POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Design and implementation of a tool to improve error reporting for eBPF code

Supervisors

Prof. Riccardo SISTO

Prof. Fulvio VALENZA

Candidate

Rosario RIZZA

October 2024

Summary

The eBPF technology is the uprising trend in the cloud computing world, allowing programs to run directly into the kernel space. This leads to having much more control over the system the program is run upon, especially in security and performance sensitive environments like the server operating system. For instance, it enables efficient monitoring and observability by tracking system performance, system calls, and recognising latency issues with minimal overhead, allowing easy implementation of those security policies that need to be enforced in the kernel space. Additionally, eBPF is highly effective in networking and packet filtering, facilitating the creation of custom firewalls, load balancers, and traffic optimizations.

In order to load eBPF programs into the kernel space, they need to be severely scrutinized by the eBPF verifier, a set of deep checks that prevent it from crashing the kernel or, even worse, from escalating privileges and taking control of the system. The main drawback of this process is that error messages that the verifier produces are extremely difficult to read, and also detached from the language the developer writes the code in. This is because the common language used for this type of task is the C language (recently Rust is starting to be adopted), while the verifier performs the check on the eBPF bytecode, which is the result of the compilation process. Moreover, the checks are executed only during the loading of the program into the kernel space, and not during the compilation of the C code.

The main goal of my thesis is to improve the developer's experience by inserting into the eBPF compilation pipeline a set readability improvements, creating a tool that allows the developers to easily reach the line of C code that generated the error and adding a simple explanation of what happened, with some suggestion to fix it.

The most common eBPF compilation pipeline at the moment uses the libbpf project and the Clang compiler, so this thesis aims to add the integrations using the error output of the libbpf *load* command. The tool will parse the error, the source and object files in order to inform the developer on the reason and the location. The python regex library *re* will be used, coupled with some bash script and the *pipe* command.

Acknowledgements

ACKNOWLEDGMENTS

Table of Contents

List of Tables

List of Figures

Acronyms

BPF
: Berkeley Packet Filter

eBPF
: Extended Berkeley Packet Filter

LLVM
: Low Level Virtual Machine

BCC
: BPF Compiler Collection

CVE
: Common Vulnerabilities and Exposures

Introduction

eBPF, or extended Berkeley Packet Filter, is a powerful kernel technology that allows developers to execute custom code within the kernel of the Linux operating system. This capability enables elevated flexibility and performance for networking, observability, and security tools, all while maintaining a high level of safety and reliability. By running within the kernel, eBPF programs can monitor and modify the behavior of system calls, network packets, and various other kernel-level events without requiring modifications to the underlying applications.

The core enabler of this technology is the eBPF verifier, a static analyzer applied on the eBPF programs intended to run on the kernel with the purpose to reject any program that could compromise the execution or the security of the kernel. For instance, a program could accidentally deference a null pointer, leading to a system failure, or even worse could access to unauthorized memory, leaking kernel space resources. For this reason the usage of this application is crucial and mandatory if we want to run eBPF programs on the kernel. In order to run the eBPF program, the developer needs to first compile the eBPF program to a common target assembler-like language, called eBPF bytecode, that then will be analyzed by the eBPF verifier. In particular the verification process will be performed during the *loading* process into the kernel space, through the `bpf()` system call. However usually a C library, called `libbpf`, is used to perform this procedure inside an external program, the user space program, that manages the life cycle of the eBPF program (also called kernel space program), like also communicating information to the user space. A command line tool of this library is also available, `bpftool`, to allow loading the eBPF program if there is no need of communicating with the user space. During the loading process, the `bpf()` system call, or the libbpf library, outputs the eBPF verifier log, that communicate if the program analysis yielded a correct result or an error was found. In this situations, the output shown is often hard to understand, for example it refers to C enums defined in the kernel source code, or most commonly it refers to the eBPF bytecode it used to perform the analysis. Considering that the developer writes eBPF programs usually in C code, and sometimes in Rust, this can be a limitation.

The goal of this thesis is to improve readability of the eBPF verifier log and enrich its connections with the C source code, which is the most used language for this purpose, thus improving the developer experience. In order to achieve this result, multiple path will be taken into consideration, initially considering to mimic the eBPF verifier checks directly in the C language, through the usage of the Clang static analyzer, a suite that allows to implement static analysis techniques on the C code before compilation. This approach will then be discarded in favour of the message error analysis, using directly the eBPF verifier log, with the help of the debug mode of the Clang compilation, managing to deliver an error message referred to the C source code and possibly similar to the other approach. For this reason a full study of the eBPF verifier source code will be performed, in order to gather the information to choose which error messages need to be managed and to then deliver this information to the developers in the occurrence of an error message. Numerous difficulties will be encountered, mainly the struggle into creating test cases for most of the error that have been handled. However the tool functionalities for the most common message errors will be tested, yielding a decently satisfying result.

In the chapter 1 there will be an overview of the eBPF technology, the history, the main usage and some information about the Linux kernel and how it works.
In the chapter 2 there will be a deep dive in the the way the eBPF programs are developed and executed, and a section reserved to explaining how the verifier works and its criteria for accepting or rejecting a program. These first two chapters use the information found in Rice [1], the Wikipedia pages on eBPF [2] and the Linux Kernel [3], the eBPF docs [4], the Linux Kernel documentation [5], and the work of Shung-Hsi Yu at SUSE Labs [6].
In the chapter 3, there will be the description of the work associated with the thesis,the context that is taken into consideration and the approaches that where identified and the considerations made upon their feasibility, pros and cons, and a description of the study process of the eBPF verifier. This one in particular was the most time consuming part of the work, due to the considering size of the eBPF verifier.
In the chapter 4 there will be a presentation of the tool developed, the structure and the execution flow, other then the framework that it created in order to deliver the best message error, and some specific of the actual improvement on the error messages.
In chapter 5 the testing of the tool is presented, talking about the test suite developed, the testing methodologies, difficulties encountered and the conclusions.
In the chapter 6, the final, the conclusion of the thesis are made, and a list of future work is presented, especially focusing on maintainability and testing. In the Appendix A there is analysis performed on the eBPF verifier, exposing for each

error message the type of error it represented, some notes and if it was managed and tested.

Chapter 1

Overview of eBPF

1.1 eBPF History

The roots of eBPF trace back to the Berkeley Packet Filter (BPF), which was first introduced in the early 1990s as a means to efficiently filter network packets in Unix-based systems. BPF provided the possibility to execute simple filtering programs written in a custom instruction set, designed to determine whether a network packet should be accepted or rejected. Initially, BPF was employed in packet capturing utilities like `tcpdump`, where it enabled high-performance packet filtering directly within the kernel.

As the demand for more sophisticated networking and security tools grew, BPF evolved. In 2012 `seccomp-bpf` was introduced in version 3.5 of the kernel, allowing the control of system calls by BPF programs. By 2014, with the release of Linux kernel version 3.18, BPF had been overhauled and extended, leading to the creation of eBPF. This transformation involved several key developments, like re-engineering of the original BTF instruction set to be more efficient on 64-bit machines, introduction of eBPF maps, data structures that allow communication between eBPF programs and user space applications, the introduction of the `bpf()` system call, to allow user space applications to interact with eBPF programs in the kernel, new helper functions, and most importantly the introduction of the eBPF verifier, a component that checks for the correctness and safety of the code during loading time of eBPF programs, in order to avoid adding vulnerabilities or instructions that would lead to a system failure. In 2015, `kprobes` were introduced for tracing, and later that year the eBPF compiler back end got merged into LLVM 3.7.0 release.

Since then the companies started to introduce this technology in their production projects, starting from Netflix, Facebook, and then leading to the development of the tool used nowadays for the eBPF development, like libbpf, BCC, Cilium and many others.

1.2 Applications and Capabilities of eBPF

eBPF's flexibility and power have led to its adoption across a wide range of applications, particularly in the areas of performance tracing, networking, and security.

1.2.1 Performance Tracing

eBPF allows for granular, low-overhead performance tracing across virtually any aspect of a Linux system. Developers can write eBPF programs to trace specific system calls, monitor the execution time of functions, or observe the behavior of applications at the kernel level. This capability is fundamental for diagnosing performance bottlenecks, understanding system behavior, and optimizing application performance.

1.2.2 High-Performance Networking

In the realm of networking, eBPF has revolutionized the way packet processing is handled. By attaching eBPF programs to various points in the networking stack, developers can implement custom routing, load balancing, and firewall solutions with performance that rivals or even surpasses traditional methods. For example, the XDP (eXpress Data Path)[7] feature leverages eBPF to perform high-speed packet processing directly in the network driver, significantly reducing latency and increasing throughput.

1.2.3 Security

Security is another area where eBPF is used. With eBPF, it is possible to monitor and enforce security policies at the kernel level, providing deep visibility into system behavior and the ability to prevent malicious activity. For instance, eBPF programs can be used to detect and block suspicious system calls, monitor network traffic for signs of compromise, or enforce container security policies in a Kubernetes environment.

1.2.4 eBPF in Cloud-Native Environments

As cloud-native computing becomes increasingly prevalent, eBPF's role in these environments is expanding. In Kubernetes clusters, where multiple containers share the same kernel, eBPF provides a unique vantage point for monitoring and securing workloads. Unlike traditional sidecar-based approaches, eBPF does not require modifying the application's deployment configuration. This makes it easier to deploy and maintain, and ensures comprehensive visibility across all processes and containers running on a given node.

eBPF's ability to dynamically load and unload programs without restarting the kernel or applications is particularly advantageous in cloud environments, where availability and performance are critical. This flexibility, combined with eBPF's deep integration with the kernel, makes it an ideal solution for enhancing the security, observability, and performance of cloud-native applications.

1.3 Linux Kernel

The Linux kernel is the core component of the Linux operating system, which powers a vast range of devices from personal computers to supercomputers, mobile devices, and embedded systems. Since its inception in the early 1990s, the Linux kernel has become one of the most successful open-source software projects in history, largely due to its robust architecture, flexibility, and the collaborative nature of its development.

eBPF is a technology specifically created for the operative systems using the linux kernel, even though recently Microsoft is developing the [8] project, aiming to allow Windows machines to leverage the same benefits present in the Linux machines.

1.3.1 User space and kernel space

The Linux kernel is the core software layer that interacts directly with hardware, while applications operate in user space, an unprivileged area that can't directly access hardware. Instead, applications request the kernel to perform actions on their behalf through system calls, such as file operations, network communication, or memory access. The kernel also manages multiple concurrent processes, allowing several applications to run simultaneously.

Developers usually don't interact directly with system calls because programming languages and libraries provide higher-level abstractions. This abstraction often leads to a lack of awareness about the kernel's role in running programs. eBPF takes this further by allowing us to observe and analyze how applications interact with the kernel by injecting instrumentation into the kernel itself.

For example, eBPF can intercept the system call for opening files, providing insight into which files an application accesses. Traditionally, adding such functionality would require modifying the Linux kernel, which is a complex and challenging task. The kernel has around 30 million lines of code, and making changes requires a deep understanding of its existing code base. Furthermore, contributing changes to the official Linux kernel is difficult, as they must be accepted by the broader community and meet the diverse needs of all Linux users. Even if a change is accepted, it may take years to become widely available due to the long release cycles of Linux distributions.

An alternative to modifying the kernel directly is writing kernel modules, which can be loaded and unloaded on demand. However, kernel modules still involve complex kernel programming, and there are significant risks involved. If a kernel module crashes, it can bring down the entire system. Additionally, users must trust that the module is secure and free from vulnerabilities, as it has access to all system data.

Linux distributions take considerable time to adopt new kernel releases to ensure that the kernel is stable and secure. In contrast, eBPF provides a safer approach by including a eBPF verifier that checks if an eBPF program is safe to run before it is loaded into the kernel. The verification process ensures that the program won't cause any error that may stop the kernel or introduce possible vulnerabilities, offering a secure way to extend kernel functionality without the risks associated with traditional kernel programming.

Chapter 2

eBPF programming

The creation of an eBPF program is performed with specific tools and libraries that allow it to interact with the kernel, as during the loading, as during the execution. Usage of eBPF maps, kernel and helper functions, global variables and other abstractions are necessary for the accomplishment of more complex programs.

2.1 User space and kernel space program

When programming an eBPF program it's necessary to make a distinction between user space programs and kernel space programs.

The kernel space program is the one that we refer to as eBPF program, since it is the one running in the kernel space. It is the one that the eBPF verifier checks, so also the most delicate to work on. It is usually written in C, but Rust can be used as well. The usage of a compiled language is fundamental to execute code on the kernel. Programming in the eBPF kernel space is usually complex, since the limits imposed by the eBPF verifier. For this reason the helper functions were created, a list of functions [9] that allow the program to interact with maps, with the kernel and to perform basic functions that would not be allowed in kernel space (like loops and backwards jumps). Recently also kernel functions were introduced, leveraging the usage of kernel code directly from the eBPF program. In many use cases it can be used as a stand alone program, for example when it acts modifying network packets (so it does not need any user interaction) or when logging system call execution (the user can see the log visualizing the file where it is traced).

However, the user may need to interact with the program running in the kernel space, in order to condition the execution of the kernel program, feeding data, or also getting data in a more structured way. In these scenarios the execution of the eBPF program should be parallel to the execution of a program in the user space. Those programs can be created besides their necessity, since they allow the user to manage the loading, the unloading, the attachment in a single entry point. The main example of this kind of programming is the usage of the BCC framework. BCC is a python eBPF development framework that integrates the eBPF kernel code as a block of text inside the python user space program. It is used for simple applications and usually not in a production environment. For more complex applications the libbpf C library is used to manage the life cycle of the user space program. The two programs can then communicate through the usage of eBPF maps, which are various types of data structures that the programmer can implement simultaneously in the user space as in the kernel space to allow data exchange.

eBPF maps

eBPF maps serve as data structures that allow communication between user space and kernel space programs. These maps act as shared memory spaces where data can be stored and retrieved concurrently by both user space and kernel space programs. There are several types of eBPF maps, each suited for different data handling needs, such as hash maps, arrays, and ring buffers.

In the typical workflow, maps can be used to communicate data metrics gathered from the eBPF program to the user space or even to other eBPF programs. In this scenario, the eBPF program can be limited to the gathering operations, letting it being more performing, and the user space program can manage the data analysis part. Maps are also useful to change the configuration state of an eBPF program during runtime, setting configuration variables that allow the program to take different execution paths that were previously designed. For instance, a user space program might update a configuration map to adjust the behavior of an eBPF packet filter changing the filtering policies.

eBPF maps use predefined data structures to organize and exchange data between user space and kernel space. These structures are defined and validated through BTF, which encodes type information about these structures, allowing the eBPF verifier to ensure that the data layout is consistent and safe to use. When loading an eBPF program, BTF metadata is checked by the eBPF verifier and it is rejected if inconsistencies are found.

In order to use maps they need to first be first created, and this can be done using the `bpf()` system call, with the appropriate parameters. However, it is usually performed automatically using the libbpf library in the user space program, that will automatically load the map before the verification step. At this point user space program and kernel space program can use it, retrieving and writing data into it. The kernel space program will use eBPF helper functions like `bpf_map_lookup_elem()`, `bpf_map_update_elem()`, `bpf_map_delete_elem()`, which correct use is then checked by the eBPF verifier. The user space program will use instead the `bpf()` system call, with the `BPF_MAP_LOOKUP_ELEM` command, the `BPF_MAP_UPDATE_ELEM` command, or the `BPF_MAP_DELETE_ELEM` command, but usually libraries like `libbpf` and BCC are used in common applications.

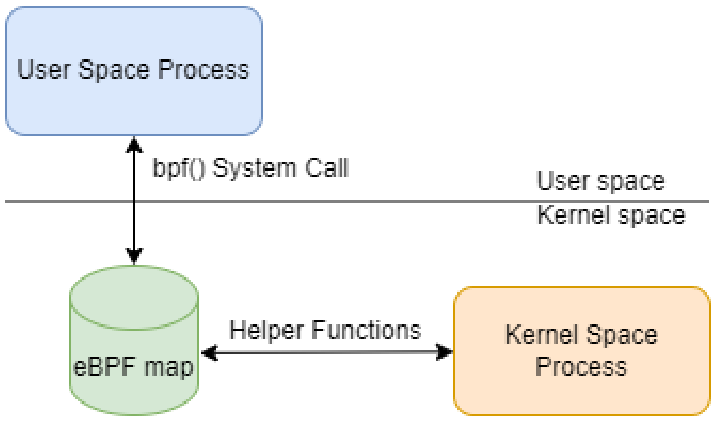


Figure 2.1: User Space/Kernel Space interaction.

2.2 Compilation process

In order to understand where the verifier performs its tasks, we are going to describe the compilation process of a simple eBPF program that has no user space program, where the user manually performs the loading, attachment, detachment and unloading steps using the libbpf command line tool called bpftool. Those can also be performed in the user space using the libbpf library.

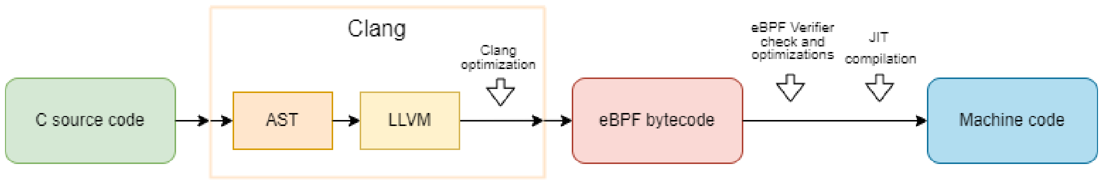


Figure 2.2: The compiling pipeline.

The eBPF program written in C is usually compiled with LLVM's project Clang, an open source compiler for the C language. The Clang compiler acts as a frontend for the LLVM compiler backend. In particular, Clang compiles the C code to an Abstract sysntax tree, that describes the syntax of the source code, and then to the LLVM intermediate representation, which is then transformed in the target representation. LLVM is a language independent intermediate representation, used as a first step of compilation by C, but also Rust compilers. It is a high level assembly language, containing information about the C (or Rust) code, but still resembling the low level machine instructions. It is then optimized by the compiler and translated to the target machine, that, in case of the compilation of a classic C or Rust file run in the user space, would be the architecture of the machine it is performed on. In the case of eBPF kernel space programs, we add the option "-target bpf", that indicates the compiler to use as the target machine language the eBPF bytecode.

The eBPF bytecode is an assembly language with the features of the most common RISC instruction sets. It has 10 registers, as well as the load, store, jump and the common ALU operations. The execution of the eBPF bytecode is then demanded to the eBPF virtual machine. In the earlier version of the kernel where eBPF was introduced, the eBPF virtual machine translated runtime the bytecode to the native machine code, with relevant performance issues. Later, with the introduction of JIT compilation, the program is translated to native machine code just once, during loading time. Older architectures still cannot use the JIT compilation.

The eBPF bytecode is stored by the Clang compiler in the object output file `.o`. It can be inspected through the usage of the `llvm-dump` tool. This file is the one that is going to be loaded into the kernel. In order to load an eBPF program, the kernel exposes a system call called `bpf()`, that is the user hook to manage the eBPF program lifecycle. Usually the final user does not directly use the `bpf()` system call, but instead uses the libbpf library, and if no user space is used, the libbpf's command line tool bpftool.

During the loading different steps are performed, but the main one is the verification and optimization of the code. In particular, not only the static analysis of the eBPF bytecode is executed, but also the check of the BTF structures. BTF allows eBPF programs to be portable across different kernel versions so that it's possible to compile a program on one machine and use it on another, which might be using a different kernel version and so have different kernel data structures. BTF also contains information about the structures introduced with maps and kernel functions. The main task is however the static analysis performed by the eBPF verifier, in order to see if the program may cause any system fault or security issue. In the end of the loading process trhe program is JIT compiled to the local machine code, in order to increase the eBPF program performances.

When a program is loaded, it can be attached to an event that will trigger it. The event is strictly related to the type of the program, and also determines the type of the context structure passed in the main function. When a program is attached to an event it can execute. To stop it, the program must be detached from the event, and then, if needed, it can also be unloaded from the kernel.

2.3 The eBPF verifier

The eBPF verifier is the backbone upon which the technology managed to be widely adopted in production applications. The checks made during the loading phase allow the developers not to worry about the eBPF program possibly inserting error that could stop the kernel, and so the entire operating system, or about security issues in such a delicate part of the application, since the degree of privileges.

2.3.1 The verification process

The eBPF verifier is a static analysis tool, which means it can analyze the code without the need to execute it. It's important to remember that the code being analyzed is the eBPF bytecode. The choice to analyze bytecode rather than source code stems from the fact that the source code can be compiled in different ways. Additionally, bytecode is the form that is actually interpreted (not compiled) for execution, so it makes sense to analyze it, as it is what will ultimately run.

To perform the static analysis, the verifier processes the list of eBPF instructions provided in the object file through the `bpf()` system call and generates, based on logical branches, all possible execution paths of the given code. For example, in the case of an if statement, at least two possible execution paths are generated: one where the condition holds true and one where it doesn't. As a result, the number of execution paths increases exponentially with the number of branches, and thus with the size of the code. To handle this, the verifier is highly optimized for its specific task, applying pruning techniques when equivalent execution paths are identified, meaning these only need to be checked once.

Once all possible execution paths have been generated, the verifier checks, for each path, how the eBPF registers (which roughly correspond to C code variables, with the exception of spilled registers on the stack) interact with each other and the nature of the data they contain. This interaction is tracked through a structure called bpf_reg_state, which holds information about the inferred type of the register, necessary to ensure consistency in function calls or when performing operations like dereferencing, and an estimation of the value it holds. Based on this information, the verifier then proceeds to scan through the instructions to identify any cases where an instruction might cause a kernel crash, information leakage, or other security issues.

2.3.2 Register State

Register Type

The verifier tracks the state of each register at a given instruction through the `bpf_reg_state` structure. This structure contains essential information needed to predict potential faults in the code, such as the type and nature of the data held by the register. There are three fundamental types of registers:

- **Uninitialized registers**, which should not hold any information.
- **Scalar registers**, which hold a value that doen't rapresent a pointer.
- **Pointers**, which refer to various data structures in the eBPF environment. These include the context (the value passed as a parameter to the main function), packets (in network applications), or pointers to map keys and values, which are necessary to interact with those data structures.

These pointer types may also be marked as *read-only* or *nullable*, meaning the verifier tracks whether a pointer can be null (and thus flag a potential dereference error). Tracking the type of each register is necessary for validating all operations performed on them, especially when passed to helper functions or kernel functions that have specific signatures requiring particular data types, or when common operations like dereferencing are executed.

Value Tracking

It is equally important to track the potential value of registers, as this helps predict whether a memory access is out-of-bounds. The verifier includes a dedicated function, `check_mem_access`, which checks for every pointer type whether any dereference falls within legal memory limits. This is done through two methodologies:

1. **Tracking minimum and maximum values**: This approach considers, for each register, the minimum and maximum values it can hold. These values are tracked for both 64-bit and 32-bit registers, as well as for signed and unsigned operations, resulting in a total of eight fields that define the register's boundaries and how memory access should be handled.

2. **Variable offset with tristate numbers**: To reduce cases where the verifier wrongly rejects registers with valid values, an auxiliary structure called *variable offset*, based on tristate numbers, was introduced. This abstraction of numbers, implemented by default in the Linux kernel, tracks the possible values of each bit in a register based on previous operations. For example, if an `if` statement checks whether a register equals 2 or 3 (binary `10` or `11`), the variable offset for that register would mark the least significant bit (LSB) as *unknown* (since it can be either 0 or 1) and the next most significant bit as *1* (since it's always 1 in both cases). This allows for more precise control over the registers, enhancing both developer experience and eBPF program performance while preserving correctness.

Both the minimum/maximum values and the variable offset are updated during ALU operations, ensuring that the information is propagated as long as it is needed. This approach improves the accuracy of register tracking and ensures that programs can be verified more efficiently without rejecting valid code unnecessarily.

2.3.3 Verification criteria

Validating Helper Functions

In eBPF, direct calls to kernel functions are not allowed unless the function is registered as a `kfunc`. Instead, eBPF programs can use a set of predefined helper functions to interact with kernel data structures and retrieve information. These helper functions are specific to different types of eBPF programs. If a helper function is called from an incompatible eBPF program type, the verifier will raise an error indicating that the function is "unknown" for that particular program type.

Helper Function Arguments

Each helper function in eBPF is defined with a specific function prototype structure, which defines, for each argument, the register type that can be accepted. The verifier uses this structure to ensure that the correct argument types are passed to the helper function. So if the verifier detects a function call where the arguments passed have a register type different from the one specified in the helper structure, the verifier will raise an error.

License Verification

The eBPF verifier ensures that programs using GPL-licensed helper functions are also licensed under GPL-compatible terms. If an eBPF program calls a helper function that is restricted by GPL, and the program does not specify a compatible license, the verifier will produce an error. For instance, if a program uses the `bpf_probe_read_kernel()` helper, which is GPL-restricted, the verifier will return an error if the program lacks a proper license. To resolve this, the eBPF program must include a license declaration in the format `Dual BSD/GPL`, which ensures compliance with the required licensing conditions.

Invalid Memory Access

One of the most common errors encountered during eBPF program verification is related to invalid memory access, particularly when interacting with maps or arrays. The verifier tracks all memory accesses and ensures that indices remain within valid bounds. For instance, when accessing an array of length 12, any attempt to use an index equal to or greater than 12 will be flagged as an out-of-bounds memory access.

For example an eBPF program attempted to access an array named `message` using an index stored in register `R1`. The verifier traced the register's potential values and determined that the index could exceed the valid range (0–11), triggering an "invalid memory access" error. This issue often happens when programmers mistakenly use `<= sizeof(message)` instead of `< sizeof(message)`. Since global variables such as arrays are implemented as maps in eBPF, any incorrect access will be caught by the verifier, preventing the program from being loaded into the kernel.

Pointer Dereferencing and Null Checks

The verifier checks if the dereferencing of pointers can happen without causing any fauls. In eBPF, pointers returned by helper functions or obtained from maps can be null, and any attempt to dereference a null pointer without checking if not null first will lead to a verifier error.

For example the function `bpf_map_lookup_elem()` returns a pointer to an element in the map. If the element does not exist, the pointer will be null. Dereferencing this null pointer without a prior check results in an error, such as *invalid memory access 'map_value_or_null'*. The correct approach is to first verify that the pointer is non-null before attempting to dereference it. In some cases, helper functions like `bpf_probe_read_kernel()` can safely handle null pointers by taking unsafe pointers as arguments and performing necessary checks internally.

Context and BPF Helper Functions

Each eBPF program is passed a specific context (e.g., `xdp_md`), though not all fields within this context are accessible. For example, tracepoint programs can only access fields specific to the tracepoint, and attempting to access other fields results in an "invalid context access" error. eBPF programs frequently interact with the kernel through a set of helper functions, such as `bpf_printk()` and `bpf_map_lookup_elem()`. These helpers facilitate communication between the program and the kernel, while enforcing safety checks on memory access and pointer validation.

Ensuring Program Termination

The verifier ensures that eBPF programs terminate properly. Infinite loops in eBPF would cause the kernel to hang, so the verifier enforces a strict limit of one million instructions per execution. Any program exceeding this limit fails verification.

Handling Loops in eBPF

Loop Restrictions Before Kernel 5.3 Prior to Linux kernel version 5.3, backward jumps were prohibited, which made loops infeasible in eBPF. Developers resorted to the `#pragma unroll` directive to instruct the compiler to manually unroll loops, simulating repetitive logic without actual jumps.

Loops in Kernel 5.3 and Later From kernel version 5.3 onward, the verifier began supporting certain types of loops, following both forward and backward branches, provided the total number of instructions remains under the one-million threshold. For example, a loop with a fixed and hard coded number of iterations is allowed. In kernel version 5.17, a new helper function `bpf_loop()` was introduced, in order to loops in eBPF programs. This helper takes a function and the number of iterations to execute, allowing the verifier to validate the function's instructions once and reuse them for each iteration. Additionally, the `bpf_for_each_map_elem()` helper enables more efficient and verifier-friendly iteration over map elements.

Uninitialized Return Values

A frequent issue that leads to verifier errors involves uninitialized return values. Every eBPF program must return a valid code, stored in register `R0`. If `R0` remains uninitialized, the verifier rejects the program, producing errors like `R0 !read_ok`. Adding a return statement or using a helper function like `bpf_printk()` (which initializes `R0`) resolves the issue.

Invalid Bytecode Instructions

The verifier checks for invalid bytecode instructions that could lead to kernel instability. Although modern compilers generally do not emit invalid instructions, manually writing eBPF bytecode or using newer atomic operations on an outdated kernel can trigger such errors.

Unreachable Instructions

The verifier also rejects unreachable instructions, i.e., code that cannot be executed. While unreachable code is often optimized out by the compiler, if it remains in the program, it will be flagged as an error by the verifier.

Program Size Limit

The eBPF verifier enforces a strict limit on the size of the program, rejecting any program that exceeds 1 million bytecode instructions. This constraint is in place to prevent excessively large programs that could overwhelm system resources or lead to performance issues. Most eBPF programs are under this limit, but complex or improperly written programs that generate excessive instructions, either through unrolled loops with excessive dimensions, can trigger this error. The eBPF verifier counts every instruction during its analysis, and if the total number surpasses the maximum allowed, the program is rejected.

Chapter 3

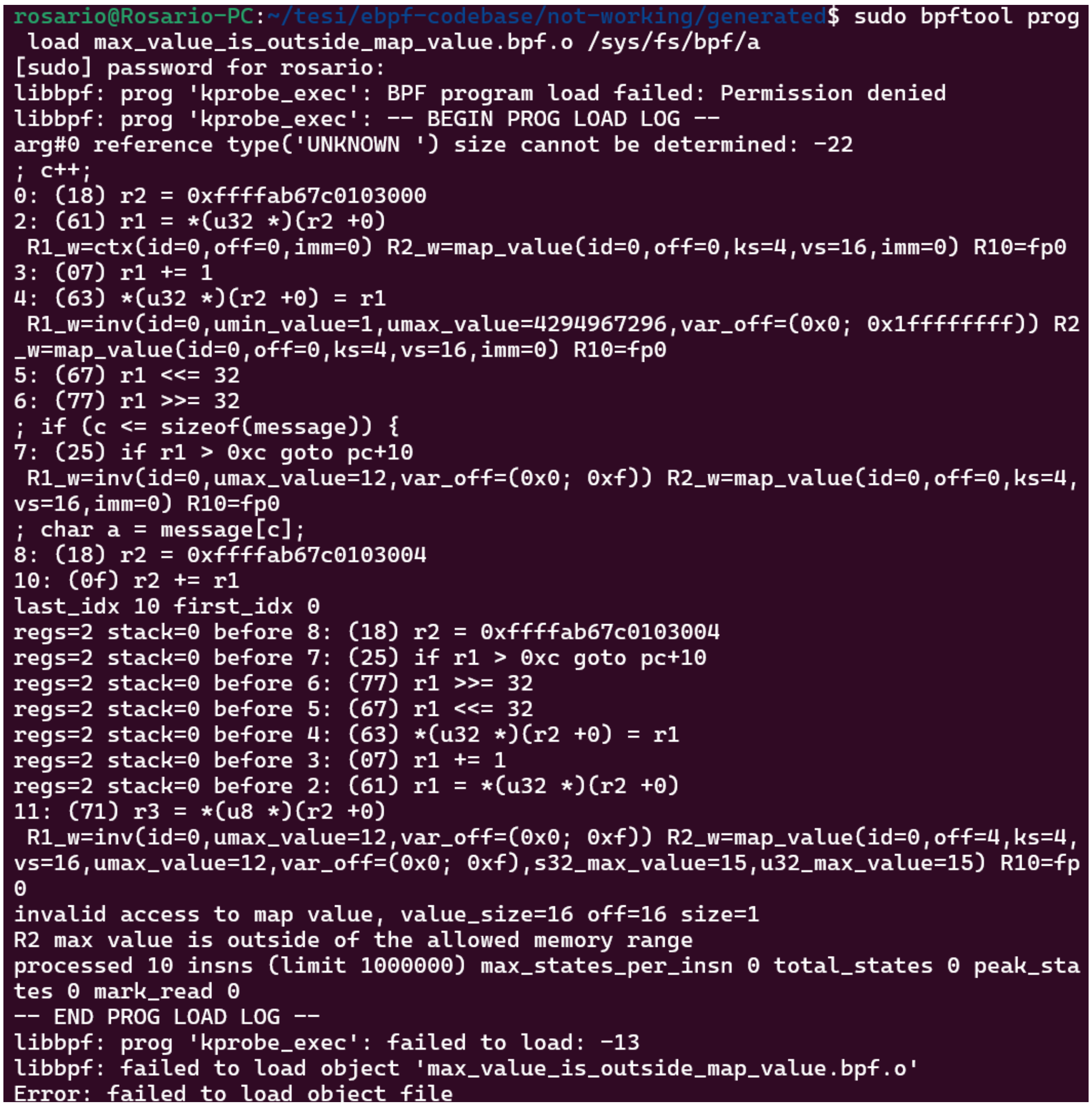
Improving the eBPF verifier errors readability

During the loading of a program and its subsequent verification, various levels of logging may be presented based on the selected option. If a potential flaw is found in the code, the verifier produces a specific log, indicating the type of error occurring and the eBPF bytecode instruction where it happens. If debug mode is enabled during compilation with Clang using the "-g" option to generate the object file containing the eBPF bytecode instructions, it is also possible to reference the C code from which the corresponding instructions were generated. However, in this context, the presented error is often complex to interpret, specifically referring to eBPF bytecode and eBPF registers, rather than the variables of the original C code, and does not indicate the location within the C source file where the line of C code that is shown is located (Figure 3.1). The goal of this thesis is to improve the developer experience by providing more solid references to the original source code from which the error was generated and potentially making the error message issued by the verifier for that specific case more understandable.

3.1 Choosing the Context

3.1.1 Kernel version

The development of the static analysis tool will be based on the Linux kernel version 6.6, which was the Long Term Support (LTS) version at the time this thesis work began. The choice of an LTS version is driven by the need for a stable and long-term supported environment, which minimizes disruptions caused by frequent kernel updates and provides a solid foundation for the tool's functionality. As LTS versions are maintained with security patches and bug fixes over several years, this



```
rosario@Rosario-PC:~/tesi/ebpf-codebase/not-working/generated$ sudo bpftool prog
 load max_value_is_outside_map_value.bpf.o /sys/fs/bpf/a
[sudo] password for rosario:
libbpf: prog 'kprobe_exec': BPF program load failed: Permission denied
libbpf: prog 'kprobe_exec': -- BEGIN PROG LOAD LOG --
arg#0 reference type('UNKNOWN ') size cannot be determined: -22
; c++;
0: (18) r2 = 0xffffab67c0103000
2: (61) r1 = *(u32 *)(r2 +0)
 R1_w=ctx(id=0,off=0,imm=0) R2_w=map_value(id=0,off=0,ks=4,vs=16,imm=0) R10=fp0
3: (07) r1 += 1
4: (63) *(u32 *)(r2 +0) = r1
 R1_w=inv(id=0,umin_value=1,umax_value=4294967296,var_off=(0x0; 0x1ffffffff)) R2
_w=map_value(id=0,off=0,ks=4,vs=16,imm=0) R10=fp0
5: (67) r1 <<= 32
6: (77) r1 >>= 32
; if (c <= sizeof(message)) {
7: (25) if r1 > 0xc goto pc+10
 R1_w=inv(id=0,umax_value=12,var_off=(0x0; 0xf)) R2_w=map_value(id=0,off=0,ks=4,
vs=16,imm=0) R10=fp0
; char a = message[c];
8: (18) r2 = 0xffffab67c0103004
10: (0f) r2 += r1
last_idx 10 first_idx 0
regs=2 stack=0 before 8: (18) r2 = 0xffffab67c0103004
regs=2 stack=0 before 7: (25) if r1 > 0xc goto pc+10
regs=2 stack=0 before 6: (77) r1 >>= 32
regs=2 stack=0 before 5: (67) r1 <<= 32
regs=2 stack=0 before 4: (63) *(u32 *)(r2 +0) = r1
regs=2 stack=0 before 3: (07) r1 += 1
regs=2 stack=0 before 2: (61) r1 = *(u32 *)(r2 +0)
11: (71) r3 = *(u8 *)(r2 +0)
 R1_w=inv(id=0,umax_value=12,var_off=(0x0; 0xf)) R2_w=map_value(id=0,off=4,ks=4,
vs=16,umax_value=12,var_off=(0x0; 0xf),s32_max_value=15,u32_max_value=15) R10=fp
0
invalid access to map value, value_size=16 off=16 size=1
R2 max value is outside of the allowed memory range
processed 10 insns (limit 1000000) max_states_per_insn 0 total_states 0 peak_sta
tes 0 mark_read 0
-- END PROG LOAD LOG --
libbpf: prog 'kprobe_exec': failed to load: -13
libbpf: failed to load object 'max_value_is_outside_map_value.bpf.o'
Error: failed to load object file
```

Figure 3.1: The eBPF verifier log: the lines of the source C code starts with ';'.

ensures that the tool will remain compatible with widely-used systems during its initial development phase.

However, since the kernel evolves rapidly, and with version 6.8 already being adopted as the new LTS version by distributions such as Debian, future work will need to consider porting the tool to newer kernels. It's a given that the the flexibility to adapt to newer kernel versions is an essential requirement for the project's long-term success.

3.1.2 Compiler

For the compilation of eBPF programs, Clang with the `-target bpf` option will be used, as it is currently the preferred and most widely-supported compiler for generating eBPF bytecode. Clang has deep integration with the LLVM project, providing significant advantages for eBPF development, such as the ability to produce highly optimized bytecode and extensive debugging capabilities when combined with the `-g` option. It is important that the compilation of eBPF programs is performed using the highest optimization level available in the compiler, particularly Clang. eBPF programs often need to be highly optimized due to their constrained execution environment within the kernel, where performance and efficiency are critical. This is why it is common to compile eBPF programs with the `-O3` optimization flag, which enables aggressive optimizations, such as loop unrolling, inlining, and vectorization, to ensure the most efficient bytecode possible. These optimizations are crucial because even small performance improvements can have a significant impact on the overall performance of kernel-level programs.

Clang offers multiple levels of optimization [10], which range from `-O0` (no optimization) to `-O3` (maximum optimization). The optimization levels are as follows:

- `-O0`: No optimization. The compiler does not perform any optimization on the code, which can make debugging easier but results in larger and slower executables.

- `-O1`: Basic optimization. This level focuses on improving performance without significantly increasing compile time or complexity. It performs optimizations like eliminating unreachable code and simplifying expressions.

- `-O2`: Moderate optimization. This level balances between speed and compile time, performing a broader range of optimizations such as more aggressive inlining and code reordering.

- `-O3`: Maximum optimization. This level applies the most aggressive optimizations, including advanced techniques like loop unrolling and vectorization. It aims to produce the fastest possible executable, though it may increase compilation time and binary size.

Given the nature of eBPF programs, which often need to run as efficiently as possible, the tool must be resilient to issues that arise from these aggressive optimizations. It must ensure that the error messages and debugging information remain clear and useful, even when the compiler has transformed the original source code into highly optimized bytecode.

GCC

That being said, other compilers are also making strides in supporting eBPF. Recently, GCC has started to include a BPF backend [11], allowing the generation of eBPF bytecode directly from C source code. As of now, GCC's support for BPF is still evolving, and it might not yet match Clang in terms of tooling and ecosystem integration. However, this could be a potential avenue for future work, especially if demand for cross-compiler compatibility rises.

3.1.3 Programming Language Support

The initial implementation of the static analysis tool will focus on parsing C code, as C is the predominant language used to write eBPF programs. However, future iterations of the tool could expand to support additional programming languages that are increasingly being used to generate eBPF bytecode. For example, Rust is gaining traction as an alternative to C for systems programming, due to its strong emphasis on memory safety and concurrency. The Linux kernel has already begun integrating Rust for some subsystems, and the potential for Rust to be used in eBPF programs is being actively explored. The Rust community has developed numerous crates that allow developers to use Rust for their eBPF projects [12]. Supporting multiple programming languages would significantly broaden the tool's applicability, making it more versatile and future-proof as the eBPF ecosystem continues to evolve.

3.2 Approaches

During the development of this thesis, two main approaches were identified to address the problem of improving error feedback from the eBPF verifier.

3.2.1 Clang Static Analyzer

The first approach considered was the use of Clang's static analyzer, a tool for checking C source code for potential errors integrated in the Clang and LLVM environment. The idea was to develop specific checks to partially replicate the behavior of the eBPF verifier, but operating on the C source code before the compilation into eBPF bytecode. This would enable the detection of errors typically caught by the verifier without needing to load the actual eBPF code.

Development Pipeline

One of the main advantages of the Clang static analyzer approach is that it could be integrated into the developer's regular workflow, creating a development pipeline that allows for early detection of potential issues. In this pipeline, the developer would write the C code as usual, and before proceeding to compile it into eBPF bytecode, they could run the static analyzer. The analyzer would check the C code for errors or potential problems that might later be caught by the eBPF verifier during runtime. This process would act as a first line of defense, offering the developer the opportunity to identify potential issues before compiling and loading the program into the kernel. The Clang static analyzer could catch errors at an earlier stage in development, potentially saving time and effort in debugging and re-compiling eBPF bytecode. However, this does not mean that the results of the static analyzer would be considered definitive. Given the nature of static analysis, there is a possibility of false negatives—i.e., the analyzer could miss some errors that the eBPF verifier would eventually catch. Therefore, the developer would still need to run the eBPF program through the verifier, as this final check ensures that the bytecode is valid and safe for execution within the kernel. Additionally, static analysis can also lead to false positives—situations where the analyzer incorrectly flags an error that does not actually exist in the code. This can result in the developer spending unnecessary time trying to fix a non-existent problem, thus delaying the development process. In such cases, the program may appear to have an issue, but the error could be a misinterpretation of the static analyzer, leading to wasted debugging efforts. Therefore, while the static analyzer serves as a useful tool, its findings must be taken with caution, and further verification by the eBPF verifier remains crucial. In figure 3.2 is described the development pipeline.

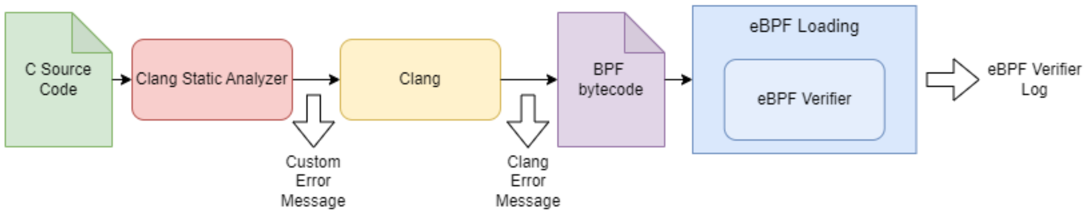


Figure 3.2: Clang Static Analyzer Approach

False positive and false negatives

As illustrated earlier, the static analyzer approach would lead to the presence of **false positive**, situations in which the static analyzer finds issues in the C source code, but the eBPF verifier doesn't, and **false negatives**, where the static analyzer doesn't identify error in the C source code, while the eBPF bytecode does. The reasons behind this event lay on the nature of the pipeline: having multiple source of truth (in this case the C static analyzer and the eBPF verifier) may lead to discordance between the two, forcing the developer in a state of distrust of the tool that should be developed. Moreover, due to the nature of the task, this cases may happen frequently. There are different reasons behind this:

1. **Compilation process**: The main reason for false positives and false negatives lies in the different domains of the two analyzers: the C static analyzer works on C code, while the eBPF verifier operates on eBPF bytecode. Even though the bytecode is generated from the C code, the compilation process adds additional information about register usage, such as spilling[1] or applies optimizations that may render some checks performed by the static analyzer irrelevant, for example, checks for division by zero or dead code, which are handled by the compiler, are often resolved before the bytecode is analyzed by the eBPF verifier.

2. **Complexity of the Verifier**: The eBPF verifier is a highly sophisticated and optimized piece of software designed to meet the specific requirements of verifying eBPF programs. It takes into account a wide range of factors, tracking register types, possible values, and other key parameters. Attempting to replicate many of the common checks performed by the verifier would require developing extremely large and complex static analysis checks. Some of these checks might even be impossible to implement with a static analyzer. The eBPF verifier is a standalone program with full access to both the analysis variables and its execution flow, whereas the Clang static analyzer is a static analysis framework that can face limitations in recreating certain checks. As a result, some checks may not be fully emulated by the static analyzer, leading to potential false negatives.

3. **Performance Issues**: The process of static analysis is inherently exponential with respect to the lines of code, as each decision in a line may lead to numerous paths that need to be evaluated. Significant efforts have been made to improve the performance of the eBPF verifier through techniques like state pruning and the use of specific helper functions [5]. Attempting to emulate the verifier using a framework not specifically designed for this task can lead to severe performance issues that may be difficult to resolve, especially when the Clang static analyzer lacks complete control over the analysis process. Similar to the situation where the complexity of the check becomes too high, this can result in the inability to emulate certain checks, potentially leading to false negatives.

4. **Extensiveness of the Verifier** The eBPF verifier is a very large code base accounting for a wide range of errors. In Appendix A there is a list of all the

[1]Spilling is the practice of moving register values to the stack (RAM memory) when there are not enough registers to hold all the variables in use [13]

errors that the eBPF verifier emits, and most of them are not directly mapped to the high level description of the checks it performes, since it goes deeper into some aspect of the error, usually bounded to the eBPF bytecode syntax. Emulating the eBPF verifier in such detail to account for all the errors it emits would likely be extremely difficult, leading to potential false negatives.

3.2.2 Message Error Analysis

Given the challenges of static analysis, a different approach was taken—using the error messages generated by the verifier itself during program loading. By analyzing these messages and correlating them with the C source code, it was possible to provide more meaningful error feedback to the developer.

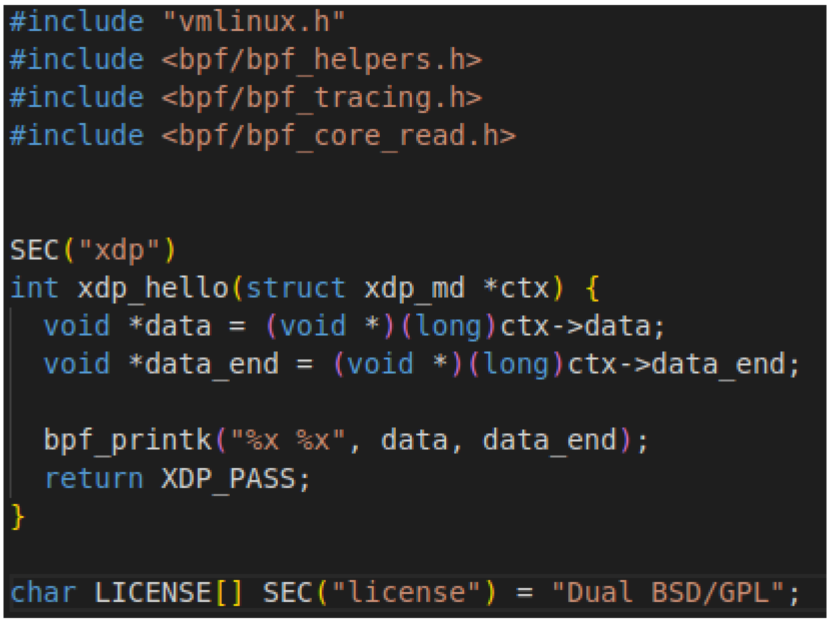
Approach Details

In this approach, the focus is on intercepting and analyzing the stderr output generated by the `bpftool load` command during program loading or the output given by the equivalent `load` operation performed by libbpf in the C user space program. Using the `-g` option during the compilation, the resulting `.o` eBPF bytecode file retains the information about the C line responsible for the generation of the bytecode lines. This information is then shown in the eBPF verifier log in case an error is found (or if debug mode is active in the libbpf command) 3.1. It is possible to retrieve the bytecode (and the relative C lines) using the `llvm-objdump -S` utility. In the figure 3.3 a simple eBPF program is shown. Using the `-g` option in the compilation process, we obtain a `test.bpf.o` file containing the C lines of the source C file, commented with the ';', as shown in the figure 3.4.

Using this feature provided by the Clang compiler, it is possible to use the C line information in order to give the developer an error message possibly similar to the one that would have been give using the static analyzer approach. Through the parsing of the output much information can be given to help the developer solve the issue, like suggestions, improvement of readability, error and other enhancements like adding the line number to the C line referenced in the error log, emulating the result given by the Clang static analyzer. In figure 3.5 is showed the development pipeline.

Error messages

In order to implement this approach, it is necessary a deep study of the eBPF verifier code, looking for all the possible output it may give. The eBPF verifier emits numerous error messages, logs, and warnings, which are all printed through the `verbose` function and its wrappers. Other functions used to print messages in the output of the eBPF verifier are the `bpf_log` function, used for logging purpose,



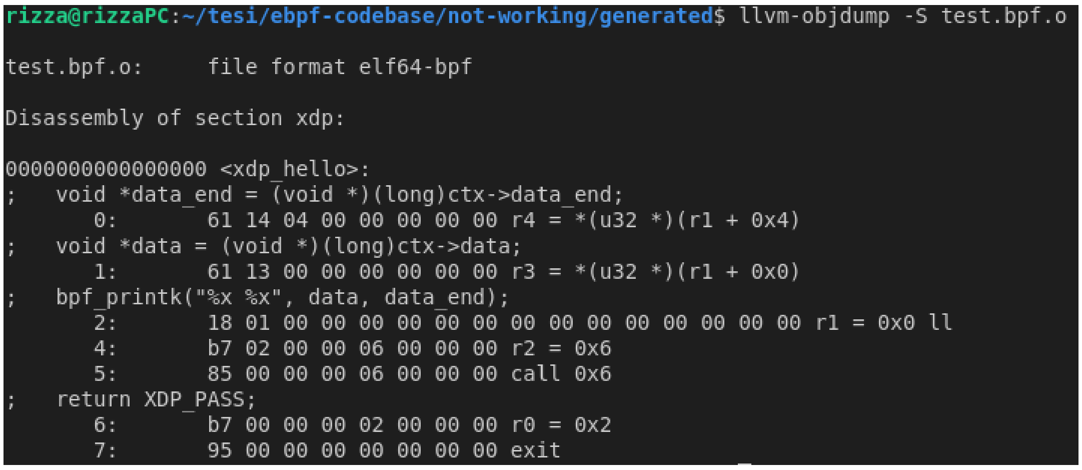
```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>


SEC("xdp")
int xdp_hello(struct xdp_md *ctx) {
  void *data = (void *)(long)ctx->data;
  void *data_end = (void *)(long)ctx->data_end;

  bpf_printk("%x %x", data, data_end);
  return XDP_PASS;
}

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

Figure 3.3: A simple eBPF test program



```
rizza@rizzaPC:~/tesi/ebpf-codebase/not-working/generated$ llvm-objdump -S test.bpf.o

test.bpf.o:     file format elf64-bpf

Disassembly of section xdp:

0000000000000000 <xdp_hello>:
;   void *data_end = (void *)(long)ctx->data_end;
       0:       61 14 04 00 00 00 00 00 r4 = *(u32 *)(r1 + 0x4)
;   void *data = (void *)(long)ctx->data;
       1:       61 13 00 00 00 00 00 00 r3 = *(u32 *)(r1 + 0x0)
;   bpf_printk("%x %x", data, data_end);
       2:       18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0x0 ll
       4:       b7 02 00 00 06 00 00 00 r2 = 0x6
       5:       85 00 00 00 06 00 00 00 call 0x6
;   return XDP_PASS;
       6:       b7 00 00 00 02 00 00 00 r0 = 0x2
       7:       95 00 00 00 00 00 00 00 exit
```

Figure 3.4: The llvm-objdump utility allow to see the disassembled code

and the `WARN_ONCE` function, used in context where just a warn message is needed. These two functions however do not play a role in the log printing when an error is found in the code, therefore they can be ignored. The total number of calls to the `verbose` function for the eBPF verifier in the Linux kernel version 6.6 LTS exceeds 500. However, this number is not indicative of the potential error messages encountered by the verifier. Several cases have been identified where calls to the function do not correspond to actual errors found in the code.

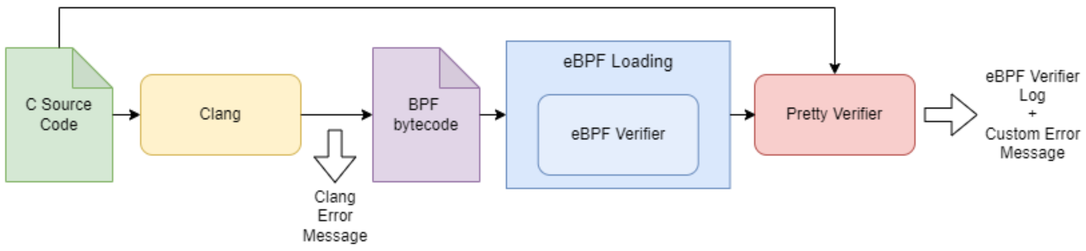


Figure 3.5: Message Error Analysis Approach

Logging with `verbose` The `verbose` function is used to log the verifier's state. Depending on the execution mode, which is selectable with the `bpf()` system call, logs about the verifier's progress may be produced. For instance, the function `print_verifier_state` displays the verifier's internal state at the time of the call. This function is typically invoked in cases of internal errors, although a logging level can be set to force its call for each instruction. These types of calls to the `verbose` function are not relevant for understanding the type of faults found in the code, as they do not represent error messages. A total of 54 log messages were found using the `verbose` function.

Internal Errors The `verbose` function is also used to notify internal errors in the verifier. During the verifier's development, the developers inserted unconditional exits in situations that, at the time of design, were considered impossible. These exits are usually accompanied by an error message starting with `"verifier internal error:"` or `"BUG"` to flag faults in the verifier's own code. However, there are cases where this formatting is not respected. These errors are not relevant for interpreting eBPF code errors, as their occurrence would indicate issues in the eBPF verifier code itself or problems that require investigation by the kernel developers. A total of 97 verifier error messages were found using the `verbose` function

Errors not related to the C code

Among the remaining cases, which are caused by errors in the bytecode passed to the verifier, there are two situations where the error is not directly caused by the original C source code. Therefore, improving the message error readability with respect to the C code would not significantly aid in resolving the issue.

Errors designed exclusively for eBPF code : Many error messages that can be emitted by the verifier are exclusively triggered by writing incorrect eBPF bytecode manually or when the compiler translates LLVM code from the original C source into invalid eBPF bytecode. In these situations, explaining the error would not significantly help the developer in resolving it. A total of 123 errors message of this type were found.

Errors caused by BTF and system configurations BTF (BPF Type Format) is an abstraction used by the framework to manage external data structures and functions in eBPF code, ensuring security (for eBPF map structures) and compatibility (for kernel structure references, which might not exist if hardcoded and used on a machine with a different kernel version or configuration). These types of errors are usually not resolvable by modifying the C source code, and they are often not directly related to it. Similarly, errors related to missing kernel privileges (CAP) have been left unhandled. A total of 78 BTF messages were found using the `verbose` function

Rare Errors

Lastly, many errors are extremely rare, and limited documentation exists online. A Google search often results in just a few references, most of which point to the kernel pull request that introduced the message. These errors have been left for future work due to the difficulty in testing the tool in scenarios where such errors are triggered, especially given the complexity of generating C code that would cause them. The total of message errors deriving this type summed to the ones that are caused by configuration errors of the OS are 98.

3.2.3 Pro and Contra of the Two Approaches

The second approach addresses many of the issues encountered with the first one, primarily by eliminating the risk of false positives and false negatives. This is because the source of truth is reduced to just one — the eBPF verifier itself. Additionally, the message parsing approach is less dependent on external tools like the Clang static analyzer, making it more robust to changes in those tools and potentially reusable for other programming languages, such as Rust.

However, this approach is not without its drawbacks. It is closely tied to changes in the Linux kernel, although these changes tend to be relatively minor and manageable. Another downside is that this method operates through the command line, which makes it less user-friendly compared to other tool, that might have been integrated directly into an IDE or editor. Moreover, while it offers reliable error information, the amount of information provided is limited, and in certain cases, the error message might be less detailed than desired, though generally still sufficient for debugging. It's also important to note an important upside present in the Clang static analyzer approach that could not be emulated in the message analysis approach, that is the possible use of this tool as a defense mechanism towards code fragments exploiting eBPF verifier vulnerabilities and managing to compromise the kernel security. Considering the high number of CVE reported, it might be a relevant downside, since the message analysis approach fully relies on the eBPF verifer functionalities, hence it cannot be used to prevent these scenarios.

To provide a clearer comparison, the table 3.1 summarizes the pros and cons of each approach.

3.2.4 Study of the eBPF Verifier

The process of reviewing the eBPF verifier code is inherently complex but can be quantified and predicted, making it the chosen approach. In Appendix A, the results of this process are presented, listing all the possible errors the eBPF verifier can produce, along with a label categorizing each type of error. The analysis was carried out by examining where and why each error is triggered, understanding the eBPF verifier code, and supplemented by a Google search to determine whether the issue had been encountered by other developers in forums or eBPF repositories.However, it was found that most of these errors are uncommon, and only a few have been reported multiple times by the online community. This follows a pattern describable by the Zipf's law, where a small number of errors are encountered most frequently, while the majority occur very rarely or are seldom experienced. As a result, numerous error messages were left unhandled, even though all those with evidence of occurrence and relevance to the C code have been addressed, even if they could not be reproduced locally. The chapter 2 describes the tool developed to interact with this output and the error management mechanisms used to handle the eBPF verifier error log.

Approach	**Pros**	**Cons**
Clang Static Analyzer	• Offers the ability to catch errors pre-compilation • Has a deeper control on the source code when the error occurs • Not particularly affected by small eBPF verifier changes • Can be used to intercept code that exploits newly discovered eBPF verifier vulnerabilities	• Prone to false positives and false negatives due to differences between C and eBPF bytecode • Unable to fully match the complexity and depth of the eBPF verifier's checks • Performance overhead due to trying to emulate the verifier's complex behavior • Hard to manage all the errors that the eBPF verifier may find
Message Error Analysis	• Directly relies on actual error messages from the eBPF verifier, ensuring accuracy • Less dependent on external tools like Clang, making it adaptable and robust • Could be extended for use in other languages, such as Rust	• Closely tied to kernel changes, although usually minor • Limited by command-line interaction, making it less user-friendly • Provides less detailed information in certain edge cases • Cannot add a security layer to the eBPF programming experience

Table 3.1: Comparison of Clang Static Analyzer and Message Error Analysis Approaches

Chapter 4

Pretty Verifier

4.1 Introduction

To handle the eBPF verifier errors, a Python application was developed that utilizes the output from the eBPF verifier to return enhanced error messages with supplementary information. This tool serves to bridge the gap between the complex verification output and developer understanding, making debugging eBPF programs more accessible.

The complexity of eBPF programs often leads to complex interactions within the kernel, resulting in various verification errors. Traditional error messages from the eBPF verifier can be cryptic and difficult to interpret, making it challenging for developers to pinpoint the exact cause of an error. The goal of this application is to provide more meaningful and context-rich error messages, simplifying the debugging process.

As illustrated in the chapter 3, the developement of this tool followed the extensive study of the eBPF verifier, that produced the Appendix A, containing all the possible error messages from the eBPF verifier, and how they were classified, if they were handled by the tool, a description of the error and how it was managed, and whether it was possible to test it.

4.2 Usage

The Pretty Verifier tool was specifically developed to assist developers working with eBPF programs in the Linux kernel, particularly targeting the error messages generated by kernel version 6.6. This version was the Long Term Support (LTS) at the beginning of the thesis; however, kernel version 6.8 later became the new LTS and was subsequently adopted by the Debian distribution.

The tool relies on the following components:

- **Python3:** The python 3.10 runtime used to run the Pretty Verifier scripts.
- **Clang Compiler:** Essential for compiling the eBPF programs. The use of other compiler may be deepen in the future.
- **libbpf Library:** A C library that manages the life cycle of eBPF in user-space applications.
- **bpftool:** A command-line utility that provides an interface using the libbpf functionalities in bash

For the application to function correctly, it must be compiled with the `-g` option, which enables the inclusion of original source line information. Testing has demonstrated that using the highest optimization option in the Clang compiler does not adversely affect the operation of the tool.

The application can be invoked from the command line using the pipe operator. This Bash utility directs the `stdout` or `stderr` of one application to another application placed to its right. By appending this tool to the invocation of `bpftool load`, developers can streamline their workflows. In this case the usage of the tool would look like:

```
bpftool prog load your_bpf.o /sys/fs/bpf/your_bpf 2>&1 |
    python3 path/to/pretty_verifier.py -c your_bpf.c
```

The `2 &1` operator is used to direct the `stderr` output of the `bpftool` to the `stdout`, in order for the tool to capture it. This integration can also be used in user-space functions that utilize the `libbpf` library for C when loading eBPF programs using the same syntax (or removing the `2 &1` operator if the error is printed directly in the `stdout`. In order to simplify this process, an *alias* can be created in the local Bash configuration to substitute the `python3 path/to/pretty_verifier.py` with just `pretty_verifier`, simplifying the command to:

```
bpftool prog load your_bpf.o /sys/fs/bpf/your_bpf 2>&1 |
            pretty_verifier -c your_bpf.c
```

The accepted parameters by the tool are the C source code file path, after the `-c` option, and the `.o` eBPF bytecode compiled object, after the `-o` option. Both the commands are optional. If they are not passed, or their value has been changed after the compilation is performed, some error messages will be limited. The usage of the `-c` option is strongly suggested to benefit the tool functionalities.

4.3 Tool Functionality

The application is designed to accept parameters, primarily the C source files of the eBPF program. These parameters enable the tool to associate errors with specific lines of code, making it easier for developers to understand the context of each error. Additionally, the option to provide the original eBPF bytecode allows for more comprehensive checks, especially in scenarios where the bytecode may not correspond directly to the C source files due to optimizations or modifications.

4.3.1 Code Structure and Execution Flow

The structure of the tool is organized to facilitate ease of use and maintainability, allowing to add and remove error managers and thus making it easily upgradable to newer version of the eBPF verifier . These components include:

Main Module in /pretty_verifier.py. The entry point of the application, responsible for handling command-line arguments and initiating the input processing.

Error Handler in /handler.py. This component processes the output from the eBPF verifier, utilizing regular expressions to detect specific error patterns. Upon identifying an error, it calls the appropriate error management functions to provide detailed feedback to the developer.

Error Managers in /error_managers.py. Each manager corresponds to different types of errors that the verifier may produce. These functions generate user-friendly error messages, adding relevant context that aids developers in diagnosing the issues.

Utilities in /utils.py. A set of helper functions that perform common tasks such as adding line numbers to error messages and retrieving bytecode information from eBPF files. These utilities streamline the processing of the verifier's output.

Tests in /test.py. The test suite developed for the tool, but not containing the test programs. More about this in chapter 5.

Execution Flow

The execution flow of the Pretty Verifier can be summarized in several steps:

1. Upon launching the application, the Pretty Verifier first parses the command-line arguments to identify the source files and any bytecode files provided by

the user. The files are crucial to improve the readability of the error, but they can also be omitted, with the result of the limitation of the message output given by the tool.

2. The tool reads the standard input in the main module, which contains the output of the eBPF verifier, and begins scanning for error keywords that indicate problems. It specifically look for the *processed* keyword, that implies, under the condition specified in the tool usage, the detection by the eBPF verifier of an error.

3. In the handler module the lines of the verifier's output are processed in reverse, and the tool searches for patterns that signify errors using regular expressions. When an error is detected, it triggers the error handling mechanism. Before the pattern matching, in case the user has passed the source C code to the tool, the tool also associate the C lines presented in the output with the line number, in order to be parsed and used by the error manager when creating the output message.

4. The error manager is invoked and starts parsing the output with the knowledge of what error is being thrown, since it is composed of various function for each error. It performs different tasks, adding information from the verifier source code, translating kernel enums into plain text, manipulating the strings in order to get a more strait forward error message.

5. The tool formats and outputs the enhanced error message to the console, right after the original output of the bpftool an the eBPF verifier.

4.3.2 Error handling

The core functionality of the application involves processing the output of the eBPF verifier. The tool scans for keywords that indicate the presence of an error. Regular expressions (regex) are employed to identify specific error types based on the error messages generated by the verifier. This regex-based approach allows the application to be flexible and extensible; new error types can be added by simply defining additional regex patterns without significant changes to the underlying code.

The regex used to detect the errors are the ones provided by the Python `re` library. Since the parsing of error messages is based upon the `verbose` function invocatiions in the eBPF verifier source code, it is possible to deduce the regex from the C formatted string. The common C format specifier are used, like %d for integers or %s for strings. Those can be captured using `(\d+)` for integers (or the (-?\d+) for possibly negative integers) and the `(.*?)` wildcard for generic strings. Adjustments can be made based on the specific error, for example when the `verbose` function can print just two or a restricted amount of values for a specific format specifier, it can be intercepted using the `(optionA|optionB)` pattern. The values read in the patterns are then passed to the error manager function and used in the final error message.

Error not managed

If an error message does not match any known patterns, the tool outputs a default message of “error not managed.” The study performed on the eBPF verifier code makes this option unlikely, however the occurrence of this message is an option to take into consideration, and should lead to an addition of a new error manager in the tool. A case scenario in which this message can be shown is when a newer kernel version is used, and then new error messages with new formatting are introduced, or less likely when an older version of the kernel is used, and some deprecated errors are removed in the newer versions. In the future developments of the tool the deprecated errors will be marked, but kept in the code base for backward compatibility.

4.3.3 Output

The enhanced error messages generated by the application follow a structured format, comprising several key components:

- **Additional error message**: A new message, related to the C code, usually providing high-level information about the type of error.
- **Location**: The line number and file name of the C source file where the error occurred.
- **Appendix**: This section provides additional detail for certain error messages that require more context.
- **Suggestion**: In some error messages, a suggestion is provided to help resolve the issue.

For each error message an output was crafted using this framework aiming to connect the error to the C source code counterpart and ease the debugging process. A more detailed description of how the errors where managed is shown in the notes of the OK and TESTED verbose messages in the Appendix A.

4.3.4 Added value

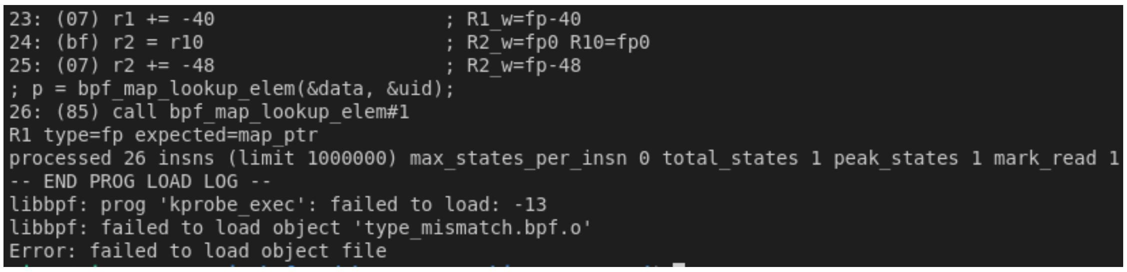
The additional information provided in the output varies depending on the specific error message encountered. The enhancements to the error logs can be structured as follows:

- **Translation of Terms:** It is common to use functions that translate the usage of specific terms by the eBPF verifier—typically C enumerations—into plain text. This makes the information comprehensible even to those unfamiliar with the kernel source code.
  - `get_type` function: This function translates the type of a BPF register into a human-readable description, as defined in the `reg_type` enumeration located in the `bpf.h` file of the kernel source code.
- **Error Explanations:** Explanations regarding the presence of specific errors can often be found in:
  - Comments within the source code.
  - Links to online forums that provide in-depth explanations about why certain errors occur and how to resolve them.
- **Specific Error Handling:** Different error types are processed with specific enhancements:
  - `type_mismatch`: This error indicates that a register is used in a helper function with a type that does not match the expected signature. Through the use of the register number provied by the output and the original C code, the tool manages to displays the specific C variable passed as a parameter that caused the error.
  - `invalid_access_to_map_value`: For errors related to mismatches in the number of bytes, this tool parses the relevant numbers to calculate the correct size required in that segment of the program.
- **Additional Enhancements:** Along with the improvements mentioned above, the output also includes:
  - The line number of the C source code where the error occurred.
  - Suggestions for resolving the errors to guide developers towards effective fixes.

4.3.5 C line number

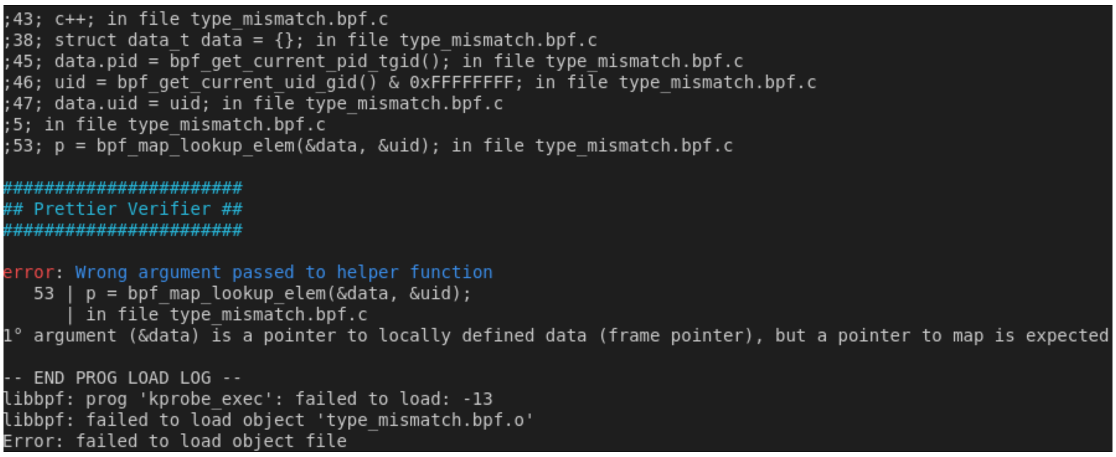
The debug mode of the Clang compilation manages to write as a comment in the BPF bytecode object the C lines from which it was generated. However it doesn't give any information about the location in the code. Specifically in the context of multiple file eBPF programs, when the eBPF verifier outputs these C lines, it might not be clear which file they belonged to. Therefore it was crucial to implement these feature directly in the tool.

Before the handler module starts parsing the error, a function found in the utilities module, called `add_line_number()` is invoked, binding the C line number directly to the C lines comments found in the eBPF verifier log. With multiple file support, the function manages to add also the file name, already formatted to be then shown in the output. In order to obtain a correct result, a double pointer approach is employed, advancing in the source C file when a new line in the raw output is found. Repeating line inside a file are tracked in a counter in both the file and the output, which gets decremented as they are encountered, avoid ambiguity. Lastly, this process is performed for each file, eventually choosing the one that has an higher number of processed lines In common scenarios the correct file has all its line processed, while the others has close to zero line processed. If two files have the same number of processed lines, it means they are the same.



```
23: (07) r1 += -40                    ; R1_w=fp-40
24: (bf) r2 = r10                     ; R2_w=fp0 R10=fp0
25: (07) r2 += -48                    ; R2_w=fp-48
; p = bpf_map_lookup_elem(&data, &uid);
26: (85) call bpf_map_lookup_elem#1
R1 type=fp expected=map_ptr
processed 26 insns (limit 1000000) max_states_per_insn 0 total_states 1 peak_states 1 mark_read 1
-- END PROG LOAD LOG --
libbpf: prog 'kprobe_exec': failed to load: -13
libbpf: failed to load object 'type_mismatch.bpf.o'
Error: failed to load object file
```

Figure 4.1: The *type_mismatch* error not managed



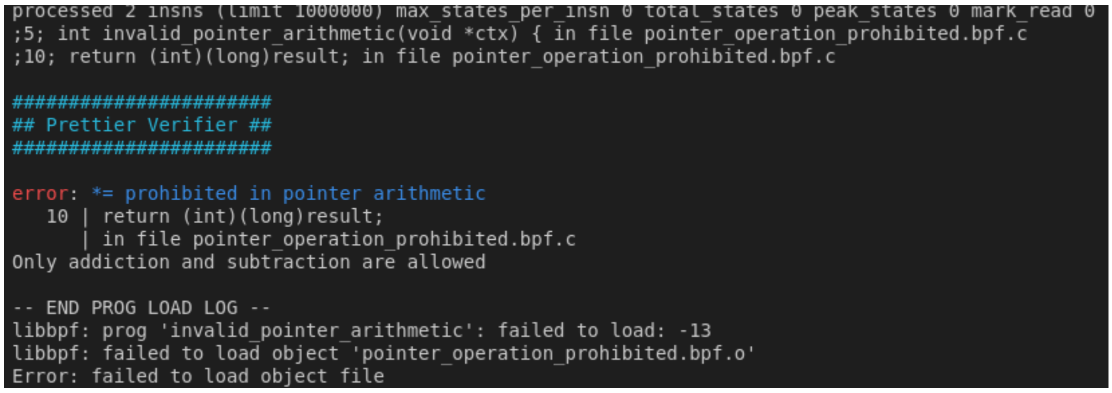
```
;43; c++; in file type_mismatch.bpf.c
;38; struct data_t data = {}; in file type_mismatch.bpf.c
;45; data.pid = bpf_get_current_pid_tgid(); in file type_mismatch.bpf.c
;46; uid = bpf_get_current_uid_gid() & 0xFFFFFFFF; in file type_mismatch.bpf.c
;47; data.uid = uid; in file type_mismatch.bpf.c
;5; in file type_mismatch.bpf.c
;53; p = bpf_map_lookup_elem(&data, &uid); in file type_mismatch.bpf.c

######################
## Prettier Verifier ##
######################

error: Wrong argument passed to helper function
   53 | p = bpf_map_lookup_elem(&data, &uid);
      | in file type_mismatch.bpf.c
1° argument (&data) is a pointer to locally defined data (frame pointer), but a pointer to map is expected

-- END PROG LOAD LOG --
libbpf: prog 'kprobe_exec': failed to load: -13
libbpf: failed to load object 'type_mismatch.bpf.o'
Error: failed to load object file
```

Figure 4.2: The *type_mismatch* error managed



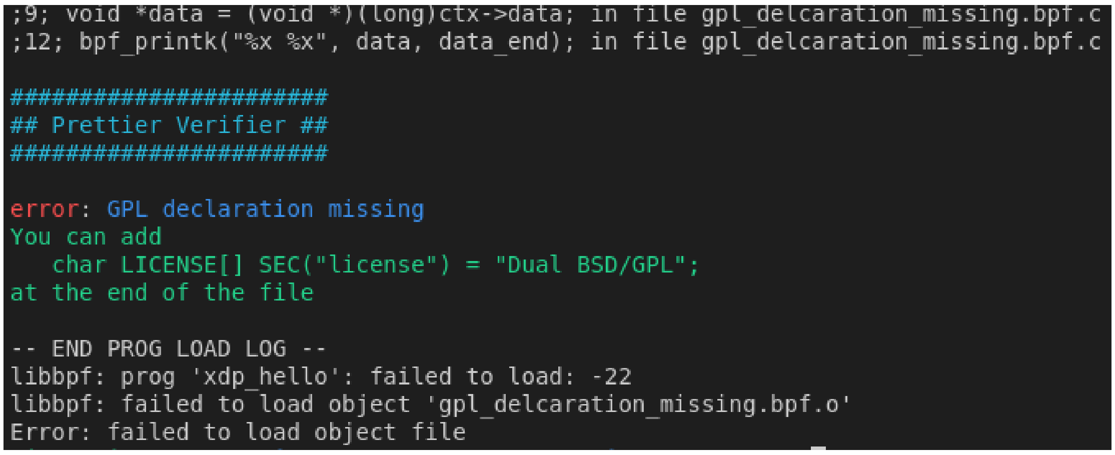
```
processed 2 insns (limit 1000000) max_states_per_insn 0 total_states 0 peak_states 0 mark_read 0
;5; int invalid_pointer_arithmetic(void *ctx) { in file pointer_operation_prohibited.bpf.c
;10; return (int)(long)result; in file pointer_operation_prohibited.bpf.c

######################
## Prettier Verifier ##
######################

error: *= prohibited in pointer arithmetic
   10 | return (int)(long)result;
      | in file pointer_operation_prohibited.bpf.c
Only addiction and subtraction are allowed

-- END PROG LOAD LOG --
libbpf: prog 'invalid_pointer_arithmetic': failed to load: -13
libbpf: failed to load object 'pointer_operation_prohibited.bpf.o'
Error: failed to load object file
```

Figure 4.3: The *pointer_operation_prohibited* error managed



```
;9; void *data = (void *)(long)ctx->data; in file gpl_delcaration_missing.bpf.c
;12; bpf_printk("%x %x", data, data_end); in file gpl_delcaration_missing.bpf.c

######################
## Prettier Verifier ##
######################

error: GPL declaration missing
You can add
   char LICENSE[] SEC("license") = "Dual BSD/GPL";
at the end of the file

-- END PROG LOAD LOG --
libbpf: prog 'xdp_hello': failed to load: -22
libbpf: failed to load object 'gpl_delcaration_missing.bpf.o'
Error: failed to load object file
```

Figure 4.4: The *gpl_declaration_missing* error managed

Chapter 5

Pretty Verifier Tests

The primary objective of the tool developed is to provide more user-friendly and structured output from the BPF verifier when it encounters errors in eBPF programs. The tool associates error codes with specific line numbers in the source code, suggests possible solutions, and provides more context around the issues found. Given this scope, the testing phase focused entirely on scenarios where the BPF verifier produces errors.

5.1 Structure of the Test Suite

The test suite follows a structured approach, where each test case is designed to trigger a specific verifier error. Each test case is represented by a function name and an expected output. The test process loads the corresponding eBPF program, runs the verification tool, and compares the verifier's output with the expected error message. This structure allows for easy identification of mismatches and ensures that the tool generates the correct diagnostic information for each error type.

The test framework is divided into two main components:

- **BPFTestCase**: This class is responsible for defining each individual test case. It includes the function name, the expected error output, and optionally the filename if it differs from the function name.
- **BPFTestSuite**: This manages the overall collection of test cases. It adds each test case to a list and runs them all sequentially. If a test case fails, the suite stops and provides detailed feedback on the error.

The test process also involves cleaning up the verifier's output before validation. This is necessary due to the presence of ANSI escape codes in the output, which can interfere with string comparisons. The cleaned output is then compared to the expected result to determine whether the test passes or fails.

5.1.1 Test folder

All the tests are stored in specific folder that can be chosen in the testing process. Specifically for the tests developed for this tool, they are stored in the eBPF code base repository, in the not-working/generated folder, as well as a copy of the tests present in the other folders of the repository, for a convenience. Those are all compiled through a `Makefile` and then loaded through a bash script called `load.sh`, even though

5.2 Test sources

The main sources of eBPF corrupted programs where the Github repository associated with the Liz Rice book Learning eBPF [14], and the blog post from Anteon [15], that covered about half of test cases. The other test cases were gathered from Stack Overflow and other online forums where the users were experience them, and some were appositely crafted in order to trigger the eBPF verifier error log.the test utility load them directly.

5.3 Testing Methodology

Testing required eBPF programs that intentionally cause verification failures. Unfortunately, publicly available resources for such programs are limited. After extensive research, a few relevant resources were found. As a result, the majority of the test cases had been purposefully created by writing custom C code that triggers the verifier's errors. This was a time-intensive process, as a wide range of common eBPF verification issues needed to be simulated.

5.3.1 Fuzz testing

One possible suggestion for improving the testing process was the use of fuzz testing. However, due to the nature of the tool working on C source code (rather than eBPF bytecode), implementing fuzz testing would have been significantly more complex. Moreover, the fuzzing tools currently available are focused on generating eBPF bytecode with the goal of making the eBPF verifier reach specific vulnerable states [16] [17].

5.3.2 Limitations in Error Message Coverage

One of the significant challenges faced during the testing process was the inability to generate C code that would trigger every possible error message from the BPF verifier. While error messages that are designed to be raised only from bytecode-level checks and thus couldn't be triggered by C source code were not taken into account during the development of the tool, there were also several messages that, theoretically, should have been triggered by C code but were not.

The inability to generate all error messages was due to the complex interaction between the eBPF verifier and the C language, which sometimes makes it difficult to predict exactly what kind of code will trigger certain errors. Despite countless efforts to craft specific test cases, certain error conditions remained elusive. As a result, not all possible verifier error messages are covered in the current testing suite. This limitation, while acknowledged, does not diminish the effectiveness of the tool for the majority of common verification issues, since the error managers not tested often have no documentation of people experiencing them on the online forums and open eBPF projects. As a matter of fact, the error messages that could not be tested are usually managed in the tool in order to address for isolated cases finding this error in online forums, that otherwise would be considered as rare errors.

5.4 Test Results and Coverage

In total the number of tested handler functions were 17, on a total of 79.

The test results for the tool show complete success in terms of passing all the tests that were designed. Since the test suite was developed alongside the tool itself, the pass rate of the tests is not the most significant metric. Rather, the focus should be on the coverage achieved by the tests.

The key module under testing, `error_managers.py`, where most of the core functions reside, achieved a code coverage of 36%, while approximately 20% of the total functions in the module were explicitly tested. This suggests that the coverage metric extends beyond the basic functionality tests, as auxiliary functions, such as the `get_type` function (used for translating file types), are also being implicitly tested as part of the overall process.

Additionally, a manual review of the code revealed that the test suite effectively covers nearly all portions of the code for which tests were created, indicating a well-constructed and comprehensive test design. Although there are some auxiliary functions not directly tested, the overall coverage demonstrates that the majority of the critical logic of the tool has been validated.

Chapter 6

Conclusions and Future Work

This thesis aimed to address the challenges that developers face when debugging eBPF programs due to the complexity and low readability of the error messages generated by the eBPF verifier. The work focused on improving the readability and interpretability of these messages, mapping them back to the original C code and making the developer experience more user-friendly.

The solution involved two main approaches: an initial attempt to leverage Clang's static analyzer and the development of a message error analysis tool. While the static analyzer approach proved insufficient due to the high number of false positives and the complexity of mimicking the eBPF verifier's operations on C code, the second approach, which focused on enhancing the verifier's error messages, yielded more promising results.

The tool developed in this thesis operates by intercepting and parsing the error messages generated by the eBPF verifier, linking them back to the original C code, and improving the clarity of the messages. This was achieved through the integration of utilities like `bpftool`, and by analyzing the most common error types to provide meaningful suggestions to developers.

The tests conducted on the tool show that all the designed tests passed successfully and the test cover achieved in the module `error_managers.py`, which contains the core logic of the tool, indicates that the error manager functions that were possible to be tested were covered. However, it is important to note that there remains a significant gap in the overall test suite, particularly regarding untested error cases. These untested cases, although included for completeness to account for isolated cases of developer encountering them online, they couldn't be reproduced in the testing environment. As a result, the tool lacks comprehensive coverage for all the error scenarios it is intended to handle. In the future, it will be essential to extend the test suite, aiming to cover the remaining error types and edge cases. Completing the test coverage will enhance the tool's reliability and ensure that it can manage all the error types encountered during eBPF program verification.

In conclusion, this thesis provides a robust framework for improving the developer experience when working with eBPF programs, through better error message analysis and clearer references to the original source code.

6.1 Future Works

6.1.1 Improved Test Coverage

An important step to be taken is to compete the test coverage for all the errors managed in the Pretty Verifier tool, since just about 20% of the function are properly tested, finding difficult to reproduce locally some of the errors that the tool manages. This would lead to a more complete and reliable work, allowing the tool to be more robust and correct.

6.1.2 Extending Compatibility to Newer Kernel Versions

The thesis was developed using Linux kernel version 6.6, which was the Long Term Support (LTS) release at the time. However, since the thesis began, Linux kernel version 6.8 has become the new LTS, and is being rapidly adopted by distributions like Debian. Expanding the tool to support future kernel versions would ensure that it remains relevant in a fast-evolving environment. Kernel upgrades often bring enhancements to the eBPF verifier and new error types, which would require ongoing adjustments to the tool to maintain its effectiveness.

6.1.3 Support for Additional Compilers

Currently, the tool relies on Clang for compiling eBPF programs with the `-target bpf` option, as Clang has been the most widely supported and integrated compiler for eBPF development. However, recent developments in GCC have introduced support for generating BPF bytecode [11]. A natural extension of this work would involve adapting the tool to handle GCC-generated bytecode, ensuring compatibility across different compilers. This would also involve addressing potential differences in error reporting between Clang and GCC, as each compiler may introduce its own nuances in terms of error handling and optimizations.

6.1.4 Handling Additional Programming Languages

At present, the tool focuses on interpreting C code, the primary language used for eBPF programs. However, there is potential to extend support to other languages, such as Rust, which is gaining popularity in systems programming. Rust provides safety guarantees that could reduce certain categories of errors typically encountered with C. Adapting the tool to parse and interpret Rust code, or other eBPF-compatible languages, could make it more versatile and beneficial to a broader developer audience.

Appendix A

Verbose list and checks

The following table shows all the occurrences of the verifier function `verbose`, used to display errors encountered by the verifier. The table lists the line number alongside the error message (or the function invoked to deliver the message), the status, and any relevant notes, such as links to forum threads discussing the error.

The **status** column, as discussed in Chapter 3, indicates the nature of the error:

- **LOG**: Message used for logging purposes.
- **VERIFIER ERROR**: Message used to signal an internal error of the verifier.
- **CE**: Message triggered by a compiler error, hence only triggerable via plain BPF bytecode.
- **NR**: Non-reachable error due to machine configuration or the nature of the code (It accounts also for rare errors).
- **BTF**: Error related to BTF (not directly related to the C code).
- **OK**: Managed error.
- **TESTED**: Managed and tested error.

Line	**Code**	**Status**	**Notes**
369	__printf(2, 3) static void verbose(...);	LOG	
391	__printf(3, 4) static void verbose_linfo(...);	LOG	

Line	**Code**	**Status**	**Notes**
410	verbose_invalid_scalar(...)	OK	This is a function used in other part of the verifier to signal a register holding a scalar that is out of bound (considering the variable offset). It was handled communicating the location in the C source code of the error and the range and the current value (is available) of the register.
670	print_liveness(...)	LOG	
712	%s has to be at a constant offset\n	NR	
718	cannot pass in %s at an offset=%d\n	NR	
724	cannot pass in %s at an offset=%d\n	NR	
1028	verifier internal error: misconfigured ref_obj_id\n	VERIFIER ERROR	
1069	cannot overwrite referenced dynptr\n	NR	
1356	print_verifier_state(...)	LOG	
1545	%c;	LOG	
1547	%d:	LOG	
2065	The sequence of %d jumps is too complex.\n	VERIFIER ERROR	
2149	mark_reg_known_zero(regs, %u)\n	LOG	
2499	mark_reg_unknown(regs, %u)\n	LOG	
2519	mark_reg_not_init(regs, %u)\n	LOG	
2600	The sequence of %d jumps is too complex for async cb.\n	VERIFIER ERROR	
2657	call to invalid destination\n	NR	
2664	too many subprograms\n	NR	
2765	too many different module BTFs\n	BTF	

Line	Code	Status	Notes
2770	kfunc offset > 0 without fd_array is invalid\n	BTF	
2781	invalid module BTF fd specified\n	BTF	
2786	BTF fd for kfunc is not a module BTF\n	BTF	
2827	negative offset disallowed for kernel module function call\n	BTF	
2854	calling kernel function is not supported without CONFIG_DEBUG_INFO_BTF\n	BTF	
2859	JIT is required for calling kernel function\n	NR	
2864	JIT does not support calling kernel function\n	NR	
2869	cannot call kernel function from non-GPL compatible program\n	TESTED	This error occurs when a program is missing the GPL compatibility declaration. It can be added using the `libbpf` macro `SEC("license")`. Suggesting this addition is the way it was handled, giving also a predefined formulation with the Dual BSD/GPL.
2897	failed to find BTF for kernel function\n	BTF	
2905	too many different kernel function calls\n	OK	This error indicates the usage of more kernel functions than allowed. It is not directly linkable to the C code, so the maximum number of functions allowed was suggested.
2912	kernel btf_id %u is not a function\n	BTF	

Line	**Code**	**Status**	**Notes**
2918	kernel function btf_id %u does not have a valid func_proto\n	BTF	
2926	cannot find address for kernel function %s\n	NR	
2938	address of kernel function %s is out of range\n	NR	
3027	loading/calling other bpf or kernel functions are allowed for CAP_BPF and CAP_SYS_ADMIN\n	NR	
3047	func#%d @%d\n	LOG	
3081	jump out of range from insn %d to %d\n	OK	This error is common in the presence of kernel functions calls, and it is specifically controlled in the cgf section. It occurs whenever the jump instruction is lower than 0 or more than the entire program length. It was managed indicating the two C lines (from and to) that triggered the error, and a suggestion to use helper function is added (since most of the online threads indicated this) `https://stackoverflow.com/questions/78373013/failed-to-load-ebpf-program`

Line	**Code**	**Status**	**Notes**
3093	last insn is not an exit or jmp\n	OK	This error occours during the check of subprograms, i.e. functions called by the eBPF program defined by the user. Each one of these must terminate with an exit instruction (exit, jump,ecc). It is handled giving back the C line where the function is called, and a suggestion based on the online forum threads.`https://stackoverflow.com/questions/62936008/attaching-ebpf-to-kprobe`
3122	verifier BUG type %s var_off %lld off %d\n	VERIFIER ERROR	
3346	R%d is invalid\n	CE	
3357	R%d !read_ok\n	TESTED	The occurrence of this error is common in scenarios where there is a program with an empty body. In this case the error has "R0" as register, and the tool suggest that this is not allowed. It can also occur when a function uses an uninitialized value. In this scenario the C variable passed to the parameter triggering the error is deduced, and an error message suggesting that an uninitialized variable is being used with the name of the varible is shown, as well as the C line where the error occurs.
3372	frame pointer is read only\n	CE	

Line	Code	Status	Notes
3502	BUG subprog enter from frame %d\n	VERIFIER ERROR	
3513	BUG subprog exit from frame 0\n	VERIFIER ERROR	
3662	mark_precise: frame%d: regs=%s	LOG	
3664	stack=%s before	LOG	
3665	%d:	LOG	
3727	BUG spi %d\n	VERIFIER ERROR	
3744	BUG spi %d\n	VERIFIER ERROR	
3775	BUG regs %x\n	VERIFIER ERROR	
3790	BUG regs %x\n	VERIFIER ERROR	
3818	BUG regs %x\n	VERIFIER ERROR	
3843	BUG regs %x\n	VERIFIER ERROR	
3862	BUG regs %x\n	VERIFIER ERROR	
3982	mark_precise: frame%d: falling back to forcing all scalars precise\n	LOG	
4001	force_precise: frame%d: forcing r%d to be precise\n	LOG	
4013	force_precise: frame%d: forcing fp%d to be precise\n	LOG	
4263	mark_precise: frame%d: last_idx %d first_idx %d subseq_idx %d \n	LOG	
4313	BUG backtracking func entry subprog %d reg_mask %x stack_mask %llx\n	VERIFIER ERROR	
4349	BUG backtracking idx %d\n	VERIFIER ERROR	
4408	mark_precise: frame%d: parent state regs=%s	LOG	

Line	Code	Status	Notes
4411	stack=%s:	LOG	
4564	attempt to corrupt spilled pointer on stack\n	CE	
4619	verbose_linfo(...)	LOG	
4620	invalid size of register spill\n	TESTED	This error occurs when the spilling of a pointer into the stack is attempted, but it's the size is different from the size of an eBPF register. It is handled showing the C line where the error occurred.
4624	cannot spill pointers to stack into stack frame of the caller\n	CE	
4745	spilled ptr in range of var-offset stack write; insn %d, ptr off: %d	CE	
4768	uninit stack in range of var-offset write prohibited for !root; insn %d, off: %d	CE	
4866	verbose_linfo(...)	LOG	
4867	invalid size of register fill\n	CE	
4893	invalid read from stack off %d+%d size %d\n	CE	
4918	leaking pointer from stack off %d\n	CE	
4932	invalid read from stack off %d+%d size %d\n	CE	
5021	variable offset stack pointer cannot be passed into helper function; var_off=%s off=%d size=%d\n	CE	
5091	write into map forbidden, value_size=%d off=%d size=%d\n	NR	
5097	read from map forbidden, value_size=%d off=%d size=%d\n	NR	

Line	**Code**	**Status**	**Notes**
5119	invalid access to map key, key_size=%d off=%d size=%d\n	TESTED	Managed in the error_manager.py since it is consequent other errors. See 5172, 5189, for details.
5123	invalid access to map value, value_size=%d off=%d size=%d\n	TESTED	Managed in the error_manager.py since it is consequent other errors. See 5172, 5189, for details.
5129	invalid access to packet, off=%d size=%d, R%d(id=%d,off=%d,r=%d)\n	TESTED	Managed in the error_manager.py since it is consequent other errors. See 5172, 5189, for details.
5134	invalid access to memory, mem_size=%u off=%d size=%d\n	TESTED	Managed in the error_manager.py since it is consequent other errors
5165	R%d min value is negative, either use unsigned index or do a if (index >=0) check.\n	OK	This error occurs whenever the fixed offset of the register has a minimum value that is negative during its usage. It is handled giving the C line, the suggestion is already inside the error message.

Line	**Code**	**Status**	**Notes**
5172	R%d min value is outside of the allowed memory range\n	TESTED	This error signals that the minimum value, considering variable and fixed offset, is outside the allowed memory range. It occurs because the bound check in the C program was not performed, so the minimum value is not yet acceptable from the eBPF verifier to access memory. It is followed by another message, signaling the type of pointer the user is trying to access. These are the messages 5123, 5129, 5134. For each one of those the error is managed in order to suggest the bound check to be performed when this is determinable from the information. It is also displayed the error in order to express the types in descriptive representation, and the C line where the error occurs is displayed.
5182	R%d unbounded memory access, make sure to bounds check any such access\n	CE	
5189	R%d max value is outside of the allowed memory range\n	TESTED	Same as 5172, but this is triggered for the maximum value stored by the eBPF verifier.

Line	Code	Status	Notes
5206	negative offset %s ptr R%d off=%d disallowed\n	OK	This error (and the next two) is triggered in presence of a modification of pointer that should not be modified, like context pointers. Each one of these three error check for different parameters tracked by the verifier. Is is handled returning the C line where the error occurred, and suggesting not to modify the pointer in the line (as written in the comment in the eBPF verifier source code).
5212	dereference of modified %s ptr R%d off=%d disallowed\n	OK	
5221	variable %s access var_off=%s disallowed\n	OK	
5298	invalid kptr access, R%d type=%s%s	BTF	
5299	expected=%s%s	BTF	
5302	or %s%s\n	BTF	
5304	\n	BTF	
5356	kptr in map can only be accessed using BPF_MEM instruction mode\n	CE	
5364	store to referenced kptr disallowed\n	CE	
5386	BPF_ST imm must be 0 when storing to kptr at off=%u\n	CE	
5390	kptr in map can only be accessed using BPF_LDX/BPF_STX/BPF_ST\n	CE	
5430	kptr cannot be accessed indirectly by helper\n	CE	
5434	kptr access cannot have variable offset\n	CE	

Line	**Code**	**Status**	**Notes**
5439	kptr access misaligned expected=%u off=%llu\n	CE	
5443	kptr access size must be BPF_DW\n	CE	
5449	%s cannot be accessed directly by load/store\n	CE	
5518	R%d min value is negative, either use unsigned index or do a if (index >=0) check.\n	OK	see 5165
5526	R%d offset is outside of the packet\n	TESTED	Same as 5172, but this error is triggered specifically for accessing to packet structures, where the bounds are specified in the structure itself. It is managed similarly.
5576	invalid bpf_context access off=%d size=%d\n	TESTED	This error occurs when the context structure provided during the triggering of the eBPF program is not performed correctly, This can happen when a program that is not allowed to directly access the context perform a dereferencing on it. It is managed by suggesting the C line where the error occurs.

Line	Code	Status	Notes
5586	invalid access to flow keys off=%d size=%d\n	OK	This error is caused by an incorrect access to flow keys, an abstraction to identify a data flow in a network It is usually caused by out of bound access to pointer, specifically because it was not bound checked before. Hence the error is managed suggesting how to fix the error, with the bound check to be added, and if needed the maximum size is reported (as the eBPF verifier also chack for this). The C line where the error is found is also displayed.
5603	R%d min value is negative, either use unsigned index or do a if (index >=0) check.\n	OK	see 5165
5632	R%d invalid %s access off=%d size=%d\n	CE	
5742	misaligned packet access off %d+%s+%d+%d size %d\n	OK	This error occurs when the access to a packet is not aligned, so it can be caused by mixing 32 and 64 bit types. It is managed suggesting the line where it occurs.
5766	misaligned %saccess off %s+%d+%d size %d\n	OK	Like the error before, but for generic types, it was handled also translating the type for the enum reg_type to the descriptive representation.

Line	Code	Status	Notes
5866	tail_calls are not allowed when call stack of previous frames is %d bytes. Too large\n	OK	This error occurs when the stack size of a subprogram is too large for a tail program to be called. It can be triggered by an elevated number of recursive calls to subprograms. It is handled suggesting the maximum number of calls.
5875	combined stack size of %d calls is %d. Too large\n	OK	This error is similar to the previous one, but takes into account the sum of all the stacks of all the subprogram in the program. It suggests the maximum number allowed.
5899	verifier bug. subprog has tail_call and async cb\n	VERIFIER ERROR	
5915	the call stack of %d frames is too deep !\n	OK	This error occurs when the stack size of a subprogram is too large. It can be triggered by an elevated number of recursive calls to tail functions. It is handled suggesting the maximum number of calls.
5983	R%d invalid %s buffer access: off=%d, size=%d\n	OK	It signals an invalid buffer to a pointer of type buffer. It is managed displaying the C line where the error occurs and suggesting that the offset should be negative (as the eBPF verifier code checks).
5992	R%d invalid variable buffer offset: off=%d, var_off=%s\n	CE	
6399	'struct %s' access is allowed only to CAP_PERFMON and CAP_SYS_ADMIN\n	BTF	

Line	Code	Status	Notes
6405	Cannot access kernel 'struct %s' from non-GPL compatible program\n	BTF	
6411	R%d is ptr_%s invalid negative access: off=%d\n	BTF	
6420	R%d is ptr_%s invalid variable offset: off=%d, var_off=%s\n	BTF	
6427	R%d is ptr_%s access user memory: off=%d\n	BTF	
6434	R%d is ptr_%s access percpu memory: off=%d\n	BTF	
6440	verifier internal error: reg->btf must be kernel btf\n	VERIFIER ERROR	
6450	only read is supported\n	BTF	
6456	verifier internal error: ref_obj_id for allocated object must be non-zero\n	VERIFIER ERROR	
6548	map_ptr access not supported without CONFIG_DEBUG_INFO_BTF\n	NR	
6554	map_ptr access not supported for map type %d\n	NR	
6564	'struct %s' access is allowed only to CAP_PERFMON and CAP_SYS_ADMIN\n	NR	
6570	R%d is %s invalid negative access: off=%d\n	BTF	
6575	only read from %s is supported\n	BTF	
6647	invalid unbounded variable-offset%s stack R%d\n	CE	
6666	invalid%s stack R%d off=%d size=%d\n	CE	

Line	Code	Status	Notes
6672	invalid variable-offset%s stack R%d var_off=%s size=%d\n	TESTED	This error occurs when the access to the stack is performed using an offset that is not within the allowed range. It is managed communicating the C line where the error is found.
6708	write to change key R%d not allowed\n	OK	The error occurs when the map key is overwritten. It is handled displaying the C line where the error occurs.
6723	R%d leaks addr into map\n	OK	It occurs when a pointer is stored in a map, leaking a reference. It is handled displaying the C line where the error occurs.
6763	R%d invalid mem access '%s'\n	TESTED	This error occurs when there is an access to not specific pointers that might be null. It is handled by showing the C line where the error occurred and suggesting to add a null check before.
6769	R%d cannot write into %s\n	OK	It occurs when writing in a read only memory part. It is handled displaying the C line where the error occurs and translating the type of the pointer that is read only.
6775	R%d leaks addr into mem\n	OK	It occurs when a write operation has as a value a pointer, leaking a reference. It is handled displaying the C line where the error occurs.

Line	Code	Status	Notes
6790	R%d leaks addr into ctx\n	OK	It occurs when a pointer is stored in a context data structure, leaking a reference. It is handled displaying the C line where the error occurs.
6801	verbose_linfo(...)	LOG	
6842	cannot write into packet\n	OK	It occurs when a write operation is performed on a read only packet data structure. It is handled displaying the C line where the error occurs.
6848	R%d leaks addr into packet\n	OK	It occurs when a pointer is stored in a packet data structure, leaking a reference. It is handled displaying the C line where the error occurs.
6858	R%d leaks addr into flow keys\n	OK	It occurs when a pointer is stored in a flow key data structure, leaking a reference. It is handled displaying the C line where the error occurs.
6868	R%d cannot write into %s\n	OK	see 6769
6892	R%d cannot write into %s\n	OK	see 6769
6907	R%d invalid mem access '%s'\n	TESTED	see 6763
6940	BPF_ATOMIC uses invalid atomic opcode %02x\n	CE	
6945	invalid atomic operand size\n	CE	
6968	R%d leaks addr into mem\n	OK	see 6775
6974	R%d leaks addr into mem\n	OK	see 6775
6984	BPF_ATOMIC stores into R%d %s is not allowed\n	NR	
7052	invalid zero-sized read\n	CE	

Line	Code	Status	Notes
7085	R%d%s variable offset stack access prohibited for !root, var_off=%s\n	CE	
7123	potential write to dynptr at off=%d disallowed\n	CE	
7138	verifier bug: allocated_stack too small	VERIFIER ERROR	
7167	invalid%s read from stack R%d off %d+%d size %d\n	CE	
7173	invalid%s read from stack R%d var_off %s+%d size %d\n	CE	
7207	R%d cannot write into %s\n	OK	see 6769
7223	R%d cannot write into %s\n	OK	see 6769
7234	R%d cannot write into %s\n	OK	see 6769
7279	R%d type=%s	TESTED	This error (that is completed considering the following) describes the situation in which a parameter with a wrong type is passed to an helper function. It is handled by deducing the parameter number and the C variable that caused it, the message suggest the C line where it happened, the variable that caused it and the type of the variable found and expected, in descriptive representation.
7280	expected=%s\n	TESTED	

Line	Code	Status	Notes
7315	R%d min value is negative, either use unsigned or ’var &= const’\n	OK	This error occurs whenever the fixed offset of the register has a minimum value that is negative during its usage. It is handled giving the C line, the suggestion is already inside the error message. `https://github.com/iovisor/bcc/issues/2391`
7329	R%d unbounded memory access, use ’var &= const’ or ’if (var <const)’\n	OK	This error occurs whenever the fixed offset of the register has a maximum value that exceeds 2^{29} during its usage. It is handled giving the C line, the suggestion is already inside the error message. `https://github.com/iovisor/bcc/issues/3409`
7436	R%d doesn’t have constant offset. bpf_spin_lock has to be at the constant offset\n	CE	
7444	map ’%s’ has to have BTF in order to use bpf_spin_lock\n	OK	This error, that can be considered of type BTF, so not directly depended by the C code, was managed because it occurred on the online forums. It can happen if the libbpf definition of a map is erroneus. It was managed giving the location and the name of the map in the C code.`https://github.com/cilium/ebpf/issues/1524`
7454	%s ’%s’ has no valid bpf_spin_lock\n	BTF	

Line	Code	Status	Notes
7459	off %lld doesn't point to 'struct bpf_spin_lock' that is at %d\n	NR	
7465	Locking two bpf_spin_locks are not allowed\n	NR	
7482	bpf_spin_unlock without taking a lock\n	NR	
7487	bpf_spin_unlock of different lock\n	CE	
7510	R%d doesn't have constant offset. bpf_timer has to be at the constant offset\n	BTF	
7515	map '%s' has to have BTF in order to use bpf_timer\n	BTF	
7519	map '%s' has no valid bpf_timer\n	BTF	
7524	off %lld doesn't point to 'struct bpf_timer' that is at %d\n	BTF	
7528	verifier bug. Two map pointers in a timer helper\n	VERIFIER ERROR	
7547	R%d doesn't have constant offset. kptr has to be at the constant offset\n	CE	
7552	map '%s' has to have BTF in order to use bpf_kptr_xchg\n	BTF	
7556	map '%s' has no valid kptr\n	BTF	`https://stackoverflow.com/questions/76035116/cannot-use-ebpf-kptr-ref-feature`
7564	off=%d doesn't point to kptr\n	BTF	
7568	off=%d kptr isn't referenced kptr\n	CE	
7610	verifier internal error: misconfigured dynptr helper type flags\n	VERIFIER ERROR	

Line	**Code**	**Status**	**Notes**
7633	Dynptr has to be an uninitialized dynptr\n	OK	It may occur in functions accepting dynamic pointers, that found it as uninitialized. It is managed by displaying the C line.
7649	cannot pass pointer to const bpf_dynptr, the helper mutates it\n	CE	
7656	Expected an initialized dynptr as arg #%d\n	OK	It may occur in functions accepting dynamic pointers, that found it initialized. It is managed by displaying the C line and the arg number.
7664	Expected a dynptr of type %s as arg #%d\n	OK	It may occur in functions accepting dynamic pointers, that found it with the wrong type. It is managed by displaying the C line, the argument number and translating the type of the dynptr.
7727	expected uninitialized iter_%s as arg #%d\n	OK	It occurs when an uninitialized iterator was expected. It is handled displaying the C line and the argument number.
7745	expected an initialized iter_%s as arg #%d\n	OK	It occurs when an initialized iterator was expected. It is handled displaying the C line and the argument number.
7941	verifier internal error: unexpected iterator state %d (%s)\n	VERIFIER ERROR	
7951	bug: bad parent state for iter next call	VERIFIER ERROR	
8014	invalid map_ptr to access map->type\n	NR	
8024	invalid arg_type for sockmap/sockhash\n	CE	

Line	**Code**	**Status**	**Notes**
8167	verifier internal error: unsupported arg type %d\n	VERIFIER ERROR	
8202	R%d type=%s expected=%s	TESTED	see 7279
8217	%s() may write into memory pointed by R%d type=%s\n	CE	
8239	Possibly NULL pointer passed to helper arg%d\n	BTF	
8245	verifier internal error: missing arg compatible BTF ID\n	VERIFIER ERROR	
8256	verifier internal error:	VERIFIER ERROR	
8258	R%d has non-overwritten BPF_PTR_POISON type\n	VERIFIER ERROR	
8267	R%d is of type %s but %s is expected\n	BTF	
8276	verifier internal error: unimplemented handling of MEM_ALLOC\n	VERIFIER ERROR	
8289	verifier internal error: invalid PTR_TO_BTF_ID register for type match\n	VERIFIER ERROR	
8340	R%d must have zero offset when passed to release func or trusted arg to kfunc\n	CE	
8392	verifier internal error: multiple dynptr args\n	VERIFIER ERROR	
8399	verifier internal error: no dynptr arg found\n	VERIFIER ERROR	
8441	verifier internal error: invalid spi when querying dynptr type\n	VERIFIER ERROR	
8470	R%d leaks addr into helper function\n	CE	

Line	**Code**	**Status**	**Notes**
8478	helper access to the packet is not allowed\n	OK	This error occurs when an access to a read only packet is made by an helper function. It is handled by showing the location in the C source code of the helper function.
8521	arg %d is an unacquired reference\n	CE	
8525	cannot release unowned const bpf_dynptr\n	CE	
8530	R%d must be referenced when passed to release function\n	NR	
8534	verifier internal error: more than one release argument\n	VERIFIER ERROR	
8544	verifier internal error: more than one arg with ref_obj_id R%d %u %u\n	VERIFIER ERROR	
8570	timer pointer in R1 map_uid=%d doesn't match map pointer in R2 map_uid=%d\n	NR	
8588	invalid map_ptr to access map->key\n	NR	
8604	invalid map_ptr to access map->value\n	NR	
8614	Helper has invalid btf_id in R%d\n	BTF	
8622	can't spin_{lock,unlock} in rbtree cb\n	NR	
8634	verifier internal error\n	VERIFIER ERROR	
8671	R%d is not a known constant'\n	CE	

Line	Code	Status	Notes
8698	R%d does not point to a readonly map'\n	OK	This error occurs when a not read only map is used by an helper function that requires one, in the case of argument that need to be strings. For this reason a suggestion was added in the handling of the error, as well as the location of the C line.`https://github.com/cilium/ebpf/discussions/722`
8703	R%d is not a constant address'\n	CE	
8708	no direct value access support for this map type\n	NR	
8721	direct value access on string failed\n	NR	
8727	string is not zero-terminated\n	NR	
8776	cannot update sockmap in this context\n	LOG	
8924	tail_calls are not allowed in non-JITed programs with bpf-to-bpf calls\n	NR	
9033	cannot pass map_type %d into func %s#%d\n	OK	It occurs when in the context of a helper call, a map is passed that has an incompatible type. It is handled displaying displaying the C line where the function is called.`https://github.com/xdp-project/xdp-tutorial/issues/65`
9224	the call stack of %d frames is too deep\n	OK	see 5915
9230	verifier bug. Frame %d already allocated\n	VERIFIER ERROR	

Line	Code	Status	Notes
9286	verifier bug: kfunc %s#%d not marked as callback-calling\n	VERIFIER ERROR	
9291	verifier bug: helper %s#%d not marked as callback-calling\n	VERIFIER ERROR	
9345	verifier bug. No program starts at insn %d\n	VERIFIER ERROR	
9355	Caller passes invalid args into func#%d\n	NR	
9360	Func#%d is global and valid. Skipping.\n	LOG	
9384	caller:\n	LOG	
9386	callee:\n	LOG	
9444	tail_call abusing map_ptr\n	CE	
9451	callback function not allowed for map\n	CE	
9641	cannot return stack pointer to the caller\n	CE	
9651	R0 not a scalar value\n	OK	The R0 is the regster used to return values, and it needs to be a scalar. So if a value that is not a scalar is returned, this error is triggered. It is handled with this explanation and the C line of the returned value
9662	verbose_invalid_scalar(...)	OK	see 410
9667	BUG: in callback at %d, callsite %d !calls_callback\n	VERIFIER ERROR	
9698	returning from callee:\n	LOG	
9700	to caller at %d:\n	LOG	
9782	kernel subsystem misconfigured verifier\n	VERIFIER ERROR	

Line	**Code**	**Status**	**Notes**
9795	write into map forbidden\n	OK	It prevents maps to be modified when read only using helper functions. It is handled displaying the C line where the error occurs.
9821	kernel subsystem misconfigured verifier\n	VERIFIER ERROR	
9858	Unreleased reference id=%d alloc_insn=%d\n	TESTED	This error occurs when a reference to some specific maps is not released before the termination of the program. Specifically it occurs for ring buffer, where to operate into it, we need to reserve the slot. It this is not released, the error occurs. It is managed by signaling where the reference was first acquired.
9887	verifier bug\n	VERIFIER ERROR	
9897	Invalid format string\n	CE	
9910	func %s#%d supported only for fentry/fexit/fmod_ret programs\n	NR	
9919	func %s#%d not supported for program type %d\n	NR	
9975	invalid func %s#%d\n	OK	This error occurs when the helper function has an id not valid. It usually signals that the helper function used is deprecated. It is handled displaying the C line of the handler function and suggesting to upgrade to the newer version.

Line	**Code**	**Status**	**Notes**
9983	unknown func %s#%d\n	OK	This error occurs when the helper function has an unknown id. It usually signals that the helper function used is deprecated. It is handled displaying the C line of the handler function and suggesting to upgrade to the newer version. `https://stackoverflow.com/questions/77225068/why-is-using-the-bpf-trace-printk-f`
9989	cannot call GPL-restricted function from non-GPL compatible program\n	TESTED	Similar to 2869, but for the usage of kernel functions.
9994	helper call is not allowed in probe\n	NR	
9999	helper call might sleep in a non-sleepable prog\n	NR	
10007	kernel subsystem misconfigured func %s#%d: r1 != ctx\n	NR	
10017	kernel subsystem misconfigured func %s#%d\n	NR	
10024	sleepable helper %s#%d in rcu_read_lock region\n	NR	
10068	verifier internal error: CONST_PTR_TO_DYNPTR cannot be released\n	VERIFIER ERROR	
10082	func %s#%d reference has not been acquired before\n	CE	
10091	tail_call would lead to reference leak\n	CE	
10100	get_local_storage() doesn't support non-zero flags\n	CE	
10134	frame%d bpf_loop iteration limit reached\n	LOG	

Line	Code	Status	Notes
10140	Unsupported reg type %s for bpf_dynptr_from_mem data\n	NR	
10151	BPF_LSM_CGROUP that attach to void LSM hooks can't modify return value!\n	NR	
10167	verifier internal error: meta.dynptr_id already set\n	VERIFIER ERROR	
10171	verifier internal error: meta.ref_obj_id already set\n	VERIFIER ERROR	
10177	verifier internal error: failed to obtain dynptr id\n	VERIFIER ERROR	
10183	verifier internal error: failed to obtain dynptr ref_obj_id\n	VERIFIER ERROR	
10252	kernel subsystem misconfigured verifier\n	VERIFIER ERROR	
10296	unable to resolve the size of type '%s': %ld\n	BTF	
10329	verifier internal error:	VERIFIER ERROR	
10331	func %s has non-overwritten BPF_PTR_POISON return type\n	VERIFIER ERROR	
10340	invalid return type %u of func %s#%d\n	BTF	
10349	unknown return type %u of func %s#%d\n	BTF	
10358	verifier internal error: func %s#%d sets ref_obj_id more than once\n	VERIFIER ERROR	
10398	verbose(env, err_str, func_id_name(func_id), func_id);	NR	
10653	max struct nesting depth exceeded\n	NR	

Line	Code	Status	Notes
10824	kernel function %s args#%d pointer type %s %s is not supported\n	BTF	
10847	arg#%d pointer type %s %s must point to %sscalar, or struct with scalar\n	BTF	
10910	kernel function %s args#%d expected pointer to %s %s but R%d has a pointer to %s %s\n	BTF	
10922	verifier internal error: ref_set_non_owning w/o active lock\n	VERIFIER ERROR	
10927	verifier internal error: NON_OWN_REF already set\n	VERIFIER ERROR	
10948	verifier internal error: ref_obj_id is zero for	VERIFIER ERROR	
10968	verifier internal error: ref state missing for ref_obj_id\n	VERIFIER ERROR	
11029	verifier internal error: unknown reg type for lock check\n	VERIFIER ERROR	
11038	held lock and object are not in the same allocation\n	NR	
11090	verifier internal error: unexpected graph root argument type %s\n	VERIFIER ERROR	
11096	verifier internal error: %s head arg for unknown kfunc\n	VERIFIER ERROR	
11117	verifier internal error: unexpected graph node argument type %s\n	VERIFIER ERROR	
11123	verifier internal error: %s node arg for unknown kfunc\n	VERIFIER ERROR	

Line	Code	Status	Notes
11140	verifier internal error: unexpected btf mismatch in kfunc call\n	VERIFIER ERROR	
11151	R%d doesn't have constant offset. %s has to be at the constant offset\n	CE	
11159	%s not found at offset=%u\n	CE	
11166	bpf_spin_lock at off=%d must be held for %s\n	NR	
11171	verifier internal error: repeating %s arg\n	VERIFIER ERROR	
11208	verifier internal error: unexpected btf mismatch in kfunc call\n	VERIFIER ERROR	
11219	R%d doesn't have constant offset. %s has to be at the constant offset\n	CE	
11226	%s not found at offset=%u\n	CE	
11242	operation on %s expects arg#1 %s at offset=%d	CE	
11252	arg#1 offset=%d, but expected %s at offset=%d in struct %s\n	CE	
11291	Function %s has %d > %d args\n	OK	This error occurs when the number of argument for a kernel function is more than the maximum value. It is handled by displaying the C line where the kernel function is declared and the maximum number of arguments allowed.

Line	Code	Status	Notes
11313	R%d is not a scalar\n	OK	This error occurs when a register (so in C a variable) that is not a scalar is passed to a kernel function that accept only a scalar in that spot. It is handled by calculating the argument number and showing the C line where the kernel function is found.
11319	verifier internal error: only one constant argument permitted\n	VERIFIER ERROR	
11323	R%d must be a known constant\n	CE	
11340	2 or more rdonly/rdwr_buf_size parameters for kfunc	NR	
11345	R%d is not a const\n	CE	
11358	Unrecognized arg#%d type %s\n	BTF	
11364	Possibly NULL pointer passed to trusted arg%d\n	OK	This error occurs when a register (so in C a variable) that might be null is passed to a kernel function that accept only "trusted" argument, so valid and not obtained from other pointers, in that spot. It is handled by showing the argument number and the C line where the kernel function is found, and suggesting to add a null check to the pointer.
11372	verifier internal error: more than one arg with ref_obj_id R%d %u %u\n	VERIFIER ERROR	
11395	R%d must be referenced or trusted\n	BTF	

Line	Code	Status	Notes
11399	R%d must be a rcu pointer\n	BTF	
11435	arg#%d expected pointer to ctx, but got %s\n	OK	This error occors when in kernel function a pointer to context is expected, but a different type of pointer is obtained. It is handled by showing the argument number and the C line where the kernel function is found, and translating the pointer received to descriptive representation.
11448	arg#%d expected pointer to allocated object\n	BTF	
11452	allocated object must be referenced\n	CE	
11468	arg#%d expected pointer to stack or dynptr_ptr\n	OK	This error occors when in kernel function a pointer to stack or dynamic pointer is expected, but a different type of pointer is obtained. It is handled by showing the argument number and the C line where the kernel function is found.
11487	verifier internal error: no dynptr type for parent of clone\n	VERIFIER ERROR	
11494	verifier internal error: missing ref obj id for parent of clone\n	VERIFIER ERROR	
11507	verifier internal error: failed to obtain dynptr id\n	VERIFIER ERROR	
11525	arg#%d expected pointer to map value or allocated object\n	BTF	
11529	allocated object must be referenced\n	CE	

Line	Code	Status	Notes
11539	arg#%d expected pointer to map value or allocated object\n	BTF	
11543	allocated object must be referenced\n	CE	
11552	arg#%d expected pointer to allocated object\n	BTF	
11556	allocated object must be referenced\n	CE	
11566	rbtree_remove node input must be non-owning ref\n	CE	
11570	rbtree_remove not allowed in rbtree cb\n	CE	
11575	arg#%d expected pointer to allocated object\n	BTF	
11579	allocated object must be referenced\n	CE	
11593	arg#%d is %s	OK	This message and the next one signal that a kernel function expected a pointer to a type or a socket but obtained another one. It is handled by showing the C line, the argument number, and the types expected and obtained in representative description.
11596	expected %s or socket\n	OK	
11607	arg#%d reference type('%s %s') size cannot be determined: %ld\n	BTF	
11624	arg#%d arg#%d memory, len pair leads to invalid memory access\n	CE	
11631	verifier internal error: only one constant argument permitted\n	VERIFIER ERROR	
11635	R%d must be a known constant\n	CE	

Line	**Code**	**Status**	**Notes**
11648	arg%d expected pointer to func\n	OK	This error occurs when a pointer to a function is expected in a kernel function. It is handled by showing the argument where it occurs and the C line of the kernel function invocation.
11655	arg#%d is neither owning or non-owning ref\n	NR	
11663	verifier internal error: Couldn't find btf_record\n	VERIFIER ERROR	
11668	arg#%d doesn't point to a type with bpf_refcount field\n	CE	
11680	release kernel function %s expects refcounted PTR_TO_BTF_ID\n	BTF	
11750	calling kernel function %s is not allowed\n	NR	
11759	destructive kfunc calls require CAP_SYS_BOOT capability\n	NR	
11765	program must be sleepable to call sleepable kfunc %s\n	NR	
11779	kfunc %s#%d failed callback verification\n	VERIFIER ERROR	
11792	Calling bpf_rcu_read_{lock,unlock} in unnecessary rbtree callback\n	NR	
11797	nested rcu read lock (kernel function %s)\n	NR	
11808	kernel func %s is sleepable within rcu_read_lock region\n	NR	
11814	unmatched rcu read unlock (kernel function %s)\n	NR	
11825	kfunc %s#%d reference has not been acquired before\n	CE	

Line	Code	Status	Notes
11839	kfunc %s#%d conversion of owning ref to non-owning failed\n	CE	
11846	kfunc %s#%d reference has not been acquired before\n	CE	
11862	acquire kernel function does not return PTR_TO_BTF_ID\n	NR	
11882	local type ID argument must be in range [0, U32_MAX]\n	NR	
11891	bpf_obj_new requires prog BTF\n	NR	
11897	bpf_obj_new type ID argument must be of a struct\n	NR	
11937	kfunc bpf_rdonly_cast type ID argument must be of a struct\n	NR	
11952	verifier internal error: bpf_dynptr_slice(_rdwr) no constant size\n	VERIFIER ERROR	
11966	the prog does not allow writes to packet data\n	CE	
11972	verifier internal error: no dynptr id\n	VERIFIER ERROR	
11983	kernel function %s unhandled dynamic return type\n	CE	
12002	kernel function %s returns pointer type %s %s is not supported\n	BTF	

Line	Code	Status	Notes
12125	math between %s pointer and %lld is not allowed\n	OK	This error occurs when the eBPF verifier detects a variable offset out of bound (2^{29}). This can be caused by alu operations performed between a pointer and scalar (to access it, like when an array is accessed by the [] operator). So it is handled by translating the type of the pointer to a descriptive representation, the C line where the error occourred is shown, and the offset used by the developer as well, comparing it to the maximum value possible, also shown.
12131	%s pointer offset %d is not allowed\n	OK	This error is similar to the one before and handled in a similar manner, but the fixed offset is the one exceeding in this situation.
12137	math between %s pointer and register with unbounded min value is not allowed\n	CE	
12143	value %lld makes %s pointer be out of bounds\n	TESTED	Error similar to the 12125, but it detect the minimum value of a register out of bound. It is handled similarly.
12375	R%d has unknown scalar with mixed signed bounds, %s\n	CE	
12379	R%d has pointer with unsupported alu operation, %s\n	CE	
12383	R%d tried to %s from different maps, paths or scalars, %s\n	NR	

Line	**Code**	**Status**	**Notes**
12387	R%d tried to %s beyond pointer bounds, %s\n	NR	
12391	R%d could not be pushed for speculative verification, %s\n	NR	
12395	verifier internal error: unknown reason (%d)\n	VERIFIER ERROR	
12423	R%d variable stack access prohibited for !root, var_off=%s off=%d\n	CE	
12429	R%d stack pointer arithmetic goes out of range,	CE	
12457	R%d pointer arithmetic of map value goes out of range,	NR	
12511	R%d 32-bit pointer arithmetic prohibited\n	OK	This error occurs when an alu operation between a pointer and a 32 bit register (so C variable) is performed. It is handled by communicating the location in the C code of the error and explaining the above description of the error.
12517	R%d pointer arithmetic on %s prohibited, null-check it first\n	OK	This error occurs when an alu operation is performed on a pointer that might be null. It is handled by communicating the location in the C code of the error and translating the type to descriptive representation.

Line	**Code**	**Status**	**Notes**
12537	R%d pointer arithmetic on %s prohibited\n	OK	This error occurs when an alu operation is performed on a pointer to network packet bounds. It is handled by communicating the location in the C code of the error and translating the type to descriptive representation.
12618	R%d tried to subtract pointer from scalar\n	TESTED	This error occurs because a pointer was subtracted from a scalar, leading to an unknown scalar. This is not allowed, and it is signaled as well as the location where this error happened in the C source code.
12627	R%d subtraction from stack pointer prohibited\n	CE	
12679	R%d bitwise operator %s on pointer prohibited\n	TESTED	This error is caused by bitwise operations on pointer, that are not allowed. This is handled by suggesting that only addition and subtraction are allowed and the line of C code that caused it.
12684	R%d pointer arithmetic with %s operator prohibited\n	OK	This error occurs as a default branch of a switch in the eBPF verifier among the alu operation. It can occur with multiplications on pointers. It is handled by showing the operation the C line where the error occurred and the operation rejected.

Line	**Code**	**Status**	**Notes**
13468	R%d pointer %s pointer prohibited\n	TESTED	This error is caused because two pointer have been used in an alu operation that is not subtraction. This is handled by reminding that only subtraction is allowed when combining two pointers and showing the location on the C source code of this error.
13509	verifier internal error: unexpected ptr_reg\n	VERIFIER ERROR	
13514	verifier internal error: no src_reg\n	VERIFIER ERROR	
13532	BPF_NEG uses reserved fields\n	CE	
13540	BPF_END uses reserved fields\n	CE	
13552	R%d pointer arithmetic prohibited\n	OK	
13565	BPF_MOV uses reserved fields\n	CE	
13571	BPF_MOV uses reserved fields\n	CE	
13577	BPF_MOV uses reserved fields\n	CE	
13588	BPF_MOV uses reserved fields\n	CE	
13623	R%d sign-extension part of pointer\n	OK	This error occurs when a pointer with an alignment inferior of 64 bit is sign extended. It is handled by suggesting the C line where the error occurs.

Line	**Code**	**Status**	**Notes**
13646	R%d partial copy of pointer\n	OK	This error occurs when the pointer is partially copied to a register (or C variable) with a saller alignment. It is handled with the displaying of the C line where the error occurred.
13700	invalid BPF_ALU opcode %x\n	CE	
13708	BPF_ALU uses reserved fields\n	CE	
13718	BPF_ALU uses reserved fields\n	CE	
13730	div by zero\n	CE	
13739	invalid shift %d\n	CE	
14498	invalid BPF_JMP/JMP32 opcode %x\n	NR	
14510	BPF_JMP/JMP32 uses reserved fields\n	CE	
14523	R%d pointer comparison prohibited\n	OK	Pointer comparison is not usually allowed in eBPF programs, except for the pointers to packet This is an error that occurs when a comparison between pointers not to packets is performed. It is handled suggesting this information and showing the C line where it occurred.`https://stackoverflow.com/questions/71351495/no-direct-packet-access-in-bpf-prog`
14528	BPF_JMP/JMP32 uses reserved fields\n	CE	
14722	R%d pointer comparison prohibited\n	OK	see 14523
14740	invalid BPF_LD_IMM insn\n	CE	

Line	Code	Status	Notes
14744	BPF_LD_IMM64 uses reserved fields\n	CE	
14778	bpf verifier is misconfigured\n	VERIFIER ERROR	
14790	missing btf func_info\n	BTF	
14794	callback function not static\n	CE	
14816	bpf verifier is misconfigured\n	VERIFIER ERROR	
14858	BPF_LD_[ABS|IND] instructions not allowed for this program type\n	CE	
14863	bpf verifier is misconfigured\n	VERIFIER ERROR	
14870	BPF_LD_[ABS|IND] uses reserved fields\n	CE	
14885	BPF_LD_[ABS|IND] cannot be mixed with socket references\n	NR	
14890	BPF_LD_[ABS|IND] cannot be used inside bpf_spin_lock-ed region\n	NR	
14895	BPF_LD_[ABS|IND] cannot be used inside bpf_rcu_read_lock-ed region\n	NR	
14901	at the time of BPF_LD_ABS|IND R6 != pointer to skb\n	CE	
14971	R0 leaks addr as return value\n	OK	This error occurs when a pointer is returned from the eBPF program, leaking it. It is handled by showing the line where the return occurred and suggesting not to return a pointer variable.
14981	In async callback the register R0 is not a known value (%s)\n	OK	see 15078, but for async callbacks

Line	**Code**	**Status**	**Notes**
14986	async callback	NR	
14995	At subprogram exit the register R0 is not a scalar value (%s)\n	OK	see 15078, but for subprograms
15078	At program exit the register R0 is not a known value (%s)\n	OK	This error signals that the eBPF program return a value that is not a scalar, which is no allowed. This is handled showing the C line of the return and its type in descriptive representation.
15083	verbose_invalid_scalar(...) exit	OK	see 410
15087	Note, BPF_LSM_CGROUP that attach to void LSM hooks can't modify return value!\n	LOG	
15189	verbose_linfo(...):	LOG	
15190	jump out of range from insn %d to %d\n	CE	
15211	verbose_linfo(...):	LOG	
15212	verbose_linfo(...):	LOG	
15213	back-edge from insn %d to %d\n	OK	This error occurs when a back jump is performed in the bytecode, that corresponds to a go to or a loop in C. It is handled by showing the lines from where the jump started to where if ended.`https://stackoverflow.com/questions/56872436/bpf-verifier-rejecting-xdp-program-`
15219	insn state internal bug\n	VERIFIER ERROR	
15384	visit_insn internal bug\n	VERIFIER ERROR	

Line	Code	Status	Notes
15392	pop stack internal bug\n	VERIFIER ERROR	
15401	unreachable insn %d\n	OK	This error occurs when dead code is found in the byte-code. Technically the compiler should remove unreachable code, but it may not be considered an error, so it is managed. It shows the location of the dead code in the C file.
15407	jump into the middle of ldimm64 insn %d\n	CE	
15429	LD_ABS is not allowed in subprogs without BTF\n	BTF	
15433	tail_call is not allowed in subprogs without BTF\n	BTF	
15468	number of funcs in func_info doesn't match number of subprogs\n	BTF	
15476	invalid func info rec size %u\n	CE	
15497	nonzero tailing record in func info	NR	
15520	nonzero insn_off %u for the first func info record	CE	
15526	same or smaller insn offset (%u) than previous func info record (%u)	CE	
15531	func_info BTF section doesn't match subprog layout in BPF program\n	CE	
15539	invalid type id %d in func info	CE	
15552	LD_ABS is only allowed in functions that return 'int'.\n	NR	
15556	tail_call is only allowed in functions that return 'int'.\n	NR	

Line	**Code**	**Status**	**Notes**
15634	nonzero tailing record in line_info	NR	
15663	Invalid line_info[%u].insn_off:%u (prev_offset:%u prog->len:%u)\n	NR	
15671	Invalid insn code at line_info[%u].insn_off\n	CR	
15678	Invalid line_info[%u].line_off or .file_name_off\n	NR	
15688	missing bpf_line_info for func#%u\n	CE	
15700	missing bpf_line_info for %u funcs starting from func#%u\n	CE	
15757	nonzero tailing record in core_relo	NR	
15773	Invalid core_relo[%u].insn_off:%u prog->len:%u\n	CE	
16438	frame %d: propagating r%d	LOG	
16440	,r%d	LOG	
16457	frame %d: propagating fp%d	LOG	
16459	,fp%d	LOG	
16465	\n	LOG	
16700	verbose_linfo(...)	LOG	
16701	infinite loop detected at insn %d\n	TESTED	This error is triggered by the presence of an infinite loop. This is possible since from version 5.3 of the Linux kernel, loops are allowed, if bounded. This error is handled by showing the location of the loop and suggesting the usage of loop unrolling directives or the bpf_loop function.

Line	**Code**	**Status**	**Notes**
16702	cur state:	LOG	
16704	old state:	LOG	
16977	same insn cannot be used with different pointers\n	OK	This error occurs when a load and a store instruction in a register have ponter with mismatched typers. It is handled with the suggestion above and the C line where it occurred.
17003	invalid insn idx %d insn_cnt %d\n	CE	
17013	BPF program is too large. Processed %d insn\n	OK	This error occurs when programs with more than 1 million instructions are found. This occurs usually when unrolling loops with too many cycles. It is handled showing the location of the C loop and suggesting the above information.`https://stackoverflow.com/questions/78603028/bpf-program-is-too-large-processed-`
17030	\nfrom %d to %d%s: safe\n	LOG	
17032	%d: safe\n	LOG	
17054	\nfrom %d to %d%s:	LOG	
17069	;	LOG	
17071	%d:	LOG	
17134	BPF_STX uses reserved fields\n	CE	
17164	BPF_ST uses reserved fields\n	CE	
17197	BPF_CALL uses reserved fields\n	CE	
17206	function calls are not allowed while holding a lock\n	NR	
17226	BPF_JA uses reserved fields\n	CE	

Line	**Code**	**Status**	**Notes**
17242	BPF_EXIT uses reserved fields\n	CE	
17248	bpf_spin_unlock is missing\n	NR	
17254	bpf_rcu_read_unlock is missing\n	NR	
17314	invalid BPF_LD mode\n	CE	
17318	unknown insn class %d\n	CE	
17379	invalid module BTF object FD specified.\n	BTF	
17384	kernel is missing BTF, make sure CONFIG_DEBUG_INFO_BTF=y is specified in Kconfig.\n	BTF	
17393	ldimm64 insn specifies invalid btf_id %d.\n	BTF	
17399	pseudo btf_id %d in ldimm64 isn't KIND_VAR or KIND_FUNC\n	BTF	
17408	ldimm64 failed to find the address for kernel symbol '%s'.\n	BTF	
17448	ldimm64 unable to resolve the size of type '%s': %ld\n	CE	
17518	tracing progs cannot use bpf_{list_head,rb_root} yet\n	NR	
17525	socket filter progs cannot use bpf_spin_lock yet\n	NR	
17530	tracing progs cannot use bpf_spin_lock yet\n	NR	
17537	tracing progs cannot use bpf_timer yet\n	NR	
17544	offload device mismatch between prog and map\n	NR	
17549	bpf_struct_ops map cannot be used in prog\n	NR	

Line	Code	Status	Notes
17572	Sleepable programs can only use array, hash, ringbuf and local storage maps\n	NR	
17606	BPF_LDX uses reserved fields\n	CE	
17620	invalid bpf_ld_imm64 insn\n	CE	
17655	unrecognized bpf_ld_imm64 insn\n	CE	
17663	fd_idx without fd_array is invalid\n	NR	
17679	fd %d is not pointing to valid bpf_map\n	NR	
17697	direct value offset of %u is not allowed\n	NR	
17703	no direct value access support for this map type\n	NR	
17711	invalid access to map value pointer, value_size=%u off=%u\n	NR	
17751	only one cgroup storage of each type is allowed\n	NR	
17765	unknown opcode %02x\n	CE	
17889	insn %d cannot be patched due to 16-bit range\n	NR	
18235	verifier bug. zext_dst is set, but no reg is defined\n	VERIFIER ERROR	
18275	bpf verifier is misconfigured\n	VERIFIER ERROR	
18281	bpf verifier is misconfigured\n	VERIFIER ERROR	
18396	bpf verifier narrow ctx access misconfigured\n	VERIFIER ERROR	
18415	bpf verifier is misconfigured\n	VERIFIER ERROR	
18423	bpf verifier narrow ctx load misconfigured\n	VERIFIER ERROR	
18616	JIT doesn't support bpf-to-bpf calls\n	NR	

Line	Code	Status	Notes
18713	calling kernel functions are not allowed in non-JITed programs\n	NR	
18720	tail_calls are not allowed in non-JITed programs with bpf-to-bpf calls\n	NR	
18728	callbacks are not allowed in non-JITed programs\n	NR	
18802	invalid kernel function call not eliminated in verifier pass\n	NR	
18815	verifier internal error: kernel function descriptor not found for func_id %u\n	VERIFIER ERROR	
18841	verifier internal error: kptr_struct_meta expected at insn_idx %d\n	VERIFIER ERROR	
18864	verifier internal error: kptr_struct_meta expected at insn_idx %d\n	VERIFIER ERROR	
18944	bpf verifier is misconfigured\n	VERIFIER ERROR	
19072	adding tail call poke descriptor failed\n	LOG	
19090	tail_call abusing map_ptr\n	NR	
19191	bpf verifier is misconfigured\n	VERIFIER ERROR	
19380	kernel subsystem misconfigured func %s#%d\n	NR	
19392	bpf verifier is misconfigured\n	VERIFIER ERROR	
19398	tracking tail call prog failed\n	VERIFIER ERROR	
19691	Func#%d is safe for any args that match its prototype\n	LOG	
19715	verification time %lld usec\n	LOG	
19716	stack depth	LOG	
19720	%d	LOG	

Line	**Code**	**Status**	**Notes**
19722	+	LOG	
19724	\n	LOG	
19730	processed %d insns (limit %d) max_states_per_insn %d	LOG	
19743	struct ops programs must have a GPL compatible license\n	BTF	
19751	attach_btf_id %u is not a supported struct\n	BTF	
19759	attach to invalid member idx %u of struct %s\n	BTF	
19769	attach to invalid member %s(@idx %u) of struct %s\n	BTF	
19778	attach to unsupported member %s of struct %s\n	BTF	
20137	Syscall programs can only be sleepable\n	BTF	
20142	Only fentry/fexit/fmod_ret, lsm, iter, uprobe, and struct_ops programs can be sleepable\n	BTF	
20271	in-kernel BTF is malformed\n	BTF	

Bibliography

[1] Liz Rice. *Learning EBPF: Programming the linux kernel for Enhanced Observability, networking, and security.* Sebastopol, CA: O'Reilly Media, Inc, 2023 (cit. on p. 2).

[2] Wikipedia contributors. *eBPF – Wikipedia, The Free Encyclopedia.* Online; accessed 1 September 2024. `https://en.wikipedia.org/wiki/EBPF`. 2023 (cit. on p. 2).

[3] Wikipedia contributors. *Linux Kernel – Wikipedia, The Free Encyclopedia.* Online; accessed 1 September 2024. `https://en.wikipedia.org/wiki/Linux_kernel`. 2023 (cit. on p. 2).

[4] eBPF Community. *eBPF Documentation.* Accessed: 2024-10-15. URL: `https://docs.ebpf.io/` (cit. on p. 2).

[5] Linux Kernel Documentation. *The Linux Kernel documentation.* Online; accessed 1 September 2024. `https://docs.kernel.org/`. 2023 (cit. on pp. 2, 23).

[6] Shung-Hsi Yu. *More than you want to know about BPF verifier.* Online; accessed 1 September 2024. Work at SUSE Labs, `https://www.youtube.com/watch?v=T4QAWIHb9ZU`. 2021 (cit. on p. 2).

[7] Prototype Kernel Documentation. *XDP - eXpress Data Path.* `https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/`. Accessed: October 16, 2024. 2024 (cit. on p. 5).

[8] Microsoft. *eBPF for Windows.* `https://github.com/microsoft/ebpf-for-windows`. Accessed: October 16, 2024. 2024 (cit. on p. 6).

[9] Linux Kernel Documetation. *eBPF Helper Functions.* `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`. Accessed: October 16, 2024. 2024 (cit. on p. 8).

[10] LLVM Foundation. *Clang User Manual.* Accessed: 2024-10-08. 2024. URL: `https://clang.llvm.org/docs/UsersManual.html` (cit. on p. 20).

[11] GCC Team. *GCC BPF Backend Documentation.* Accessed: 2024-10-08. 2024. URL: `https://gcc.gnu.org/wiki/BPFBackEnd` (cit. on pp. 21, 43).

[12] Ayar-s. *Ayar-s Documentation*. Accessed: 2024-10-08. 2024. URL: `https://aya-rs.dev/` (cit. on p. 21).

[13] Wikipedia contributors. *Register allocation*. Accessed: 2024-10-08. 2024. URL: `https://en.wikipedia.org/wiki/Register_allocation#:~:text=If%20there%20are%20not%20enough,then%20considered%20as%20%22split%22.` (cit. on p. 23).

[14] Liz Rice. *Learning eBPF*. `https://github.com/lizrice/learning-ebpf`. GitHub repository. 2023 (cit. on p. 39).

[15] Anteon Blog. *Unveiling eBPF Verifier Errors*. `https://getanteon.com/blog/unveiling-ebpf-verifier-errors/`. Accessed: October 16, 2024. 2023 (cit. on p. 39).

[16] bpfverif Team. *Agni: eBPF verifier testing framework*. Accessed: 2024-10-08. 2024. URL: `https://github.com/bpfverif/agni` (cit. on p. 39).

[17] Google Team. *Buzzer: eBPF fuzzer*. Accessed: 2024-10-08. 2024. URL: `https://github.com/google/buzzer` (cit. on p. 39).