POLITECNICO DI TORINO
Repository ISTITUZIONALE

Improving error messages for eBPF programs

Original
Improving error messages for eBPF programs / Rizza, R., Sisto, R., Valenza, F.. - In: INFORMATION AND SOFTWARE TECHNOLOGY. - ISSN 0950-5849. - 199:(2026). [10.1016/j.infsof.2026.108285]

Availability:
This version is available at: 11583/3014664 since: 2026-08-19T16:07:54Z

Publisher:
Elsevier

Published
DOI:10.1016/j.infsof.2026.108285

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

18 November 2026

Information and Software Technology 199 (2026) 108285

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Improving error messages for eBPF programs

Rosario Rizza *, Riccardo Sisto, Fulvio Valenza

DAUIN, Politecnico di Torino, Turin, Italy

ARTICLE INFO

Dataset link: https://github.com/netgroup-polito/pretty-verifier

Keywords:
eBPF
Software development
Linux kernel
Development experience

ABSTRACT

Context: eBPF is an emerging technology in cloud computing, allowing user-defined programs to run in kernel space for observability, networking, and security. To ensure system integrity, the kernel relies on the eBPF verifier, a static analyzer that rejects potentially unsafe code. However, the verifier's error messages are notoriously difficult to understand, generally referencing low-level bytecode rather than the original C source and making debugging a difficult and time-consuming task.
Objective: The goal of this work is to improve the eBPF verifier error messages and make debugging easier by mapping verification errors back to the original C source code and by providing more understandable feedback to developers.
Methods: This paper presents Pretty Verifier, a tool designed to improve the eBPF verifier error messages. By analyzing the verifier log and the compiler debug information, the tool maps verification errors back to the specific lines of C code, providing human-readable explanations and actionable fix suggestions. To rigorously validate the tool despite the scarcity of faulty eBPF datasets, we developed a fuzzing framework based on the BRF semantic fuzzer, capable of generating a balanced dataset of broken programs.
Results: Experimental results on over 400 test cases demonstrate that the tool successfully localizes errors in 84% of cases and provides precise, context-aware explanations.
Conclusion: Pretty Verifier significantly improves the developer's experience and facilitates the resolution of critical security issues by improving the readability of eBPF verifier messages and strengthening their connection to the original C source code.

1. 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 applications while maintaining a high level of safety and reliability. 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 kernel or applications.

The core enabler of this technology is the eBPF verifier, a static analyzer applied to eBPF programs before loading with the purpose of rejecting any program that could compromise kernel safety, integrity and security. For instance, a program could dereference a null pointer, leading to a system failure, or even worse, could access to unauthorized memory, leaking kernel space resources or disrupting kernel integrity. The verifier identifies and rejects programs that have these dangerous features, so protecting the kernel from malware and accidental errors.

In order to run an eBPF program, a developer usually writes the program source in C and compiles it into a common target assembly-like language called eBPF bytecode, which is then analyzed by the eBPF verifier during the *loading* of the program into the kernel space, through the `bpf()` system call. During the loading process, the `bpf()` system call, usually invoked through a library, outputs the eBPF verifier log, which communicates whether the program passed the checks or errors were found. One critical aspect of this process is that, in the latter case, the verifier log is often hard to understand, for example because it refers to C enums defined in the kernel source code or, most commonly, to aspects of the eBPF bytecode that was analyzed by the verifier, such as registers, that are difficult to track back to the original source code, despite the verifier outputs the line of source code the bytecode refers to. An example of the eBPF verifier log is shown in Fig. 2, which is the output produced for the C program shown in Fig. 1 and affected by an incorrect memory access error. The log starts and ends with some libbpf log messages (a library that allows eBPF lifecycle management in C), then followed by the actual eBPF verifier log. The log starts with a listing of the instructions of the program until the error

* Corresponding author.
E-mail addresses: rosario.rizza@polito.it (R. Rizza), riccardo.sisto@polito.it (R. Sisto), fulvio.valenza@polito.it (F. Valenza).

https://doi.org/10.1016/j.infsof.2026.108285
Received 29 April 2026; Received in revised form 8 July 2026; Accepted 28 July 2026
Available online 1 August 2026
0950-5849/© 2026 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

```c
int i = 1;
char array[6] = "String";
char LICENSE[] SEC("license") = "Dual BSD/GPL";

SEC("ksyscall/execve")
int kprobe_exec(void *ctx){
   i++;
   if (i <= sizeof(array)) {
      char value = array[i];
      bpf_printk("Usage of value %c\n", value);
   }

   return 0;
}
```

Fig. 1. C code producing an incorrect memory access error.

```
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
0: R1=ctx() R10=fp0
; i++;
0: (18) r2 = 0xffffabc7032b6000       ; R2_w=map_value(map=max_v
alu.data,ks=4,vs=10)
2: (61) r1 = *(u32 *)(r2 +0)          ; R1_w=scalar(smin=0,smax=
umax=0xffffffff,var_off=(0x0; 0xffffffff)) R2_w=map_value(map=ma
x_valu.data,ks=4,vs=10)
3: (07) r1 += 1                       ; R1_w=scalar(smin=umin=1,
smax=umax=0x100000000,var_off=(0x0; 0x1ffffffff))
4: (63) *(u32 *)(r2 +0) = r1          ; R1_w=scalar(smin=umin=1,
smax=umax=0x100000000,var_off=(0x0; 0x1ffffffff)) R2_w=map_value
(map=max_valu.data,ks=4,vs=10)
5: (67) r1 <<= 32                     ; R1_w=scalar(smax=0x7ffff
fff00000000,umax=0xffffffff00000000,smin32=0,smax32=umax32=0,var
_off=(0x0; 0xffffffff00000000))
6: (77) r1 >>= 32                     ; R1_w=scalar(smin=0,smax=
umax=0xffffffff,var_off=(0x0; 0xffffffff))
; if (i <= sizeof(array)) {
7: (25) if r1 > 0x6 goto pc+10        ; R1_w=scalar(smin=smin32=
0,smax=umax=smax32=umax32=6,var_off=(0x0; 0x7))
; char value = array[i];
8: (18) r2 = 0xffffabc7032b6004       ; R2_w=map_value(map=max_v
alu.data,ks=4,vs=10,off=4)
10: (0f) r2 += r1                     ; R1_w=scalar(smin=smin32=
0,smax=umax=smax32=umax32=6,var_off=(0x0; 0x7)) R2_w=map_value(m
ap=max_valu.data,ks=4,vs=10,off=4,smin=smin32=0,smax=umax=smax32
=umax32=6,var_off=(0x0; 0x7))
11: (71) r3 = *(u8 *)(r2 +0)
invalid access to map value, value_size=10 off=10 size=1
R2 max value is outside of the allowed memory range
processed 10 insns (limit 1000000) max_states_per_insn 0 total_s
tates 0 peak_states 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.bp
f.o'
Error: failed to load object file
```

Fig. 2. Log of the eBPF verifier for the code in Fig. 1.

is detected, with each C code statement (starting with a semicolon) followed by the corresponding bytecode instructions and eBPF verifier state information. When the error is detected, the listing stops and the log continues with the error message, including some other state information. The difficulty of interpreting the verifier log is a limitation that makes fixing the issues raised by the verifier difficult and time consuming. Our goal is to improve the readability of the eBPF verifier log, enriching its connections with the C source code, and guiding the developer to understand and fix the issues that might compromise the system, thus enhancing and speeding up the developer's fixing activity. To achieve this, we developed a command-line utility, *Pretty Verifier*, which parses eBPF verifier logs and analyzes the object code generated by Clang, the compiler commonly used for C eBPF programs. It then maps the errors back to the original C source code and provides guidance to help understand and fix them.

The design of the tool was preceded by a comprehensive study of the eBPF verifier source code, aimed at identifying the error messages that have to be handled. Several difficulties were encountered in obtaining faulty eBPF C programs suitable for testing, as discussed in this paper. This limitation motivated the development of a fuzzing framework based on the BRF semantic fuzzer described in [1]. The generated test cases allowed us to effectively evaluate the tool's functionality.

This paper extends the previous conference paper [2] where we provided a preliminary presentation of our work. Starting from that work, a thorough evaluation of the tool and some improvements on it have been added to arrive at this extended version.

The rest of the paper is organized in the following way. In Section 2, we first give an overview of the related work. Then, in Section 3, we present the main features of the eBPF verifier. Then, our study of the eBPF verifier is discussed in Section 4. Next, in Section 5, the *Pretty Verifier* utility is presented, discussing its architecture, features, methodology, and improvements over the current eBPF development pipeline. The testing methodologies and experimental results are discussed in Section 6, while the conclusions and future developments are exposed in Section 7.

2. Related work

The strict safety and integrity guarantees required by the Linux kernel make eBPF development difficult. Although the eBPF verifier is essential for maintaining system stability, its growing complexity, now exceeding 20,000 lines of code as documented in [3], has become a significant obstacle to programmability. Several research efforts [4,5] have explored alternative verification approaches, pointing out structural limitations in the current design. This complexity is often framed as a security concern (see [6–8], and [9]), but it also creates a practical usability problem, increasing the gap between what developers intend to implement and what the kernel ultimately accepts. Recent studies, such as the work [10] on specification-based oracles for detecting misbehavior, highlight the complex gap between the developer's intent and the strict constraints enforced by the kernel.

There have been various attempts to improve this ecosystem. On one hand, community efforts, such as the `bpfvv` (BPF Verifier Visualizer) project [11], aim to improve the debugging experience by providing a structured visualization of verifier logs. On the other hand, [12] have analyzed the community response to eBPF, to understand major difficulties from a development perspective.

The difficulty in interpreting verifier output is not unique to eBPF but reflects a broader challenge in software engineering. Research on compiler usability, such as [13], has long established that cryptic error messages are a significant barrier to productivity and learning. Similarly, extensive studies on static analysis tools such as [14] highlight that unintelligible feedback often leads to tool under-utilization or abandonment by developers . While recent advancements explore using Large Language Models (LLMs) to explain code errors, these probabilistic approaches introduce latency, privacy concerns regarding source code, and non-determinism. In contrast, our work focuses on a deterministic, privacy-preserving approach to bridge the semantic gap between the kernel verifier and the developer.

Regarding the specific eBPF domain, we noticed that the approach we used in this article, based on enriching the eBPF verifier error messages with information coming from the analysis of the object code, has not been explored so far in the literature. However, eBPF verifier errors have been studied and employed in previous works, such as [1,15], to detect error types and improve the acceptance rate of input programs during fuzz testing of the eBPF verifier. In particular, [1] presents an analysis of the eBPF verifier source code similar to the one we present in Section 4, but with the different purpose of identifying semantically related error messages. In contrast, our focus was primarily on the connection of the errors to the original C code.

Table 1
Classification of `verbose()` calls in the eBPF verifier (Linux 6.8 LTS)

Category	Count
Log messages	54
Internal errors	93
Bytecode errors	121
Other errors not relevant for our study	161
Errors relevant for our study	111
Total	**540**

3. The eBPF verifier

The verifier is the backbone upon which the eBPF technology managed to be widely adopted in production applications, since it ensures that the loaded programs do not disrupt kernel safety, integrity and security.

3.1. Verification process

The eBPF verifier is a static analysis tool that inspects code prior to execution. It analyzes bytecode instead of source code, as the latter can be compiled in different ways, while the bytecode reflects what is actually interpreted or JIT-compiled for execution. To perform the analysis, the verifier processes the eBPF instructions in the object file and explores all possible execution paths based on logical branches. Pruning techniques are applied to eliminate equivalent paths. For each analyzed path, the verifier tracks how eBPF registers interact and the nature of the data they hold—such as type and estimated value—to ensure safe function calls and memory operations like dereferencing. It also detects instructions that could lead to kernel crashes, information leakage, or other safety and security issues.

3.2. Verification criteria

The eBPF verifier follows some verification criteria in order to ensure correctness of the program and prevent the issues it might cause. Liz Rice, in the "Learning eBPF" book [16], does an excellent job listing the main criteria. They include:

- Validation of the use of helper functions, i.e., special functions provided by the bpf module to allow specific operations otherwise prohibited. This validation makes sure that the called functions exist, that they are called in the correct program type, and that their arguments correctly fit the function signature.
- Checks on access to memory correctness, to avoid out-of-bound accesses and unauthorized read/writes in objects not allowing it.
- License verification, making sure the program is GPL-compatible whenever GPL-licensed helper functions are used.
- Pointer dereferencing checks, in order to avoid the dereferencing of null pointers, requiring null checks before usage.
- Checks on the correct usage of the context parameter, passed to the eBPF program and containing information from the system calls or the network packets that triggered the execution. The context must follow some strict usage policies, sometimes requiring helper functions to perform read/write operations.
- Termination-related checks ensure that the program does not contain loops with a variable number of iterations, unless they are handled by specific helper functions. Additional constraints include the presence of a return value, a non-empty main body, a maximum of one million instructions, and the exclusion of unreachable code.
- Correctness of the bytecode, ensuring that only existing operations are performed and that illegal operations, like writing to the read only stack pointer register R10, are not.

4. Study of the eBPF verifier errors

In order to develop the tool, a deep study of the eBPF verifier code was conducted, looking for all the possible error messages it may emit and selecting those that may refer to errors in the input code that prevent loading. As documented in [3], the eBPF verifier emits numerous error messages, logs, and warnings, all printed through the `verbose` function and wrappers. Other logging functions appearing in the eBPF verifier code are the `bpf_log` function, just used for logging purposes, and internal kernel diagnostics macros, like `WARN_ONCE`. These other functions are not relevant for our study since they do not play a role in the log printing for verification errors found in the analyzed code. Therefore, they can be ignored, focusing on `verbose` calls only.

The total number of calls to the `verbose` function for the eBPF verifier in the Linux kernel version 6.8 LTS exceeds 500 (Table 1). However, this number does not properly indicate the number of different types of *source code* errors that may trigger eBPF verifier errors. Indeed, several cases, outlined below, have been identified in which calls to the function do not correspond to errors that may occur in the C source code.

4.1. Log messages

In several cases, the `verbose` function is used just to log the eBPF verifier's state. For instance, the function `print_verifier_state` displays the eBPF verifier's internal state at the time of the call. This function is typically invoked when internal errors occur, although a logging level can be set to force its call for each instruction. Hence, these calls to the `verbose` function are not relevant for understanding the types of verification issues found in the code. A total of 54 uses of the `verbose` function of this type were found.

4.2. Internal errors

The `verbose` function is also used to notify internal errors occurring in the eBPF verifier. The eBPF verifier developers inserted unconditional exits in situations that, at design time, were considered impossible. These exits are usually accompanied by an error message starting with ``verifier internal error:'' or ``BUG'' to flag faults in the eBPF verifier's own code. Of course, these errors are not relevant for interpreting eBPF code verification errors, as their occurrence just indicates issues in the eBPF verifier code itself or problems that require investigation by the kernel developers. A total of 93 uses of the `verbose` function of this type were found.

4.3. Bytecode errors

Many error messages that can be emitted by the eBPF verifier can exclusively be triggered by writing incorrect eBPF bytecode manually or by incorrect translation of the LLVM code generated from the original C source, which, of course, should not occur. In these situations, explaining the error would not significantly help the C code developer in resolving it. A total of 121 error messages of this type were found.

4.4. Other errors not relevant for our study

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 hard coded and used on a machine with a different kernel version or configuration). BTF-related 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) do not refer to code defects. Lastly, several errors are extremely rare, and limited documentation exists online

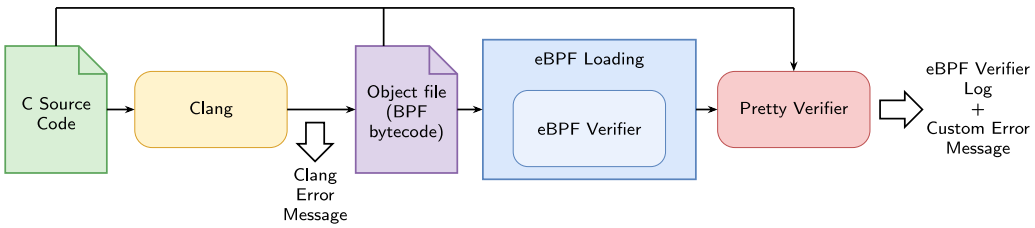


Fig. 3. Pretty Verifier role in the eBPF development pipeline.

about them. 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 study, 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 number of error messages of this type, plus the ones caused by BTF and OS configuration errors, is 161.

The remaining 111 error messages are those that may be relevant for our study and that we decided to manage in the tool.

5. Pretty verifier

The Pretty Verifier tool was developed to assist programmers in writing eBPF programs for the Linux kernel, and is specifically tailored to the messages emitted by the eBPF verifier in kernel version 6.8. However, since the most common error messages rarely change, it is expected to work with other kernel versions as well. The source code is publicly available in the Netgroup-Polito GitHub repository [17].

5.1. Methodology and design

The purpose of the tool is to introduce an additional clarity layer on top of the existing eBPF verifier error messages, with the goal of bringing the verifier log closer to the original C source code. To this end, the tool integrates information extracted from the C source code, the debug information embedded in the object file generated during compilation, and the eBPF verifier log itself. By correlating these sources, we strengthen the link between the verifier output and the corresponding C code, while also providing guidance to eBPF developers by highlighting common causes of verifier halts and suggesting typical resolution strategies.

In order to obtain this result, our approach proceeds as follows (Fig. 3):

1. The C source code is compiled into bytecode using `Clang`.
2. The loader attempts to load the program, triggering the kernel verifier.
3. If verification fails, *Pretty Verifier* intercepts the raw verifier log.
4. The tool analyzes three inputs: the verification result, the original C source code, and the compiled bytecode.
5. Finally, it augments the original log with a structured and human-readable explanation.

The tool appends its analysis output directly after the original verifier log and includes the following 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**: Additional detail for certain error messages that require more context.
- **Suggestion**: Suggestions to help resolve the issue.

As an example, the output produced for the code in Fig. 1 is provided in Fig. 4: the standard kernel log appears first (here only the last part is shown), followed by the further explanation generated by our tool, which is highlighted starting from the cyan "Pretty Verifier" header. For comparison, Fig. 2 shows the raw, unmodified verifier output for the same error.

5.2. Error messages selection

The analysis of the eBPF verifier source code, described in Section 4, identified 111 error messages that could potentially arise from how the C source code has been developed. To manage these 111 messages, we developed 91 handlers designed to intercept the errors and produce the Pretty Verifier output. These handlers are sufficient to cover all 111 errors, as multiple `verbose` messages sometimes stem from the same underlying issue or are handled by shared logic within the tool. Finally, to assess the impact of these errors on the developer experience, we conducted an online search to determine their frequency in developer discussions. We found that only a limited subset of these errors appears in online forums, while the rest are either completely absent or rarely mentioned. Nevertheless, we adopted a conservative inclusion strategy and implemented handlers also for error messages for which we could not find concrete evidence of being triggered directly through C source code examples. In these cases, the inclusion of the corresponding handlers was motivated by the previously described analysis of the eBPF verifier's source code, and their implementation was directly derived from that analysis. This design choice ensures that the tool provides comprehensive coverage, accounting not only for common errors, for which we were able to create the C source code that triggers them, but also for rare edge cases that might emerge within particularly complex or infrequently used code structures.

5.3. Enhancement strategies

The proposed tool operates as a post-processing layer that is located right after the standard eBPF verifier log, employing multiple mechanisms to reconstruct the missing context not supplied during the compilation and verification process. The enhancement process is built upon the following core mechanisms.

C source-level error localization. A limitation of raw verifier logs is their reliance on bytecode instruction indices, which offer little immediate context. The tool overcomes this by automating the mapping of these indices back to the specific filename and C source line number. By parsing the debugging information, it pinpoints exactly where the error occurred in the source files and isolates the relevant function parameters and variable names. Specifically, it is obtained by correlating the instruction numbers reported in the verifier log with the debug information stored in the compiled object file. Since the program is compiled with debug symbols, the object file contains the mapping between eBPF instructions and source locations. Pretty Verifier uses this mapping to recover the C line associated with the verifier instruction closest to the reported failure, and includes the corresponding file name and line number in the enriched diagnostic.

Heuristic analysis and reasoning. Beyond basic mapping, the tool analyzes the verifier state to derive meaningful insights. It performs straightforward computations (e.g., calculating the required bounds for safety checks) and traces program state changes (e.g., following resource allocation paths) to infer the developer's intent and provide clear, actionable feedback.

Semantic translation. The tool intercepts internal kernel terminology, often represented as raw C enumerations (e.g., `reg_type` from `bpf.h`), and translates them into plain and human-readable descriptions. This abstraction layer removes the need for developers to decipher internal register types or obscure error codes manually.

Knowledge augmentation. The enhanced logs incorporate insights derived directly from the eBPF verifier's source code comments and external documentation. This enriches generic error messages with explanations of the underlying checks (e.g., the rationale behind prohibited pointer arithmetic), providing developers with the theoretical background needed to resolve complex issues.

5.4. Enhancement examples

The following subsections detail some of the most significant error management strategies adopted by Pretty Verifier, categorized by the nature of the verification check. In all the screenshots presented (Figs. 4–14), the first part (upper portion) is the original output produced by the eBPF verifier, whereas the colored section below represents the additional diagnostic information added by Pretty Verifier.

5.4.1. Memory safety and pointer analysis

These handlers address errors related to invalid memory accesses, pointer arithmetic, and bounds checking. The tool focuses on clarifying complex error messages and identifying the specific variable constraints required by the verifier.

Memory bounds violations. Standard verifier errors regarding accesses to strictly defined memory regions, such as BPF maps, packet data, or memory buffers, often lack source-level specificity. Pretty Verifier handles these errors by extracting from the verifier log, specifically the abstract register state at the failing instruction computed by the verifier, the target region type, its size ($region_size$), the access offset (off), and the access size ($size$). The tool then models the accessed interval as $[off, off + size)$ and compares it with the valid interval $[0, region_size)$, distinguishing underflows from overflows and reporting the number of bytes by which the access exceeds the valid range. Moreover, when the failing line contains a simple indexed access, such as `array[i]` or `obj->array[i]`, the tool tries to recover the indexed object and its size from the source declaration. (See Fig. 4) In these cases, the suggestion refers directly to the source-level index and reports the valid range. If this information cannot be inferred reliably, the tool falls back to a generic bounds-check suggestion.

Unsafe scalar ranges for accessing memory. When a scalar variable is used for pointer arithmetic or as a size parameter without explicit constraints, the verifier rejects the register state itself due to potential unbounded memory accesses. The tool analyzes the context to propose specific fixes and it suggests imposing limits using bitwise masks or conditional checks. Notably, for helper functions like `bpf_probe_read`, which frequently appear in online reports for this error, the tool infers the implicit limit by checking the size of the destination buffer passed in a previous register, effectively suggesting a bound check tailored to the specific buffer being used. (See Fig. 5)

Forbidden pointer operations. The verifier emits generic messages for invalid pointer operations. The tool analyzes the specific operator that caused the failure (e.g., multiplication, division, bitwise AND/OR) and refines the message by explicitly stating that pointers may only be used with addition and subtraction, as can be inferred from the verifier's source code. Additionally, it determines whether a shift operation was likely intended to represent a multiplication or division and provides contextual information about the verifier's constraints. (See Fig. 6)

```
invalid access to map value, value_size=10 off=10 size=1
R2 max value is outside of the allowed memory range
processed 10 insns (limit 1000000) max_states_per_insn 0 total_s
tates 0 peak_states 0 mark_read 0

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

error: Invalid access to map value
   20 | char value = array[i];
      | in file /home/ubuntu/ebpf/pretty-verifier/tests/test_cas
es/max_value_is_outside_map_value.bpf.c
Access is 1 bytes past the end of the map value.

Make sure that the index 'i' is checked to be within the 'array'
 bounds (0 to 5).
```

Fig. 4. Message of the Pretty Verifier for incorrect memory access.

```
R2 min value is negative, either use unsigned or 'var &= const'
processed 15 insns (limit 1000000) max_states_per_insn 0 total_s
tates 1 peak_states 1 mark_read 1

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

error: Minimum possible value is not allowed to be negative
   29 | v4 = bpf_probe_read_kernel(v1, v3, &bpf_prog_active);
      | in file /home/ubuntu/ebpf/pretty-verifier/tests/test_cas
es/min_value_is_negative.bpf.c

Use v3 as unsigned or add 'v3 &= const'
v3 must be smaller than the size of the dest buffer v1
```

Fig. 5. Message of the Pretty Verifier for possibly negative value passed to an helper function.

```
R0 pointer arithmetic with /= operator prohibited
processed 2 insns (limit 1000000) max_states_per_insn 0 total_st
ates 0 peak_states 0 mark_read 0

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

error: Division prohibited in pointer arithmetic
   25 | void *result = (void *)(((unsigned long)ptr)/5);
      | in file /home/ubuntu/ebpf/pretty-verifier/tests/test_cas
es/pointer_arithmetic_with_operator_division.bpf.c

Only addition and subtraction are allowed
```

Fig. 6. Message of the Pretty Verifier for unauthorized pointer arithmetic.

5.4.2. Resource lifecycle and control flow

The eBPF verifier enforces strict rules regarding resource acquisition, release, and program termination. These handlers track the lifecycle of resources and analyze the execution flow to enhance verifier error messages about leaks and infinite loops.

Unreleased references in ring buffers. When using a ring buffer, a developer must reserve space and subsequently submit or discard it. If a program exits without releasing a reserved resource, the verifier blocks the load without specifying which part of the C code is responsible for the unreleased reference. The tool handles this by parsing the error message to identify the specific allocation instruction reported by the verifier. Then, using this instruction as an anchor, it scans the verifier output enriched with the source-line information obtained through `llvm-objdump` and DWARF debug data, locating the bytecode instruction and the corresponding C line where the resource was acquired, such as a call to `bpf_ringbuf_reserve`. This significantly improves the raw verifier output, which otherwise would not pinpoint the relevant source line. (See Fig. 7)

```
Unreleased reference id=3 alloc_insn=10
processed 23 insns (limit 1000000) max_states_per_insn 1 total_s
tates 2 peak_states 2 mark_read 1

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

error: Reference must be released before exiting
   33 | e = bpf_ringbuf_reserve(&rb, sizeof(*e), 0);
      | in file /home/ubuntu/ebpf/pretty-verifier/tests/test_cas
es/unreleased_reference.bpf.c
```

Fig. 7. Message of the Pretty Verifier for reference not released before exiting.

```
Unreleased reference id=3 alloc_insn=8
tail_call would lead to reference leak
processed 14 insns (limit 1000000) max_states_per_insn 0 total_s
tates 1 peak_states 1 mark_read 1

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

error: Reference must be released before tail call invocation
   24 | v2 = bpf_ringbuf_reserve(&map_1, v0, v1);
      | in file /home/ubuntu/ebpf/pretty-verifier/tests/test_cas
es/tail_call_lead_to_leak.bpf.c
```

Fig. 8. Message of the Pretty Verifier for reference not released before tai call invocation.

```
infinite loop detected at insn 13
cur state: R0_w=scalar() R10=fp0 fp-8=mmmm????
old state: R0_w=scalar() R10=fp0 fp-8_r=mmmm????
processed 29 insns (limit 1000000) max_states_per_insn 0 total_s
tates 2 peak_states 2 mark_read 1

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

error: Infinite loop detected
   32 | for (int i = 0; i < max; i++)
      | in file /home/ubuntu/ebpf/pretty-verifier/tests/test_cas
es/infinite_loop_detected.bpf.c

You may add #pragma unroll before the for loop line
```

Fig. 9. Message of the Pretty Verifier for an infinite loop detection.

Tail call resource leaks. Tail calls allow one eBPF program to call another without returning. However, passing control to another program while holding a reference (like a lock or a ring buffer reservation) is forbidden because the callee cannot release the caller's resources. The tool detects this specific condition and, similar to the standard exit check, scans the instruction history to identify, in the C source code, which resource was not released before the `bpf_tail_call` invocation. (See Fig. 8)

Loop detection. The eBPF verifier enforces strict termination guarantees, often rejecting loops that cannot be statically unrolled. When the verifier detects a potentially infinite loop, it reports the instruction index at which the issue is detected. The tool uses this index as an anchor and correlates it with the source-line information obtained through `llvm-objdump` and DWARF debug data. In this way, it locates the bytecode instruction associated with the loop and maps it back to the corresponding C source line. This mechanism allows the tool to pinpoint the exact loop construct in the high-level source code and suggests adding the `#pragma unroll` when appropriate, helping the compiler generate code that satisfies the verifier's termination and boundedness requirements. (See Fig. 9)

```
R1 type=fp expected=map_ptr
processed 26 insns (limit 1000000) max_states_per_insn 0 total_s
tates 1 peak_states 1 mark_read 1

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

error: Wrong argument passed to helper function
   56 | p = bpf_map_lookup_elem(&data, &uid);
      | in file /home/ubuntu/ebpf/pretty-verifier/tests/test_cas
es/type_mismatch.bpf.c
1° argument (&data) is a pointer to locally defined data (frame
pointer), but a pointer to map is expected
```

Fig. 10. Message of the Pretty Verifier for type mismatch in a helper function call.

```
invalid bpf_context access off=76 size=4
processed 5 insns (limit 1000000) max_states_per_insn 0 total_st
ates 0 peak_states 0 mark_read 0

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

error: Invalid access to context parameter
   21 | void *data = (void *)(long)skb->data;
      | in file /home/ubuntu/ebpf/pretty-verifier/tests/test_cas
es/invalid_bpf_context_access_socket.bpf.c
Cannot read or write in the context parameter for the socket pro
gram type

https://docs.ebpf.io/linux/program-type/ has a detailed table on
 which fields of the context, for each program type, can be read
 or written, in the "Context/Context fields section".
```

Fig. 11. Message of the Pretty Verifier for invalid access to context parameter.

5.4.3. Type consistency and compatibility

Many verification failures arise from mismatches among program type, kernel version, and specific arguments passed to helper functions. These handlers provide context-aware suggestions to resolve syntactic and compatibility issues.

Wrong argument type in helper function. When a specific argument in a helper function call fails type verification, the tool performs a syntactic analysis of the C source line. It isolates the function parameters passed in the failing position by relying on the eBPF helper calling convention, where the register number signaled by the verifier corresponds to the argument position, providing immediate context to the developer. Additionally, the tool handles "polymorphic" constraints where the verifier accepts multiple valid types for a single argument. It parses the list of expected types, translates each into a human-readable format using the internal dictionary, and constructs a comprehensive error message. (See Fig. 10)

Context access permissions. Access to fields within the BPF context structure is strictly regulated based on the program type (e.g., XDP, socket filter). When an invalid read or write is detected, the tool identifies the program type by parsing the C source code and extracting the section declaration, such as `SEC(''...'')`. It explicitly informs the developer that the attempted operation is forbidden for that specific context type and directs them to the documentation detailing the read/write permissions for the relevant fields. (See Fig. 11)

Map configuration and tail calls. The error message "kernel subsystem misconfigured verifier" appears multiple times in the eBPF verifier source code. However, in Linux kernel version 6.8, we observed that this error may be triggered by mismatches between map types and helper functions, particularly when `bpf_tail_call` is used. The tool detects this specific error signature in conjunction with the use of `bpf_tail_call` and suggests that the user is likely attempting

```
kernel subsystem misconfigured verifier
processed 3 insns (limit 1000000) max_states_per_insn 0 total_st
ates 0 peak_states 0 mark_read 0

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

error: Map configuration error
   14 | bpf_tail_call(ctx, &map_0, (u32)&map_0);
      | in file /home/ubuntu/ebpf/pretty-verifier/tests/test_cas
es/kernel_subsystem_misconfigured_verifier.bpf.c

bpf_tail_call() must be used with a map of type BPF_MAP_TYPE_PRO
G_ARRAY
```

Fig. 12. Message of the Pretty Verifier for map configuration error when `bpf_tail_call` is used.

```
unknown func bpf_ktime_get_coarse_ns#160
processed 7 insns (limit 1000000) max_states_per_insn 0 total_st
ates 0 peak_states 0 mark_read 0

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

error: Unknown function bpf_ktime_get_coarse_ns
   34 | u64 key = bpf_ktime_get_coarse_ns();
      | in file /home/ubuntu/ebpf/pretty-verifier/tests/test_cas
es/unknown_func.bpf.c

Check if the helper function you are using is compatible with th
e program type kprobe/vfs_read
for your kernel version 6.8 at https://docs.ebpf.io/linux/helper
-function/bpf_ktime_get_coarse_ns/
```

Fig. 13. Message of the Pretty Verifier for unknown helper function.

to perform a tail call with an incorrect map type, recommending the required use of a `BPF_MAP_TYPE_PROG_ARRAY`. (See Fig. 12)

Helper function compatibility. The verifier often flags functions as "unknown" not because they do not exist, but because they are not exposed to the specific BPF program type being compiled or are unsupported by the running kernel version. The tool contextualizes this error by extracting the host kernel version and deriving the program type from the C source-level section declaration. It then generates a targeted suggestion to verify the specific helper function's compatibility matrix, guiding the user to the official documentation, that contains the program types and the kernel versions accepted for the helper function. (See Fig. 13)

5.5. Fallback strategy for complex control flows

In the first development iterations of the Pretty Verifier, certain challenges were identified. Specifically, we observed some issues when Clang optimizations were enabled for programs with complex control flows, including one or more branches with significant similarities. In such cases, the debug information generated by the compiler occasionally caused the tool to misidentify the C line corresponding to an error. To mitigate this problem, a new fallback mode, called *full mode*, was developed. This modality allows developers to temporarily compile and test the C code without any optimizations, ensuring that the tool provides more accurate error reporting. It is conceptually different from the normal mode, because it can be directly called just by passing the source C code. Under the hood, it compiles and attempts to load the program automatically disabling the compiler optimizations, by the usage of the `pragma` directive *clang optimizations off*. This obviously limits its use to programs that do not strictly rely on the user space counterpart.

We must emphasize that this fallback mode is strictly a temporary diagnostic step used to pinpoint the exact C line. Once the error is identified and fixed, the final production artifact is compiled with full optimizations enabled

Although using *full mode* can, in some cases, significantly alter the resulting bytecode and potentially reduce its correspondence to the original C source code, these discrepancies were not significant in the scenarios studied. In contrast, disabling optimizations in such cases helped the tool correctly identify the lines in the C code where errors occurred. This feature provides developers with a fallback option when the default behavior of the tool struggles to produce clear and precise results.

5.6. Implementation details

The tool is implemented in python and operates downstream of the standard compilation and loading pipeline. It relies on standard eBPF loaders (such as `libbpf`, `bpftool`, `cilium/ebpf`, or `libbpf-rs`) to retrieve the raw eBPF verifier log.

To enable this mapping between bytecode and source code, it is mandatory to compile the eBPF program with the `-g` option, which embeds debug information (DWARF) into the compiled object.

Internally, the tool parses the verifier output using a set of Regular Expressions (regex) to detect specific keywords and error patterns. This regex-based architecture ensures flexibility and extensibility: new error types can be supported by simply defining additional patterns without modifying the core logic. In cases where an error message does not match any predefined pattern, the tool outputs a default "error not managed" notification. This fallback mechanism serves as a signal to developers that a new handler needs to be implemented.

Modern eBPF programs rely on diverse loading mechanisms, ranging from manual CLI operations to complex userspace agents. To accommodate this variety, *Pretty Verifier* supports two distinct integration paradigms.

5.6.1. CLI pipeline integration

For rapid prototyping and manual debugging, the tool operates as a CLI program designed to be used with the pipe operator. It intercepts the verification log stream generated by command-line loaders (most notably `bpftool`) or by any custom userspace program that emits the eBPF verifier log to standard output or error streams. To further streamline the workflow, the tool includes a *Loader Script Generator* utility (`genloader`). This utility automates the creation of Bash scripts that handle compilation, loading, and verification in a single step.

5.6.2. Native library bindings

Complex eBPF applications often employ custom userspace loaders written in high-level languages. To integrate with these environments, *Pretty Verifier* provides native libraries that intercept the verifier log buffer directly in memory.

- **C Library:** Integrates with `libbpf`, allowing developers to pass the `kernel_log_buf` directly to the `pretty_verifier` function before printing to the console.
- **Go and Rust Bindings:** Recognizing the prevalence of cloud-native tools written in Go (e.g., Cilium) and Rust, we developed idiomatic bindings. For Go, the library intercepts `Verifier-Error` types from the `cilium/ebpf` collection. For Rust, it hooks into the `libbpf-rs` logger callback.

6. Results and validation

In order to demonstrate the reliability and effectiveness of the tool, an extensive series of tests were conducted. This task turned out to be particularly difficult, considering the quite limited availability of eBPF programs already including errors. A few notable resources provided valuable starting points, like Liz Rice's "Learning eBPF" GitHub repository [18], which contains a handful of examples featuring faulty eBPF

code, or the Anteon blog post [19] where further examples of verifier errors and misbehaviors are provided. However, these resources were not sufficient to cover all the error messages handled by the tool.

To address this issue, we used the knowledge gained from the eBPF verifier study to develop several faulty eBPF C programs, aiming to cover as many error scenarios as possible. However, this task proved to be challenging because intentionally writing C code with unsafe operations often does not lead to an eBPF verifier rejection. This is mainly due to compiler optimizations and code rearrangements, which often eliminate or neutralize the unsafe behavior, resulting in an accepted program. Moreover, when a program is actually rejected, it usually triggers one of the few most common eBPF verifier error messages. As a result, this approach, together with the C code collected from faulty eBPF repositories and from online forums where developers reported similar issues let us produce test cases covering only 17 of the 111 potentially relevant handlers.

Taking into account the difficulty in getting coverage with manually developed tests, we employed an automated, large-scale validation strategy based on fuzzing. With this approach, we were able to evaluate the tool's behavior against a diverse and statistically significant set of eBPF programs, analyzing the tool correctness and precision across a wide spectrum of edge cases.

6.1. Fuzzing framework

The Fuzzing Framework is a tool that can be used to automatically generate a collection of eBPF C programs that do not pass verification, producing various verifier errors. We chose to base this framework on the state-of-the-art semantic fuzzer *BRF* (BPF Runtime Fuzzer) [1]. BRF was originally designed to generate valid C programs that successfully pass eBPF verification by applying semantic rules extracted from an analysis of the eBPF verifier logic. The motivation for this work was to avoid the limitations of approaches based on direct generation of BPF bytecode instructions, which often result in programs being rejected or discarded by the compiler when unused. Instead, BRF relies on guided C source code generation, enforcing semantic constraints such as correct argument types for helper functions and valid memory accesses ensured through proper bounds checking.

Our approach consisted of modifying BRF's source code by altering these generation mechanisms. Specifically, we introduced various configurable options, called injectors, to deliberately violate selected semantic rules. This allowed the generation of faulty eBPF programs that were expected to make verifier checks fail.

6.2. Injection methodology

This approach must be contextualized within the methodological constraints of the underlying verifier models. The semantic rules underlying BRF were developed through an iterative, error-driven process that extrapolates the eBPF verifier's logic. Although this method has proven effective, it results in an empirical approximation rather than a fully formal and complete specification. This inherent limitation is further amplified by a version mismatch, since the model was built for Linux kernel version 5.15 while our analysis focuses on version 6.8. Consequently, the integration of our fault injectors is inherently non-exhaustive. Rather than attempting a comprehensive mapping of all possible verifier violations, our strategy focused on identifying specific locations within the fuzzer's generation pipeline where deliberate modifications would trigger a rejection. These targeted areas were selected based on our understanding of the verifier's behavior and the specific error messages identified during the preliminary analysis in Section 4.

Moreover, this injection methodology serves as a targeted, heuristic approach. Its primary objective is not just to expand the absolute number of tested errors, but more importantly, to significantly increase the structural variance of the programs associated with each specific error message. By maximizing this variance, we were able to test our tool against a diverse dataset of distinct eBPF programs that trigger the exact same verifier error. This structural diversity is fundamental for comprehensively evaluating the tool's robustness and effectiveness across a wide range of code contexts and edge cases that lead to identical rejection messages.

6.2.1. Fault injection mechanisms

The injectors corrupt valid code patterns by introducing intentional faults during instruction synthesis. The following list exemplifies the most significant mechanisms implemented:

Randomized context access. Valid BPF programs must access the context (e.g., `sk_buff`) using strict offset and size alignments. This mechanism forces the generation of memory load or store instructions on the context register using random offsets and operand sizes, deliberately triggering out-of-bound or misaligned access violations that the verifier must catch.

Arbitrary helper calls. BPF helpers define a strict interface regarding allowed arguments (e.g., pointers vs scalars) and return types. This injection disrupts the compliant generation of helper calls by selecting random helper IDs or passing incompatible argument types.

Constraint bypass. Normally, the generator maintains an internal state to track register types and prevent invalid operations. This mechanism disables these semantic constraints, allowing the generator to produce syntactically valid but semantically unsafe instructions, such as performing arithmetic on incompatible register types or using uninitialized registers.

Randomized return values. The BPF verifier strictly checks the final value in register `R0` upon program exit. This injector inserts random values into the return register, producing logic errors where a program attempts to return undefined or forbidden status codes to the kernel.

Resource management violations. BPF programs using features like spinlocks must ensure they are properly released. This mechanism generates execution paths where critical sections are entered (for example, using `bpf_spin_lock`) but never exited, or where acquired references are dropped without release.

6.2.2. Generation strategy

To ensure a balanced distribution of verification errors, the generation process is managed by a heuristic control loop. The tool implements a strategy designed to prevent the most frequent error types from dominating the dataset, and obtain a balanced dataset, consisting of a balanced number of programs for each verifier error message. This process operates in two distinct phases:

1. Warm-up phase. The generator initially iterates through each injection mechanism individually for a fixed number of attempts. This phase allows the tool to associate which errors are triggered by which specific injector.

2. Adaptive feedback loop. Following the warm-up, the generator enters an adaptive cycle where injection probabilities are dynamically adjusted based on the error distribution.

- Dynamic Weighting: Each injector is assigned a base probability to be used(50%). This weight is dynamically increased (up to 90%) if the injector successfully triggers new or underrepresented errors, and penalized (down to 20%) if it fails to contribute effective samples.
- Error Saturation: To mitigate dataset imbalance, a saturation threshold is enforced (set to 21 samples per unique error message). Once an error type reaches this limit, subsequent instances producing that error are discarded, forcing the heuristic to explore alternative injection paths and uncover rarer edge cases.

This methodology produced a balanced dataset of 406 eBPF programs, serving as the ground truth for evaluation.

6.3. Evaluation metrics

To quantify the *Pretty Verifier*'s approach quality, we defined a composite scoring system. Unlike binary pass/fail metrics, we evaluated four distinct dimensions to capture the nuance of developer assistance. Each metric uses a specific scale to reflect the quality of the tool's output.

6.3.1. Correctness of reporting (line and file)

This objective metric determines whether the tool correctly identified the exact line number and file name where the error originated. The possible values are:

- 1 (Present and Correct): The tool identified the precise location.
- 0 (Not Present/Incorrect): The tool failed to identify the location or pointed to a wrong one.

6.3.2. Precision of C references

This metric assesses whether the tool correctly referenced the relevant C artifacts (variables, offsets, function names) associated with the error.

- 1.0 (Totally Precise): All references (e.g., variable names involved in an invalid access) were present and mapped correctly.
- 0.5 (Partially Precise): References were present but incomplete, or mapped with minor inaccuracies.
- 0.0 (Missing/Incorrect): References were missing or mapped to unrelated code elements.

6.3.3. Explanation quality

This metric evaluates the semantic clarity of the error message.

- 1.0 (Correct): The message correctly and clearly explained the root cause of the error
- 0.5 (Partial): The message gave a general explanation but lacked punctuality or specific context.
- 0.0 (Incorrect): The message was misleading or failed to explain the error.

6.3.4. Suggestion quality

This metric evaluates the actionable utility of the fix provided by the tool.

- 1.0 (Effective): The suggestion provided precise instructions that, if applied, fixed the error immediately.
- 0.5 (Partially Effective): The suggestion allowed fixing the error but additional effort or interpretation was required from the developer.
- 0.0 (Ineffective): The suggestion was missing or did not help fix the error.

In cases where a suggestion was not possible or required (for example whenever a certain function is not callable from a program type), the suggestion score was not included in the statistics to avoid penalizing the tool for impossible tasks.

6.4. Experimental results

The evaluation was conducted on the balanced dataset produced by the Fuzzing Framework, comprising 406 distinct eBPF programs. To ensure statistical significance, the generation process was controlled to produce approximately 21 samples for each unique verification error type, preventing common errors from skewing the results. Some possible eBPF verifier error messages could be produced by only a limited number of programs, but they represent only a small fraction of the overall set of error messages, and therefore their impact on the metrics is likely negligible.

Table 2
Summary of evaluation metrics (Percentages and counts)

Metric	Bad	Partial	Good	Avg
Line/File Correctness	15.59%	–	84.41%	0.80
Closeness to C	3.71%	26.73%	69.55%	0.81
Explanation Quality	2.72%	26.49%	70.79%	0.84
Suggestion Quality	14.07%	28.25%	57.41%	0.65

The evaluation was performed by the authors, who manually reviewed each error message produced by the programs generated by the fuzzer and then assigned a score according to the evaluation scale described above.

The overall results demonstrate the good quality of the approach, achieving a global average score of approximately 0.81 across all dimensions. Table 2 summarizes the distribution of the quality levels and the average scores (Avg) for all metrics.

6.4.1. Error location precision

The tool demonstrated high reliability in locating errors. The *Line and File Correctness* metric achieved a "Present and Correct" rating in 84.41% of cases. The remaining 15.59% of cases represent instances where the tool could not extract location information, despite being explicitly designed to handle location reporting for those error types. These failures primarily stem from inconsistencies in the underlying eBPF verifier log, which in some instances fails to correctly interleave the C source code within the log output. This behavior may be attributed to compiler optimizations that desynchronize the BTF debug information from the generated bytecode, preventing the tool from mapping the verification error back to the source line.

6.4.2. Closeness to C

Regarding the *Closeness to C*, which evaluates the mapping of bytecode to source artifacts, the tool yielded good results. In 69.55% of cases, the references were classified as "Totally Precise", while only 3.71% were deemed "Imprecise". It is worth noting that failures in the mechanism responsible for identifying the corresponding C source line where the error originated had a direct impact on this metric, negatively affecting its performance as well.

6.4.3. Explanation quality

The *Explanation Quality* was rated as "Correct" in 70.79% of cases. The incidence of inadequate or absent explanations is extremely low (2.72%), demonstrating the effectiveness of the tool in enriching the eBPF verifier output with some meaningful context for almost all the tested error messages. For the remaining messages that did not receive a fully adequate explanation, this was mainly due to the high complexity of the error type, which may depend on different factors that need to be investigated for each specific program.

6.4.4. Suggestion quality

The *Suggestion Quality* achieved an "Effective" rating in 57.41% of cases, providing precise instructions to fix the error. The average score of 0.65 indicates that while the tool typically provides the correct solution, complex scenarios occasionally result in "Partially Effective" suggestions that require further developer interpretation. However, the majority of errors were effectively addressed. Similarly to the previous metrics, in cases where a suggestion was not possible, it was not included in the statistics.

Table 3
Detailed scores for selected error types (Loc: Location, C-Ref: Closeness to C, Expl: Explanation, Sugg: Suggestion)

Error message	Loc	C-Ref	Expl	Sugg	Avg
Map configuration error	1.00	1.00	1.00	1.00	**1.00**
Wrong argument passed to helper function	1.00	1.00	1.00	0.64	**0.91**
Tail call invocation before releasing reference	0.83	0.75	0.83	0.83	**0.81**
Unbounded memory access of the variable	0.90	0.90	0.95	0.45	**0.80**
Invalid access to map value	1.00	0.55	0.95	0.50	**0.75**
32-bit ALU operations on pointers	1.00	0.50	0.50	0.00	**0.50**

```
R2 unbounded memory access, use 'var &= const' or 'if (var < con
st)'
processed 16 insns (limit 1000000) max_states_per_insn 0 total_s
tates 1 peak_states 1 mark_read 1

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

error: Unbounded memory access of the variable ''

Add ' &= const' or 'if ( < const)'
```

Fig. 14. Example of a failed attempt to enhance the verifier error.

6.4.5. Error-specific analysis

To understand the tool's behavior on specific error classes, we analyzed the detailed scores for distinct error codes using the balanced generated dataset. In Table 3, we present a breakdown of the metrics for some of the error messages obtained by the programs generated by the modified fuzzer.

For example, error handlers for "Wrong argument passed to helper function" and "Map configuration error" achieved near-perfect average scores of 0.91 and 1.00, respectively. Similarly, the detection of "Tail call invocation before releasing reference" achieved a score of 0.81. Since these error messages often follow similar patterns, the tool proves to be particularly effective in handling them.

Nevertheless, the tool also demonstrated strong robustness when dealing with error messages exhibiting more heterogeneous patterns. For instance, "Unbounded memory access of the variable" and "Invalid access to map value" maintained very high explanation-quality scores (0.95 for both), resulting in consistent average scores of 0.80 and 0.75, respectively. This suggests that, even when generating precise suggestions is challenging due to error heterogeneity, the tool remains effective in clarifying the underlying context of the reported issue.

In contrast, categories such as "32-bit ALU operations on pointers" showed a lower average score (0.50), mainly caused by the difficulty to generalize the suggestion for specific low-level arithmetic constraints. However, even in these cases, the tool successfully identified the location of the error in the majority of instances.

Finally, Fig. 14 shows an example in which a recognized error class is only partially handled. The verifier reports an unbounded memory access through register R2 and suggests constraining the involved variable. Pretty Verifier correctly matches the error class, but fails to recover the source-level variable responsible for the access. This happens because neither the verifier log nor the DWARF information dumped from the object file provide sufficient location information for the failing instruction, preventing the tool from mapping R2 back to the corresponding C variable. As a result, the diagnostic contains an empty variable name and the generated suggestion is incomplete. This case is therefore evaluated negatively for the Closeness-to-C and Suggestion Quality metrics, while still showing that the underlying verifier error was correctly classified.

6.4.6. Error message analysis for manually collected eBPF programs

To complement the evaluation performed on the fuzzer-generated dataset, we conducted a similar analysis on the real-world and manually written eBPF test programs dataset that we created initially, comprising faulty programs collected from online repositories, developer forums, blog posts, and specifically crafted manual tests designed to replicate the rejection messages studied on the eBPF verifier source code.

Table 4
Summary of evaluation metrics for real-world and manually crafted programs.

Metric	Bad	Partial	Good
Line/File Correctness	2.70%	–	97.30%
Closeness to C	0.00%	19.44%	80.56%
Explanation Quality	0.00%	2.33%	97.67%
Suggestion Quality	3.85%	30.77%	65.38%

We kept the analysis of this dataset separate to prevent the uneven distribution of real-world errors from skewing the balanced metrics discussed earlier. While the fuzzer produces a perfectly balanced dataset to avoid statistical bias across error types, the human-written dataset contains only one or a few programs for each manually replicated error message. For this reason, merging the two evaluations would have introduced distortions in the results. The evaluation metrics for the human-written dataset are summarized in Table 4.

Comparing these results with those obtained from the fuzzed dataset (Table 2), we observe a performance improvement across all dimensions. Most notably, the line correctness achieved a "Good" rating in 97.30% of cases, compared to 84.41% in the automated dataset. Similarly, explanation quality reached 97.67% (up from 70.79%), and suggestion quality improved to 65.38% (up from 57.41%).

This discrepancy is expected and highlights the high resilience of the tool when applied to standard development workflows. The fuzzer systematically generates edge-case semantic combinations and structurally chaotic execution paths to stress-test the verifier. These extreme configurations often degrade the quality of the DWARF debug information emitted by the compiler, leading to the localization failures observed in the fuzzed dataset. In contrast, programs written by humans, even when containing verifier-rejectable faults, maintain coherent logical structures, allowing the compiler to preserve accurate debug mappings.

6.5. Error message coverage

By combining the test cases generated by the fuzzer with those previously collected from online sources, forum discussions, and manually written tests, we were able to test 44 out of the 91 handlers implemented in Pretty Verifier. In the previous iteration, only 17 handlers had been tested. Considering the 111 error messages identified in the eBPF verifier source code as potentially originating from C code, Pretty Verifier now covers 56 of them, compared with the 17 covered in the previous iteration.

Although testing 48% of the handlers may appear limited, it is important to note that our initial study conducted to identify the errors to be managed and the 91 handlers to develop was highly conservative, including errors whose relevance we could not exclude at that stage but that subsequently were revealed to be impossible to trigger from C code, and then not relevant. Such error messages were handled proactively for completeness, as a defensive design choice, but they are impossible to generate starting from C programs for a number of reasons. The main one is that the compiler prevents the creation of certain malformed eBPF bytecode patterns that are necessary to trigger those errors or the verifier itself may preemptively resolve or mask the issue. An example is dead code elimination: the compiler removes unreachable instructions and therefore the unreachable instruction error that the verifier is expected to return cannot be triggered by compiled code. Another example are explicit casts between integer types: some type conversions written in C are not preserved in the corresponding bytecode by the compiler, so preventing the bytecode generated by the compiler from including some cast-related verifier errors.

Although our fuzzing strategy relies, as previously discussed, on an inherently heuristic approach, and therefore cannot be used to definitively infer that the untriggered handlers cannot be triggered by C code, our evaluation is supported by combining multiple independent testing efforts. Specifically, we complemented automated fuzzing with manual testing and an extensive review of real-world eBPF issues. Our analysis of developer forums and online repositories suggests that programmers encounter only a quite restricted subset of verifier errors in practice, all covered by our tests. The absence of the remaining errors in these real-world discussions strongly supports the hypothesis that they cannot be generated through standard C-to-eBPF compilation. Based on these observations, we can assume that Pretty Verifier successfully handles the vast majority of the error messages that are realistically triggerable by C programs. Moreover, through the fuzzer's exploration, we generated test cases for these practical errors, reaching full test coverage for this particular subset of handlers, which translates to improved reliability in the tool's error handling.

6.6. Functional validation and performance

Beyond validating as many handlers as possible, specific tests were conducted to evaluate the tool's functional robustness and efficiency.

In order to ensure complete validation, multiple test cases were developed for each triggered handler, achieving full coverage of the handler code. In particular, the fuzzer proved effective in generating diverse scenarios for each handler, allowing all execution paths and supported cases to be tested. To further assess the tool's resilience in accurately identifying the correct C source line, we employed both automated and manual stress-testing strategies. An automated utility was developed to inject minor, non-functional modifications, such as random newlines or `bpf_kprint` calls, into the source code of the developed test cases. The tool consistently preserved correct line number reporting for all errors across all generated variants. Additionally, the tool was tested against deliberate structural complexities, such as code distributed across multiple `.c` and `.h` files and the introduction of duplicated code segments (e.g., identical functions called in different contexts). In this scenario, the tool generally produced correct results, but had difficulties when the debug information associated with verifier error messages was corrupted, usually because of compiler optimizations. This issue was then mitigated by implementing the fallback mechanism described in Section 5.5, which lets the tool provide correct suggestions even in such cases.

In terms of performance, the tool operates exclusively at load time, introducing zero runtime or compile-time overhead. The analysis overhead during the loading phase is negligible compared to the eBPF verifier execution time.

6.7. Robustness across kernel versions

Pretty Verifier was primarily developed and evaluated on Linux 6.8. To assess its robustness across nearby kernel versions, we ran the same 406 fuzz-generated tests used in the main evaluation on Linux 6.6.143, 6.12.94, and 6.18.36. For each version, we compared the raw verifier error and the generated diagnostic with the corresponding baseline result.

Table 5 summarizes the results. In more than 80% of the cases, both messages are identical to the baseline. This shows that, for the large majority of programs, changing the kernel version does not affect the behavior observed by the tool.

The diagnostic is emitted in an even larger fraction of cases, ranging from 84.73% to 91.13%. These additional tests correspond to cases where the tool still handles the verifier error, although the resulting diagnostic is not identical to the baseline. These cases remain limited, ranging from 3.20% to 3.45%. They are mainly caused by differences in the verifier behavior across kernel versions: the verifier may stop at different points of the bytecode analysis and therefore expose a different reason for rejecting the same source program. Since these errors are still recognized by Pretty Verifier, their diagnostic quality is covered by the evaluation presented in the previous sections.

Table 5
Cross-kernel comparison with respect to the Linux 6.8 baseline. *Diagnostic emitted* includes all cases in which the tool produced a diagnostic, including identical ones.

Kernel	Identical messages	Diagnostic emitted	No diagnostic
6.6.143	357 (87.93%)	370 (91.13%)	36 (8.87%)
6.12.94	350 (86.21%)	364 (89.66%)	42 (10.34%)
6.18.36	330 (81.28%)	344 (84.73%)	62 (15.27%)

The remaining cases correspond to executions for which no diagnostic is generated. This may happen when the program is rejected before a meaningful verifier log is available, for example due to BTF or kernel-compatibility issues, or when the verifier emits a message whose textual structure differs from the one observed in the baseline. The latter cases mostly require extending the parser to support additional formatting variants of already known verifier errors, and are left as future work.

6.8. Possible validity limitations

While the experimental results demonstrate the effectiveness of Pretty Verifier, we acknowledge certain possible limitations to the validity of our evaluation.

A primary threat to internal validity lies in the subjective nature of some of the evaluation metrics, specifically *Explanation Quality* and *Suggestion Quality*. Although we defined strict criteria for assigning scores (1.0, 0.5, 0.0), the assessment was inherently conducted by the authors, introducing a potential interpretative bias. To mitigate this and provide a purely objective assessment, a comprehensive user study involving external eBPF developers is planned as future work.

Moreover, the generalizability of our results is subject to constraints related to the compiler, the testing dataset, and the kernel version.

First, while our heuristic approach combined with manual review suggests that the most prevalent practical errors are all covered, the inability to formally prove that the untested error messages cannot be generated from C source code remains a limitation.

7. Conclusions and future developments

The *Pretty Verifier* has been designed to improve the developer experience for eBPF programming, specifically addressing the complexity and interpretation difficulty of the eBPF verifier logs. The tool provides developers with critical insights into the issues raised by the verifier, offering precise error localization, code-referenced explanations, and actionable suggestions. The experimental results are generally satisfactory. By leveraging a semantic fuzzer alongside manual test cases, we validated 44 of the 91 implemented handlers. While this coverage extends to approximately half of the total set, the fuzzing campaign confirms that the tool effectively addresses the errors realistically triggered in practice.

Future work will focus on enhancing the tool's evaluation. A primary objective is the evolution of the testing framework through the adoption of hybrid generation techniques. By relaxing the strict semantic constraints of the current fuzzer, it will be possible to generate basic C code patterns that expose verification edge cases currently masked by the semantic enforcement. Furthermore, a comprehensive user study involving eBPF developers is planned to complement the automated evaluation. This qualitative analysis will aim to empirically assess the tool's effectiveness in reducing debugging time and improving error comprehension in real-world development scenarios. Finally, porting the codebase to a compiled language is planned to eliminate Python runtime dependencies. This transition would facilitate the distribution of the tool as a standalone binary, in order to facilitate its deployment in production environments.

CRediT authorship contribution statement

Rosario Rizza: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Conceptualization. **Riccardo Sisto:** Writing – review & editing, Supervision, Methodology, Funding acquisition, Conceptualization. **Fulvio Valenza:** Supervision, Methodology.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Riccardo Sisto reports financial support was provided by European Commission. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This paper has received funding by the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101139067. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

Data availability

The code developed for the article is at https://github.com/netgroup-polito/pretty-verifier.

References

[1] H.-W. Hung, A. Amiri Sani, BRF: Fuzzing the eBPF runtime, Proc. ACM Softw. Eng. 1 (FSE) (2024) http://dx.doi.org/10.1145/3643778.

[2] R. Rizza, R. Sisto, F. Valenza, Design and implementation of a tool to improve error reporting for ebpf code, in: 2025 IEEE International Conference on Cyber Security and Resilience, CSR, 2025, pp. 214–219, http://dx.doi.org/10.1109/CSR64739.2025.11130075.

[3] Linus Torvalds and Linux Kernel Contributors, BPF verifier source code, 2023, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/bpf/verifier.c?h=v6.8. (Accessed: 24 April 2026).

[4] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J.A. Navas, N. Rinetzky, L. Ryzhyk, M. Sagiv, Simple and precise static analysis of untrusted linux kernel extensions, in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, in: PLDI 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1069—-1084, http://dx.doi.org/10.1145/3314221.3314590.

[5] J. Jia, R. Sahu, A. Oswald, D. Williams, M.V. Le, T. Xu, Kernel extension verification is untenable, in: Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 150—-157, http://dx.doi.org/10.1145/3593856.3595892.

[6] M.H.N. Mohamed, X. Wang, B. Ravindran, Understanding the security of linux eBPF subsystem, in: Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 87—-92, http://dx.doi.org/10.1145/3609510.3609822.

[7] MITRE Corporation, CVE search results for eBPF, 2025, https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ebpf. (Accessed: 24 April 2026).

[8] H. Vishwanathan, M. Shachnai, S. Narayana, S. Nagarakatte, Verifying the verifier: eBPF range analysis verification, in: Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part III, Springer-Verlag, Berlin, Heidelberg, 2023, pp. 226—-251, http://dx.doi.org/10.1007/978-3-031-37709-9_12.

[9] Q. Liu, W. Shen, J. Zhou, Z. Zhang, J. Hu, S. Ni, K. Lu, R. Chang, Interp-flow hijacking: Launching non-control data attack via hijacking eBPF interpretation flow, in: J. Garcia-Alfaro, R. Kozik, M. Choraś, S. Katsikas (Eds.), Computer Security – ESORICS 2024, Springer Nature Switzerland, Cham, 2024, pp. 194–214.

[10] T. Lyu, K.K. Dwivedi, T. Bourgeat, M. Payer, M. Xu, S. Kashyap, eBPF misbehavior detection: Fuzzing with a specification-based oracle, in: Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP '25, Association for Computing Machinery, New York, NY, USA, 2025, pp. 701—-718, http://dx.doi.org/10.1145/3731569.3764797.

[11] I. Solodrai, J. Rome, BPF verifier visualizer (bpfvv), 2025, Presented at Linux Plumbers Conference (LPC). Available at https://github.com/libbpf/bpfvv.

[12] M. Deokar, J. Men, L. Castanheira, A. Bhardwaj, T.A. Benson, An empirical study on the challenges of eBPF application development, in: Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions, eBPF '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 1—-8, http://dx.doi.org/10.1145/3672197.3673429.

[13] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, C. Parnin, Do developers read compiler error messages? in: 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE, 2017, pp. 575–585, http://dx.doi.org/10.1109/ICSE.2017.59.

[14] B. Johnson, Y. Song, E. Murphy-Hill, R. Bowdidge, Why don't software developers use static analysis tools to find bugs? in: 2013 35th International Conference on Software Engineering, ICSE, 2013, pp. 672–681, http://dx.doi.org/10.1109/ICSE.2013.6606613.

[15] C. Peng, M. Jiang, L. Wu, Y. Zhou, Toss a fault to BpfChecker: Revealing implementation flaws for ebpf runtimes with differential fuzzing, in: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 3928—-3942, http://dx.doi.org/10.1145/3658644.3690237.

[16] L. Rice, Learning EBPF: Programming the linux kernel for enhanced observability, networking, and security, O'Reilly Media, Inc, Sebastopol, CA, 2023.

[17] Netgroup-Polito, Pretty verifier, 2025, https://github.com/netgroup-polito/pretty-verifier.

[18] L. Rice, Learning eBPF, 2023, GitHub repository. https://github.com/lizrice/learning-ebpf.

[19] Anteon Blog, Unveiling eBPF verifier errors, 2023, https://getanteon.com/blog/unveiling-ebpf-verifier-errors/. (Accessed: 24 April 2026).