign(ctx, sk, 0);
            bpf_sk_release(sk);
        }
```

```
    }

    return SK_PASS;
}

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

Verifier Rejection

```
unknown func bpf_sk_lookup_tcp#84
libbpf: prog 'sk_lookup_prog': failed to load: -22
libbpf: failed to load object 'test19.bpf.o'
```

Checker Behaviour

```
./test19.bpf.c:12:31: warning: unknown func 'bpf_sk_lookup_tcp' , not allowed
in section 'sk_lookup' [alpha.ebpf.EBPFVerifierChecker]
   12 |  struct bpf_sock *sk = bpf_sk_lookup_tcp(ctx, NULL, 0, 0, 0);
      |                               ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

test_26

```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

SEC("kprobe/sys_getgid")
int percpu_wrong_test(struct pt_regs *ctx)
{
    u64 regular_var = 42;

    // ERROR: bpf_per_cpu_ptr expects ARG_PTR_TO_PERCPU_BTF_ID
    void *ptr = bpf_per_cpu_ptr(&regular_var, 0);

    return 0;
}

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

Verifier Rejection

```
R1 type=fp expected=percpu_ptr_, percpu_rcu_ptr_, percpu_trusted_ptr_
libbpf: prog 'percpu_wrong_test': failed to load: -13
libbpf: failed to load object 'test26.bpf.o'
```

Checker Behaviour

```
./test26.bpf.c:11:11: warning: Value stored to 'ptr' during its initialization
is never read [deadcode.DeadStores]
   11 |     void *ptr = bpf_per_cpu_ptr(&regular_var, 0);
      |           ^~~   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
./test26.bpf.c:11:17: warning: call bpf_per_cpu_ptr - parameter 'regular_var'
does not match expected type ARG_PTR_TO_PERCPU_BTF_ID (actual: u64)
[alpha.ebpf.EBPFVerifierChecker]
   11 |     void *ptr = bpf_per_cpu_ptr(&regular_var, 0);
      |                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2 warnings generated.
```

5.5.4 Packet Access

These tests target improper access to packet data in XDP or other programs. The verifier ensures that all memory accesses stay within the packet boundaries, and violations should trigger warnings from the checker.

test_01

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

int counter = 0;
SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  if(data + 20 < data_end)
  {
    s[0]=  *((unsigned char *)(data + 15));
  }

  s[1]=  *((unsigned char *)(data + 12));

  bpf_printk("%s", s);
  bpf_printk("Hello World %d", counter);
  counter++;
```

```
  return XDP_PASS;
}

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

Verifier Rejection

```
invalid access to packet, off=12 size=1, R1(id=0,off=12,r=0)
R1 offset is outside of the packet
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test01.bpf.o'
```

Checker Behaviour

```
./test01.bpf.c:18:10: warning: Invalid packet access (offset=12, bound=0),
ptr 'data' [alpha.ebpf.EBPFPacketChecker]
   18 |   s[1]=  *((unsigned char *)(data + 12));
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
```

test_02

```
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

int counter = 0;

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  if(data + 5 <= data_end)
  {
    s[0]=  *((unsigned char *)(data + 4));
    s[2]=  *((unsigned char *)(data + 5));

  } else {
    s[0]=  *((unsigned char *)(data));
  }
```

```
  s[1]=  *((unsigned char *)(data));

  bpf_printk("%s", s);
  bpf_printk("Hello World %d", counter);
  counter++;
  return XDP_PASS;
}
char LICENSE[] SEC("license") = "GPL";
```

Verifier Rejection

```
invalid access to packet, off=5 size=1, R2(id=0,off=5,r=5)
R2 offset is outside of the packet
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test02.bpf.o'
```

Checker Behaviour

```
./test02.bpf.c:17:12: warning: Invalid packet access (offset=5, bound=5),|
ptr 'data' [alpha.ebpf.EBPFPacketChecker]
   17 |     s[2]=  *((unsigned char *)(data + 5));
      |            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
./test02.bpf.c:20:12: warning: Invalid packet access (offset=0, bound=0),|
ptr 'data' [alpha.ebpf.EBPFPacketChecker]
   20 |     s[0]=  *((unsigned char *)(data));
      |            ^~~~~~~~~~~~~~~~~~~~~~~~~
./test02.bpf.c:23:10: warning: Invalid packet access (offset=0, bound=0),|
ptr 'data' [alpha.ebpf.EBPFPacketChecker]
   23 |   s[1]=  *((unsigned char *)(data));
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~
3 warnings generated.
```

test_03

```
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

int counter = 0;

SEC("xdp")
```

```c
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  if(data + 10 > data_end)
  {
    s[0]=  *((unsigned char *)(data + 1));

  } else {
    s[0]=  *((unsigned char *)(data + 3));
  }

  s[1]=  *((unsigned char *)(data));

  bpf_printk("%s", s);
  bpf_printk("Hello World %d", counter);
  counter++;
  return XDP_PASS;

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

Verifier Rejection

```
invalid access to packet, off=1 size=1, R3(id=0,off=1,r=0)
R3 offset is outside of the packet
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test03.bpf.o'
```

Checker Behaviour

```
./test03.bpf.c:16:12: warning: Invalid packet access (offset=1, bound=0),
ptr 'data' [alpha.ebpf.EBPFPacketChecker]
   16 |     s[0]=  *((unsigned char *)(data + 1));
      |            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
./test03.bpf.c:22:10: warning: Invalid packet access (offset=0, bound=0),
ptr 'data' [alpha.ebpf.EBPFPacketChecker]
   22 |   s[1]=  *((unsigned char *)(data));
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~
2 warnings generated.
All done.
```

test_04

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];
  struct ethhdr *eth = data;

  if(data + 15 > data_end)
  {
    return XDP_PASS;
  }

  s[0]=  *((unsigned char *)(data));
  s[1] = *((unsigned char *)eth + 5);
  s[2]=  *((unsigned char *)(data + 17));

  eth = (struct ethhdr *)((unsigned char *)eth + 10);

  s[3] = *((unsigned char *)eth + 2);
  s[4] = *((unsigned char *)eth + 10);

  bpf_printk("%s", s);
  return XDP_PASS;
}
char LICENSE[] SEC("license") = "GPL";
```

Verifier Rejection

```
invalid access to packet, off=17 size=1, R1(id=0,off=17,r=15)
R1 offset is outside of the packet
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test04.bpf.o'
```

Checker Behaviour

```
./test04.bpf.c:20:10: warning: Invalid packet access (offset=17, bound=15),
ptr 'data' [alpha.ebpf.EBPFPacketChecker]
   20 |   s[2]=  *((unsigned char *)(data + 17));
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
./test04.bpf.c:25:10: warning: Invalid packet access (offset=20, bound=15),
ptr 'eth' [alpha.ebpf.EBPFPacketChecker]
   25 |   s[4] = *((unsigned char *)eth + 10);
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~
2 warnings generated.
All done.
```

test_05

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data_end = (void *)(long)ctx->data_end;
  char s[5];
  struct ethhdr *eth_end = data_end;

  s[0]=  *((unsigned char *)(data_end));
  s[1] = *((unsigned char *)eth_end );

  bpf_printk("%s", s);
  return XDP_PASS;

}

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

Verifier Rejection

```
R1 invalid mem access 'pkt_end'
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test05.bpf.o'
```

Checker Behaviour

```
./test05.bpf.c:13:10: warning: 'data_end' invalid mem access 'pkt_end'
[alpha.ebpf.EBPFPacketChecker]
   13 |   s[0]=  *((unsigned char *)(data_end));
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
./test05.bpf.c:15:10: warning: 'eth_end' invalid mem access 'pkt_end'
[alpha.ebpf.EBPFPacketChecker]
   15 |   s[1] = *((unsigned char *)eth_end );
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~
2 warnings generated.
All done.
```

test_06

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  if(data + 5 < data_end - 2) {
    s[0]=  *((unsigned char *)(data + 2));
  }

  bpf_printk("%s", s);
  return XDP_PASS;

}

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

Verifier Rejection

```
R1 pointer arithmetic on pkt_end prohibited
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test06.bpf.o'
```

Checker Behaviour

```
./test06.bpf.c:12:26: warning: 'data_end' pointer arithmetic on pkt_end
prohibited [alpha.ebpf.EBPFPacketChecker]
    12 |   if(data + 5 < data_end - 2) {
       |                 ~~~~~~~~~^~~
1 warning generated.
All done.
```

test_07

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  int a= 30;
  if(data + a > data_end)
  {
    return XDP_PASS;
  }
  s[0] = *((unsigned char *)(data + 10));

  bpf_printk("%s", s);
  return XDP_PASS;

}

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

Verifier Rejection

```
The code is accepted
```

Checker Behaviour

```
Using clang: /home/oscar/test-supervisor-clean/llvm-project/install/bin/clang
Using checker include (if exists): /home/oscar/test-supervisor-clean/
```

```
llvm-project/install/include/clang/StaticAnalyzer/Checkers

>>> Running analyzer...
/home/oscar/test-supervisor-clean/llvm-project/install/bin/clang
--target=bpf --analyze -Xclang -analyzer-checker=alpha.ebpf.EBPFPacketChecker
- Xclang -I/home/oscar/test-supervisor-clean/llvm-project/install/include/
clang/StaticAnalyzer/Checkers ./test07.bpf.c

All done.
```

A correct packet access

test_08

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  struct ethhdr *eth = data;

  if((void *)eth + 12 > data_end)
  {
    return XDP_PASS;
  }

  s[0] = *((unsigned char *)(eth + 16));

  bpf_printk("%s", s);
  return XDP_PASS;
}

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

Verifier Rejection

```
invalid access to packet, off=224 size=1, R1(id=0,off=224,r=12)
R1 offset is outside of the packet
```

```
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test08.bpf.o'
```

Checker Behaviour

```
./test08.bpf.c:19:10: warning: Invalid packet access (offset=224, bound=12),
ptr 'eth' [alpha.ebpf.EBPFPacketChecker]
   19 |   s[0] = *((unsigned char *)(eth + 16));
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
All done.
```

test_09

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  struct ethhdr *eth = data;

  if ((void *)(eth + 2) > data_end)
  {
    return XDP_PASS;
  }

  s[0] = *((unsigned char *)((void *)eth + 30));

  bpf_printk("%s", s);
  return XDP_PASS;

}

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

Verifier Rejection

```
invalid access to packet, off=30 size=1, R1(id=0,off=30,r=28)
```

```
R1 offset is outside of the packet
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test09.bpf.o'
```

Checker Behaviour

```
./test09.bpf.c:19:10: warning: Invalid packet access (offset=30, bound=28),
ptr 'eth' [alpha.ebpf.EBPFPacketChecker]
   19 |   s[0] = *((unsigned char *)((void *)eth + 30));
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
All done.
```

test_10

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  struct ethhdr *eth = data;

  if ((struct ethhdr *)(eth + 10) > (struct ethhdr *)data_end)
  {
    return XDP_PASS;
  }

  s[0] = *((unsigned char *)(eth + 11));

  bpf_printk("%s", s);
  return XDP_PASS;

}

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

Verifier Rejection

```
invalid access to packet, off=154 size=1, R1(id=0,off=154,r=140)
R1 offset is outside of the packet
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test10.bpf.o'
```

Checker Behaviour

```
./test10.bpf.c:19:10: warning: Invalid packet access (offset=154, bound=140), p
   19 |   s[0] = *((unsigned char *)(eth + 11));
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
All done.
```

test_11

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  struct ethhdr *eth = data;

  if ((void *)(eth + 1) > data_end)
  {
    return XDP_PASS;
  }

  s[0] = *((unsigned char *)(eth + 11));

  bpf_printk("%s", s);
  return XDP_PASS;
}

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

Verifier Rejection

```
invalid access to packet, off=154 size=1, R1(id=0,off=154,r=14)
R1 offset is outside of the packet
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test11.bpf.o'
```

Checker Behaviour

```
./test11.bpf.c:19:10: warning: Invalid packet access (offset=154, bound=14),
ptr 'eth' [alpha.ebpf.EBPFPacketChecker]
   19 |   s[0] = *((unsigned char *)(eth + 11));
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
All done.
```

test_12

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {

  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  struct ethhdr *eth = data;

  if((void *)eth + 10 > data_end)
  {
    return XDP_PASS;
  }
  s[0] = *((unsigned char *)((void *)eth + 5));
  s[1] = *((unsigned char *)(data + 8));

  eth= eth + 20;
  if(data + 12 <= (void*) eth)
  {
    s[2] = *((unsigned char *)(data+ 20));
  }

  bpf_printk("%s", s);
  return XDP_PASS;
}
```

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

Verifier Rejection

```
invalid access to packet, off=20 size=1, R1(id=0,off=20,r=10)
R1 offset is outside of the packet
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test12.bpf.o'
```

Checker Behaviour

```
./test12.bpf.c:24:12: warning: Invalid packet access (offset=20, bound=10),
ptr 'data' [alpha.ebpf.EBPFPacketChecker]
   24 |     s[2] = *((unsigned char *)(data+ 20));
      |            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
All done.
```

test_13

```
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {
  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  __u32 offset_dinamico = ctx->ingress_ifindex;

  if(offset_dinamico < 0 ||  offset_dinamico > 5)
    return XDP_PASS;

  void* tmp = data + offset_dinamico;

  if(tmp + 10 < data_end) {
    s[1] = *((unsigned char*)tmp + offset_dinamico +100);
  }

  s[2] = '\0';
```

```
  bpf_printk("%s", s);

  return XDP_PASS;
}

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

Verifier Rejection

```
invalid access to packet, off=100 size=1, R1(id=2,off=100,r=0)
R1 offset is outside of the packet
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test13.bpf.o'
```

Checker Behaviour

```
./test13.bpf.c:19:12: warning: Invalid packet access (offset=100, bound=10),
ptr 'tmp' [alpha.ebpf.EBPFPacketChecker]
    19 |     s[1] = *((unsigned char*)tmp + offset_dinamico +100);
       |            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
All done.
```

test_15

```
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int test_bounds(struct xdp_md *ctx) {
  void *data = (void *)((long)ctx->data);
  void *data_end = (void *)(long)ctx->data_end;
  char s[5];

  __u32 offset_dinamico = ctx->ingress_ifindex;

  if(offset_dinamico < 0 ||  offset_dinamico > 5)
    return XDP_PASS;

  void* tmp = data + offset_dinamico;

  if(tmp + offset_dinamico + 10 < data_end) {
```

```
    s[1] = *((unsigned char*)tmp + 100);
  }

  s[2] = '\0';
  bpf_printk("%s", s);

  return XDP_PASS;
}

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

Verifier Rejection

```
invalid access to packet, off=100 size=1, R1(id=1,off=100,r=0)
R1 offset is outside of the packet
libbpf: prog 'test_bounds': failed to load: -13
libbpf: failed to load object 'test15.bpf.o'
```

Checker Behaviour

```
./test15.bpf.c:19:12: warning: Invalid packet access (offset=100, bound=10),
ptr 'tmp' [alpha.ebpf.EBPFPacketChecker]
   19 |     s[1] = *((unsigned char*)tmp + 100);
      |            ^~~~~~~~~~~~~~~~~~~~~~~~~~~
1 warning generated.
All done.
```

5.6 Fuzzer Tests

This section reports the results obtained on fuzzer-generated programs. The programs were produced by inverting a semantic fuzzer based on the BRT framework, so that each generated test contains at least one intentional verifier rejection. The fuzzer output was additionally analyzed by the *pretty-verifier* tool, which provides a normalized classification of verifier failures [13]. Three batches were considered, containing 307, 385, and 45 non-homogeneous test files, respectively.

5.6.1 Detection classification

To quantify the effectiveness of the proposed static analyzers, outcomes are classified using the standard notions of True Positives (TP), False Positives (FP), and False Negatives (FN), by comparing checker warnings against the verifier-reported error category.

- **Detected by Verifier (V)**: number of test programs rejected by the verifier for a given error category.
- **Detected by Checkers (C)**: number of warnings generated by the implemented static analyzers.
- **True Positive (TP)**: the checker emits a warning and the verifier reports the same error for that error category.
- **False Positive (FP)**: the checker emits a warning for a given error category, while the verifier rejects the program due to an error belonging to a different category.
- **False Negative (FN)**: the checker does not emit a warning for a given error category, but the verifier reports an error of that category.
- **TP + FN = V**
- **TP + FP = C**

Although the fuzzer generates programs that always contain at least one rejection by the verifier, the evaluation is performed against specific error categories. Therefore, the presence of false positives does not imply that the analyzed program is correct. Instead, it indicates that the checker reported an error belonging to a different category than the one reported by the verifier. Therefore, the checker detects a potential problem, but the detected error does not correspond to the actual error found by the verifier for that program.

Table 5.1: Detection rate on files containing verifier errors covered by the implemented checkers

Batch	Total Files	Files with Verifier Errors in Covered Categories	Files Correctly Detected by Checkers	Detection Rate (%)
Batch 1	307	80	30	37.50
Batch 2	385	61	10	16.39
Batch 3	45	15	12	80.00

Table 5.2: Batch 1: Verifier errors detected by the static analyzer with percentages.

Verifier Error Type	**Detected by V**	**Detected by C**	**TP (% on V)**	**FP (% on C)**	**FN (% on V)**
License Missing (covered)	0	18	0 (N/A)	18 (100%)	0 (N/A)
Helper not allowed (covered)	3	17	0 (0.0%)	17 (100%)	3 (100%)
Wrong argument passed to helper function (covered)	69	59	30 (43.5%)	29 (49.2%)	39 (56.5%)
Invalid access to map value (covered)	7	0	0 (0.0%)	0 (N/A)	7 (100%)
Invalid access to packet (covered)	1	0	0 (0.0%)	0 (N/A)	1 (100%)
Others (not covered)	227	0	0 (0.0%)	0 (0.0%)	227 (100%)

Table 5.3: Batch 2: Verifier errors detected by the static analyzer with percentages

Verifier Error Type	**Detected by V**	**Detected by C**	**TP (% on V)**	**FP (% on C)**	**FN (% on V)**
License Missing (covered)	0	46	0 (N/A)	46 (100.0%)	0 (N/A)
Helper not allowed (covered)	21	22	0 (0.0%)	22 (100.0%)	21 (100.0%)
Wrong argument passed to helper function (covered)	21	76	10 (47.6%)	66 (86.8%)	11 (52.4%)
Invalid access to map value (covered)	19	0	0 (0.0%)	0 (N/A)	19 (100.0%)
Invalid access to packet (covered)	0	0	0 (N/A)	0 (N/A)	0 (N/A)
Others (not covered)	324	0	0 (0.0%)	0 (N/A)	324 (100.0%)

Table 5.4: Batch 3: Verifier errors detected by the static analyzer with percentages

Verifier Error Type	Detected by V	Detected by C	TP (% on V)	FP (% on C)	FN (% on V)
License Missing (covered)	2	4	1 (50%)	3 (75%)	1 (50%)
Helper not allowed (covered)	1	6	1 (100.0%)	5 (83.3%)	0 (0.0%)
Wrong argument passed to helper function (covered)	3	2	2 (66.7%)	0 (0.0%)	1 (33.3%)
Invalid access to map value (covered)	3	2	2 (66.7%)	0 (0.0)	1 (33.3%)
Invalid access to packet (covered)	6	6	6 (100.0%)	0 (0.0%)	0 (0.0%)
Others (not covered)	30	0	0 (0.0%)	0 (N/A)	30 (100.0%)

Other verifier errors not covered by the implemented checkers. The following error categories are currently not targeted by dedicated checkers and therefore appear in the "Others" group:

- 32-bit ALU operations on pointers produce (meaningless) scalars
- bpf_spin_lock usage requires map `map_0` to have a type with BTF
- bpf_timer usage requires map `map_0` to have a type with BTF
- Cannot read from read only map
- Function `bpf_get_func_ip` is supported only for fentry/fexit programs
- Helper function might sleep in a non-sleepable prog
- Invalid access to memory region
- Map configuration error
- Pointer arithmetic on pointer to map element value not null-checked prohibited on possibly null type
- Tail call invocation before releasing reference leads to reference leak
- The return value may contains values equal to 2
- The return value may contains values equal to 3
- Unbounded memory access of the variable `v2`
- Unbounded memory access of the variable `v7`

5.7 Evaluation results

The experimental evaluation highlights both the strengths and current limitations of the proposed static analysis framework. Manual test cases demonstrate that the implemented checkers are able to accurately reproduce several classes of verifier rejections at source level, providing precise and user-friendly diagnostics for licensing violations, invalid helper usage, out-of-bounds map accesses, and unsafe packet memory accesses. In all these scenarios, the analyzer successfully anticipates verifier failures, allowing developers to detect critical issues before bytecode generation and kernel loading. In the automatic fuzzer-based tests overall detection rate varies across batches, so the results show that the proposed analyzer achieves precision when verifier errors fall within the categories explicitly modeled by the checkers. In particular, the third batch reports a detection rate of 80%, confirming the effectiveness of the approach in controlled and well-covered scenarios. False positives mainly arise from conservative constraint approximations performed by the Clang Static Analyzer. Overall, the results confirm that the proposed static analyzer represents a valuable complementary tool to the eBPF verifier. Although it does not aim to replace kernel-level verification, it significantly improves developer feedback at source-code level, reducing debugging time and improving programmability, and also the modular design of the checker infrastructure also enables extension toward currently uncovered verifier error classes

5.8 Checker Limitations

The main limitations of the proposed checkers do not primarily derive from the checker implementation itself, but from the intrinsic differences between the analysis model adopted by the Clang Static Analyzer (CSA) and the one implemented by the eBPF verifier. Although both tools analyze the same source code, they operate on different representations and follow distinct verification strategies.
In particular, the eBPF verifier analyzes the program after compilation, on eBPF bytecode, which may differ significantly from the original C source due to compiler optimizations and transformations. As a consequence, certain control-flow structures, pointer expressions, and variable usages observed at source level may not directly correspond to the constructs analyzed by the verifier. This mismatch can lead to discrepancies between the warnings produced by CSA-based checkers and the errors reported by the eBPF verifier.
Another important source of divergence concerns the propagation of conditional bounds. CSA propagates symbolic constraints across branches following its own conservative range analysis model, while the eBPF verifier applies a different constraint reasoning strategy based on register states and path-sensitive tracking. These different models may result in situations where a bound is considered sufficiently constrained by CSA but remains partially unbounded for the verifier, or vice versa, leading to both false positives and false negatives.
A further limitation arises in the handling of values that are not known at compile time. When program variables depend on runtime inputs or external sources, CSA represents them as symbolic values with inferred ranges, whereas the eBPF verifier applies stricter safety-oriented assumptions. This difference in abstraction levels further contributes to the divergence in analysis outcomes between the two systems.
Overall, these limitations reflect the fundamental challenge of aligning a source-level static analysis framework with a bytecode-level kernel verifier. Such discrepancies do not invalidate the proposed approach, but rather highlight the complexity of eBPF verification and the need for complementary analysis techniques. The proposed checkers remain effective in anticipating a significant subset of verifier errors and provide valuable early feedback to developers, while future improvements can focus on refining the constraint models and reducing the semantic gap between CSA and the eBPF verifier.

Chapter 6

Conclusions and Future Work

This thesis addressed the challenges faced by eBPF developers when writing source code that must satisfy the strict constraints enforced by the eBPF verifier. In particular, the goal was to provide earlier feedback in the development pipeline by detecting potential verifier rejections directly at source-code level, allowing programmers to identify and correct issues before compilation and program loading. The adopted approach was based on the development of dedicated static analysis checkers using the Clang Static Analyzer framework from the LLVM ecosystem. This choice enabled source-level analysis of C programs without requiring code execution, making it possible to detect eBPF-related violations through static reasoning.

Several checkers and supporting tools were implemented to target a representative set of common verifier error categories, including missing or incompatible license declarations, incorrect helper function parameter types, out-of-bounds accesses on global arrays, and unsafe accesses to packet memory through pointer arithmetic.

The experimental evaluation demonstrated that, although the analysis still presents imprecisions in certain scenarios, the warnings produced by the proposed framework are largely consistent with the errors reported by the eBPF verifier. In particular, when the analyzed programs fall within the categories explicitly modeled by the implemented checkers, the analyzer is able to accurately anticipate verifier rejections at source level, providing meaningful and actionable diagnostics to the developer.

In conclusion, the results confirm that source-level static analysis represents an effective complementary approach to kernel-level verification. While the proposed framework does not aim to replace the eBPF verifier, it significantly improves programmability and debugging efficiency by shifting error detection earlier in the development process. Furthermore, the modular structure of the checker

architecture provides a solid foundation for future extensions toward broader coverage of verifier error classes.

6.1 Future Works

6.1.1 Bound Checking on Helper Function Parameters

The current implementation of the *EBPFVerifierChecker* validates helper function invocations by verifying that the types of the passed parameters match the types specified in the helper function signatures. A natural extension of this work consists in introducing bound checking on a subset of helper function parameters for which range validation is required by the eBPF verifier. For instance, when a helper function operates on an eBPF map, the checker could verify that the key parameter is constrained within a range that is consistent with the map size and key type. By propagating and validating such bounds at source level, the checker would be able to anticipate verifier rejections related to invalid key ranges or out-of-bounds map accesses. This enhancement would allow the static analyzer to detect a broader class of verifier errors, further improving early feedback to developers and reducing the number of failures observed only at program load time. Moreover, integrating bound checking into helper parameter validation would bring the checker closer to the semantic model adopted by the eBPF verifier, reducing the gap between source-level and bytecode-level analysis.

6.1.2 Handling One Million Instruction Limit

A well-known limitation of the eBPF verifier is the maximum program size constraint: any eBPF program whose bytecode exceeds one million instructions is automatically rejected. For this reason, an initial idea of this work was to introduce a dedicated checker aimed at predicting whether a given eBPF program could exceed this instruction limit.
The proposed approach was based on analyzing all possible control-flow paths of the input C program using the Clang Static Analyzer. Since conditional statements and loops generate multiple execution paths, the idea was to construct all Control Flow Graphs (CFGs), count the number of nodes associated with each path, and compute an overall estimation of the number of instructions that would later be processed by the eBPF verifier. This estimation could have provided a preliminary indication of whether the program might exceed the verifier instruction limit.
However, this approach was eventually discarded for two main reasons. First, the eBPF compilation process applies several optimizations and transformations that can significantly alter the number of generated bytecode instructions in a way that is difficult to predict from the source code alone. As a result, any estimation based

solely on the C-level CFG would be inherently inaccurate.
Second, the Clang Static Analyzer itself introduces additional limitations. Due to its conservative analysis strategy, CSA may prune entire execution branches, merge paths, or stop exploring paths that exceed internal complexity thresholds. Furthermore, CSA is not designed to handle programs with extremely large numbers of execution paths or instructions, making it unsuitable for reliably estimating instruction counts at this scale.
For these reasons, the implementation of a checker targeting the one-million-instruction limit was considered impractical within the scope of this work. Nevertheless, this topic remains an interesting direction for future research.

Bibliography

[1] Liz Rice. *Learning eBPF*. Isovalent, 2020. URL: https://cilium.isovalent.com/hubfs/Learning-eBPF%20-%20Full%20book.pdf (cit. on pp. 2, 3, 9).

[2] eBPF.io. *eBPF Official Documentation*. URL: https://ebpf.io/ (cit. on pp. 2, 3, 5, 8, 9).

[3] Linux Kernel Documentation. *BPF Verifier*. https://docs.kernel.org/bpf/verifier.html. Accessed: 2025-12-27 (cit. on pp. 2, 3, 9).

[4] eBPF.io Documentation. *Linux eBPF Helper Functions*. Accessed: 2025-12-27. 2025. URL: https://docs.ebpf.io/linux/helper-function/ (cit. on pp. 2, 5, 9, 26).

[5] Bruno Cardoso Lopes and Rafael Auler. *Getting Started with LLVM: Core Libraries*. Get to grips with LLVM essentials and use the core libraries to build advanced tools. Packt Publishing, 2020 (cit. on pp. 2, 16, 18, 20).

[6] LLVM Project. *The LLVM Compiler Infrastructure Project*. Accessed: 2025-12-27. 2025. URL: https://llvm.org/ (cit. on pp. 2, 16, 17).

[7] LLVM Project. *Clang Doxygen Documentation*. Accessed: 2025-12-27. 2025. URL: https://clang.llvm.org/doxygen/ (cit. on pp. 2, 16, 23).

[8] Linux Kernel Project. *GPL-compatible license identifiers for eBPF programs*. https://github.com/torvalds/linux/blob/master/include/linux/license.h. Accessed: 2025-12-27. 2025 (cit. on pp. 12, 26).

[9] Linux Kernel Project. *eBPF Helper Function Prototypes*. https://github.com/torvalds/linux/blob/master/kernel/bpf/helpers.c. Accessed: 2025-12-27. 2025 (cit. on pp. 35, 38).

[10] TRUSS Lab. *BRF eBPF Type Definitions*. https://github.com/trusslab/brf/blob/dev/prog/brf_types.go. Accessed: 2025-12-27. 2025 (cit. on pp. 35, 38).

[11] LLVM Project. *LLVM/Clang Compiler*. https://github.com/llvm/llvm-project. Commit df570dadcb93a32c308d31c2ab54d2f46c8ae0c0, accessed 2025-12-27. 2025 (cit. on p. 52).

[12] Linux Kernel Organization. *Linux Kernel Version 6.8*. `https://www.kernel.org`. Ubuntu 6.8.0-87-generic used for test execution, accessed 2025-12-27. 2025 (cit. on p. 52).

[13] Rosario Rizza, Riccardo Sisto, and Fulvio Valenza. «Design and implementation of a tool to improve error reporting for eBPF code». In: *2025 IEEE International Conference on Cyber Security and Resilience (CSR)*. 2025, pp. 214–219. DOI: `10.1109/CSR64739.2025.11130075` (cit. on p. 80).