POLITECNICO DI TORINO
Repository ISTITUZIONALE

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

Original
Design and implementation of a tool to improve error reporting for eBPF code / Rizza, R., Sisto, R., Valenza, F.. - ELETTRONICO. - (2025), pp. 214-219. (2025 IEEE International Conference on Cyber Security and Resilience (CSR) Crete (GR) 04-06 August 2025) [10.1109/csr64739.2025.11130075].

Availability:
This version is available at: 11583/3002569 since: 2025-08-27T09:56:53Z

Publisher:
IEEE

Published
DOI:10.1109/csr64739.2025.11130075

Terms of use:

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

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2025 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

16 October 2026

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

Rosario Rizza
DAUIN, Politecnico di Torino, Italy
rosario.rizza@polito.it

Riccardo Sisto
DAUIN, Politecnico di Torino, Italy
riccardo.sisto@polito.it

Fulvio Valenza
DAUIN, Politecnico di Torino, Italy
fulvio.valenza@polito.it

***Abstract*—eBPF is a rising trend in cloud computing, enabling user-defined programs to run in kernel space. This allows greater system control, especially in security and performance sensitive environments, like server operating systems, enhancing monitoring and observability. However, running user-defined programs into the kernel is a security risk, which is attempted to be mitigated by the eBPF verifier, a set of deep checks that identify and reject dangerous programs, leading to the kernel crash or, even worse, escalate privileges, leak sensitive data, or take control of the system. However, messages produced by the verifier are difficult to understand, and usually detached from the source code. This paper presents a tool designed to improve the developer's experience by introducing readability improvements and explanations into the eBPF compilation pipeline, allowing developers to easily identify the line of C code that caused the error, to understand the issue and how to fix it.**

I. 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 the safe execution or the security of the kernel. 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. 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, or the libbpf library, outputs the eBPF verifier log, which communicates if 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, then followed by the actual eBPF verifier log: it contains the instruction of the program until the error is detected with the corresponding C code and the eBPF verifier state information both as comments, then following with the error message identifying the error and some other state information. The difficulty of interpretation of the verifier log is a limitation that makes fixing the security 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, which is the most used language for this purpose, and guiding the developer to understand and fix the issues that might compromise the safety and the security of the system, thus enhancing developer ability in fixing the insecure aspects of the code. To achieve this, we developed a command-line utility, *pretty verifier*, which parses eBPF verifier logs and analyzes the object code generated by Clang. It then maps the errors back to the original C source and provides guidance to help understand and fix them.

To design this tool, a preliminary full study of the eBPF verifier source code was performed to gather the information about which error messages should be managed. Some difficulties were encountered, documented in the paper, but in the end a decently satisfying result was obtained.

In section II, we first give an overview of the related works. In the following section III, we list the main features of the eBPF verifier. Then, our study of the eBPF verifier is discussed in section IV. Next, in section V, 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 VI, while the conclusions and future developments are exposed in section VII.

```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

```
bpftool prog load max_value_is_outside_map_value.bpf.o /sys/fs
/bpf/test
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 = 0xffffa9a600d4e000       ; R2_w=map_value(map=max_
valu.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=
max_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_val
ue(map=max_valu.data,ks=4,vs=10)
5: (67) r1 <<= 32                     ; R1_w=scalar(smax=0x7fff
ffff00000000,umax=0xffffffff00000000,smin32=0,smax32=umax32=0,v
ar_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 = 0xffffa9a600d4e004       ; R2_w=map_value(map=max_
valu.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
(map=max_valu.data,ks=4,vs=10,off=4,smin=smin32=0,smax=umax=sma
x32=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_
states 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.b
pf.o'
Error: failed to load object file
```

Fig. 2. Log of the eBPF verifier for incorrect memory access

II. Related Work

Current research on eBPF security focuses mainly on the eBPF verifier [4] [15] [7] [6], which has grown to more than 20k lines of code [14] and has been the concern of numerous recent CVEs [9] [2]. These problems are compounded by the difficulties encountered by programmers in developing eBPF programs, having particular trouble with the eBPF verifier strictness. There have been attempts in improving this process [8], as well as in reporting the community response to the eBPF technology from a development perspective, studying the most asked questions about the topic on the popular programming forum Stack Overflow [3].

We noticed that the approach we employed 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 in the literature so far. However, the eBPF verifier errors have been employed in previous works [11] [5] to detect the error type and improve the acceptance rate of input programs during fuzz testing of the eBPF verifier. In particular, [5] presents a study that analyzed the eBPF verifier source code in a way similar to what presented in section IVbut to identify semantically related error messages. In contrast, our focus was primarily on the connection to the original C code.

III. The eBPF verifier

The verifier is the backbone upon which the eBPF technology managed to be widely adopted in production applications, since it provides safety and security of the loaded programs.

A. The 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 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 then checks for instructions that could lead to kernel crashes, information leakage, or other safety and security issues.

B. Verification criteria

The eBPF verifier follows some verification criteria in order to ensure correctness of the program and prevent security issues it might cause. Liz Rice, in the ”Learning eBPF” book [13], 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.

TABLE I
VERBOSE() CALLS IN THE EBPF VERIFIER (LINUX 6.8 LTS)

Category	Count
Log messages	54
Internal errors	97
Bytecode errors	123
Other errors	176
Managed errors	79
Total	**529**

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.

IV. STUDY OF THE EBPF VERIFIER

In order to develop the tool, a deep study of the eBPF verifier code was conducted, looking for all the possible error outputs it may return. The eBPF verifier [14] emits numerous error messages, logs, and warnings, all printed through the `verbose` function and wrappers. Other functions used to print messages in the output of the eBPF verifier are the `bpf_log` function, used for logging purposes, and the `WARN_ONCE` function, used in a context where just a warning 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 6.8 LTS exceeds 500 (Table I). 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 where calls to the function do not correspond to errors found in the source code.

A. 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 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 uses of the `verbose` function to log messages were found.

B. 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 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 eBPF verifier's own code. However, there are cases where this formatting is not respected. Of course, 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 uses of the `verbose` function to output internal error messages were found.

C. Errors not related to the C code

Among the remaining cases, which are caused by errors in the bytecode passed to the eBPF verifier, there are two situations, detailed in the next two sections, where the error is not directly caused by the original C source code.

D. Bytecode errors

Many error messages that can be emitted by the eBPF 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, which should not occur. In these situations, explaining the error would not significantly help the developer in resolving it. A total of 123 error messages of this type were found.

E. Other errors

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). 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) do not refer to code defects. Lastly, several errors are extremely rare, and limited documentation exists online 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 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 number of error messages of this type, plus the ones caused by BTF and configuration errors of the OS, are 176.

V. PRETTY VERIFIER

A. Tool Architecture and Features

The Pretty Verifier tool was specifically developed to assist programmers writing eBPF programs for the Linux kernel, particularly targeting the error messages generated by kernel version 6.8, the latest LTS at the time of development. It is available in the Netgroup-Polito Github [10]. The tool is developed in Python and uses the `libbpf` and `bpftool` utilities to obtain the eBPF verifier log, and relies on the compilation of the C source code with `Clang`. Its architecture is shown in Fig. 3. After obtaining the bytecode object resulting from

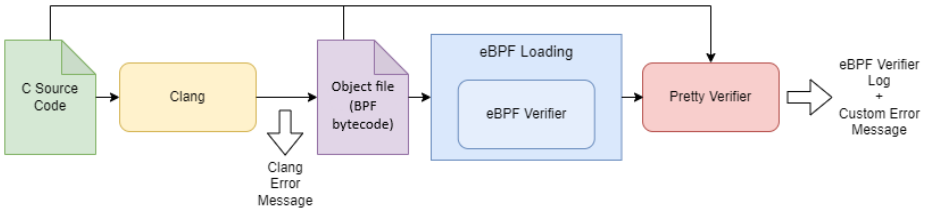

Fig. 3. Pretty Verifier role in the eBPF development pipeline

the Clang compilation, the tool works on the result of the verification, the C source code and the bytecode itself and outputs the new message right after the original eBPF verifier log, as shown in Fig. 4, where the eBPF verifier log is the initial part, while the message addeed by our tool is the one starting from the cyan "Pretty Verifier" string. In contrast, without our tool, the eBPF verifier log for the same program is the one in Fig. 2. For the application to function correctly, the source code must be compiled with the `-g` option, which allows the debug information to be included into the compiled object.

The core functionality of the application involves processing the output of the eBPF verifier and generating extended error messages. Such messages follow a structured format, including 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**: This section provides additional detail for certain error messages that require more context.
- **Suggestion**: This section provides suggestions to help resolve the issue.

B. Methodology

The tool scans the verifier output looking 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. If an error message does not match any known patterns, the tool outputs a default message of "error not managed", signaling the necessity to implement a handler for this case scenario, making the tool easy to extend.

The study of the eBPF verifier source code described in section IV identified 79 distinct error messages related to defects of the input C source code that the tool should manage. Instead, the other errors are irrelevant, as explained in IV. To assess the impact of the 79 selected errors on the developer experience, we conducted a Google search to determine the frequency of each error in developer discussions. We found that only a restricted subset of these errors was reported in online forums, while the others were either completely absent or had only one or two mentions. For many of the infrequent errors, it was also challenging, or sometimes even impossible, to build a corresponding C code that would trigger them. In some cases, this is due to compiler optimizations. For example, the division by zero error cannot be generated because the Clang compiler preemptively avoids the issue by assigning a placeholder value as the result. However, for completeness, we included all outliers among the most frequently searched eBPF verifier error messages, even when no corresponding source code could be constructed. This was decided to have full coverage of the eBPF verifier possible error messages related to the C code, hence restricting the possibility of having an error not managed occurrence only when the error is not related to the source code.

For each one of the above mentioned 79 error messages the output was crafted, aiming at connecting the error to the C source code counterpart and easing the debugging process.

C. Utility and Impact

The additional information included in the output enhances error logs by providing detailed insights tailored to specific error messages. These enhancements are designed to make the logs more accessible and actionable, even for developers who may not be familiar with kernel source code. One key improvement is the translation of technical terms used by the eBPF verifier—often represented as C enumerations—into plain, human-readable text. For instance, translating the type of a BPF register into a descriptive format, leveraging the `reg_type` enumeration from the bpf.h file in the kernel source. The enhanced logs also incorporate explanations for checks performed by the verifier, derived from comments in the eBPF verifier source code and references to online forums that discuss common pitfalls and their resolutions. For example, when a division is performed on a pointer, the eBPF verifier just outputs the message "pointer arithmetic with /= operator prohibited". However, as only additions and subtractions are allowed, we provide this information in the enhanced error message, as shown in Fig. 5, giving the developer the precise verification criteria. Specific error types receive tailored enhancements to further help developers. For example, when the eBPF verifier detects that a register's type does not align with a helper function's expected signature, the tool identifies the relative variable in the original C code, as shown in Fig. 6. Similarly, for one of the most important security vulnerabilities in C code, i.e. invalid memory accesses due to variables that have not been correctly bound checked before

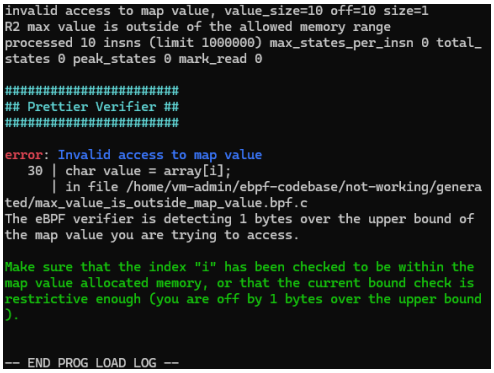

Fig. 4. Message of the Pretty Verifier for incorrect memory access

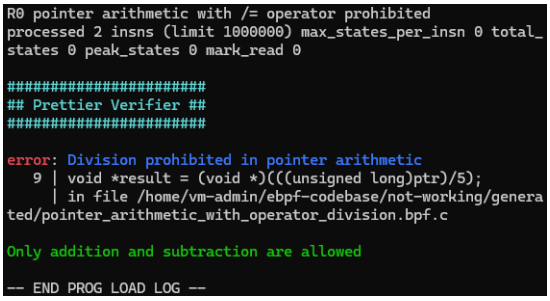

Fig. 5. Message of the Pretty Verifier for unauthorized pointer arithmetic

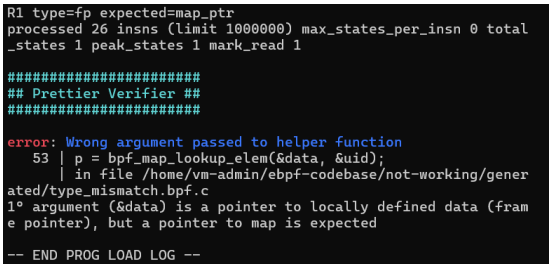

Fig. 6. Message of the Pretty Verifier for type mismatch in an helper function call

and that can lead to an exploitable buffer overflow, as in the C code in Fig. 1, the tool calculates the correct size required for the bound check and highlights it in the output (Fig. 4). Therefore the developer gains a better understanding of why the eBPF verification failed and is helped by the tool to fix the error. In addition to these targeted improvements, the tool also provides contextual information such as the exact line number in the source code where the error occurred and actionable suggestions for resolving the issue. These enhancements can improve the debugging process, reducing the time required to effectively address errors and providing insights about the security checks performed by the eBPF verifier.

VI. Results and Optimizations

In order to demonstrate the reliability and effectiveness of the tool, an extensive series of tests were conducted. Despite the availability of eBPF programs already including errors is quite limited, a few notable resources provided valuable starting points. For instance, Liz Rice's "Learning eBPF" GitHub repository [12] contains a handful of examples featuring faulty eBPF code. Additionally, the Anteon blog post [1] presented further examples of verifier errors and misbehaviors. However, these resources were not sufficient to cover all the error messages handled by the tool. Moreover, the tool is being adopted by some computer engineering students for developing eBPF code, in order to assess usability and effectiveness. In terms of performance, the tool is applied at load time and introduces no runtime or compile-time overhead. The overhead during the loading phase is negligible, given the linear complexity of Pretty Verifier compared to the exponential complexity of the eBPF verifier.

A. Writing of test cases

To address this issue, additional faulty eBPF C code test cases where developed to cover as many error scenarios as possible. The ideal goal was to test and validate all the handlers implemented in the tool. However, as mentioned in section V-B, even if handlers were created for all the errors that might be caused by C source code errors, for several of them either it is impossible to create C code that triggers them or so far we did not manage to do it. Only 17 different error messages could be triggered using the C code provided by the faulted eBPF repositories and in online forums where developers asked for help, and the test cases we created ourselves. For the errors that could be triggered, multiple test cases were created, with the goal of achieving full coverage for all the tested handlers, which we achieved.

B. Automatic testing

To further expand the scope of testing, a utility for generating additional test cases was developed. This utility aimed to evaluate the tool's robustness in identifying the correct location of errors in the C code. The test cases are generated by the utility by introducing specific modifications to the code, such as inserting new lines or injecting `bpf_kprint` calls at random locations. These manipulations had the potential to produce erroneous or misleading C lines in the tool's output messages. The results of these tests were promising, as the tool demonstrated consistent correctness even with a growing number of test cases.

C. Manual testing

In addition to these standard tests, the resilience of the tool was evaluated against deliberate code manipulations designed to stress specific aspects of its functionality. For instance, duplicated lines within the code were introduced to observe their impact on the tool's reporting of the C line responsible for an error. This was achieved through the use of functions injected into the C code that duplicated lines from their caller function. Similarly, tests were conducted with code split across multiple `.c` and `.h` files to assess whether the tool could still accurately identify the correct C line in such cases, responding positively.

D. Full Mode

During these tests, certain challenges were encountered. Specifically, some issues arose 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 mode, called *full mode*, was developed. This mode 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, since it would be used outside the eBPF pipeline, but instead of being invoked appended to the eBPF verifier invocation, it can be directly called just by passing the source C code. This obviously limits its use to programs that do not strictly rely on the user space counterpart.

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.

VII. 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 referred to safety and security issues in the eBPF code. The tool gives the developer insights about the issues raised by the verifier, including precise error location in the code, error explanation with code references, and suggestions to fix the error.The results are generally satisfactory, with a good response to all the common scenarios in which the eBPF verifier rejects the input code. Although the tool is primarily designed to improve developer experience, it also contributes to safer eBPF development by helping developers avoid security-critical mistakes such as invalid memory access or unsafe helper usage. Its integration into development workflows can enhance security by providing early detection and explanation of potentially exploitable verifier rejections. However, some future developments can benefit the tool's current state. First of all, a deeper study of the error cases for which C code that triggers them could not be generated should be done. Moreover, the testing phase can be extended by increasing the number of test cases specifically designed for the handlers, thereby covering a broader range of scenarios and edge cases. Additionally, the evaluation can be further enriched by embedding the faulty C code into larger, fully functional C programs. This approach will help assess how the tool behaves in more complex and realistic environments, providing valuable insight into its robustness, reliability, and ability to detect faults effectively.

VIII. 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.

References

[1] Anteon Blog. Unveiling ebpf verifier errors. https://getanteon.com/blog/unveiling-ebpf-verifier-errors/, 2023. Accessed: October 16, 2024.

[2] MITRE Corporation. Cve search results for ebpf. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ebpf, 2025. Accessed: 2025-01-23.

[3] Mugdha Deokar, Jingyang Men, Lucas Castanheira, Ayush Bhardwaj, and Theophilus 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, page 1–8, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly 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*, PLDI 2019, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

[5] Hsin-Wei Hung and Ardalan Amiri Sani. Brf: Fuzzing the ebpf runtime. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024.

[6] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 150–157, New York, NY, USA, 2023. Association for Computing Machinery.

[7] Qirui Liu, Wenbo Shen, Jinmeng Zhou, Zhuoruo Zhang, Jiayi Hu, Shukai Ni, Kangjie Lu, and Rui Chang. Interp-flow hijacking: Launching non-control data attack via hijacking ebpf interpretation flow. In Joaquin Garcia-Alfaro, Rafał Kozik, Michał Choraś, and Sokratis Katsikas, editors, *Computer Security – ESORICS 2024*, pages 194–214, Cham, 2024. Springer Nature Switzerland.

[8] Andrea Mayer, Lorenzo Bracciale, Paolo Lungaroni, Giulio Sidoretti, Stefano Salsano, Giuseppe Bianchi, and Pierpaolo Loreti. Composing ebpf programs made easy with hike and eclat. *IEEE Transactions on Network and Service Management*, 21(2):1359–1371, 2024.

[9] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. Understanding the security of linux ebpf subsystem. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '23, page 87–92, New York, NY, USA, 2023. Association for Computing Machinery.

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

[11] Chaoyuan Peng, Muhui Jiang, Lei Wu, and Yajin 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, page 3928–3942, New York, NY, USA, 2024. Association for Computing Machinery.

[12] Liz Rice. Learning ebpf. https://github.com/lizrice/learning-ebpf, 2023. GitHub repository.

[13] Liz Rice. *Learning EBPF: Programming the linux kernel for Enhanced Observability, networking, and security*. O'Reilly Media, Inc, Sebastopol, CA, 2023.

[14] Linus Torvalds and Linux Kernel Contributors. Bpf verifier source code. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/bpf/verifier.c?h=v6.8, 2023. Accessed: 2025-01-23.

[15] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh 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*, page 226–251, Berlin, Heidelberg, 2023. Springer-Verlag.