Unsafe kernel extension composition via BPF program nesting

Siddharth Chintamaneni
Virginia Tech
Blacksburg, VA, USA
sidchintamaneni@vt.edu

Sai Roop Somaraju
Virginia Tech
Blacksburg, VA, USA
sairoop@vt.edu

Dan Williams
Virginia Tech
Blacksburg, VA, USA
djwillia@vt.edu

ABSTRACT

BPF programs provide the ability to extend the kernel while ensuring safety. The safety guarantees are provided by the in-kernel verifier. However, the verification guarantees may not hold when multiple BPF programs interact with each other through helper functions. This is because, while verifying a BPF program, the verifier considers each BPF program as an individual unit rather than part of a composite system. One aspect affected by this unsafe composition is the kernel stack. In this paper, we highlight how different possible nesting scenarios can affect the safety of the kernel stack. To address this problem, we propose a helper-rooted callgraph-based approach, which enables the verifier to have a global view of the system. By using the callgraph and maximum stack depth information during verification, the verifier will either accept or reject a program by considering all the possible nesting scenarios, which ensures runtime stack safety.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**;

KEYWORDS

eBPF, dynamic tracing, callgraph

ACM Reference Format:
Siddharth Chintamaneni, Sai Roop Somaraju, and Dan Williams. 2024. Unsafe kernel extension composition via BPF program nesting. In *Workshop on eBPF and Kernel Extensions (eBPF '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3672197.3673440

1 BACKGROUND AND MOTIVATION

In this section, we provide background on BPF and its verifier, and then demonstrate unsafe BPF composition by considering the stack as a running example.

1.1 Overview of BPF programs and Verifier

BPF programs are extensions to the kernel that provide the flexibility to attach code from userspace at runtime. These programs are attached to the kernel through hook points. The hooks get triggered when they are in the kernel execution path. The BPF verifier plays an important role by verifying a BPF program and ensuring safety to the kernel. After verification, the verified BPF instructions are directly JIT compiled into machine instructions.

BPF programs serve a wide range of use cases like tracing, networking, security modules, etc. To facilitate these usecases there is a need for BPF programs to interact with the kernel. For this interaction BPF programs use helper functions (stable between releases) or kfuncs (unstable) as an interface to the kernel.

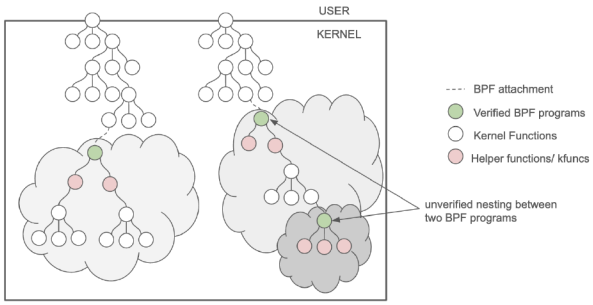

Figure 1: Nested BPF programs Overview

During the verification step, the verifier assumes BPF programs' interactions with helper functions are safe. Though it performs type checking, memory access validation, and resource allocation/release checking on the bytecode, the verifier lacks the ability to see what is happening inside the helper functions. In the case of stack usage, the verifier only restricts BPF programs from exceeding more than 512 bytes. However, helper functions' stack usage is unaccounted for. So, with a sufficient amount of nesting, the kernel stack can be overflowed. To address this issue, kernel developers [11] recently developed an approach to allocate BPF local variables on the heap[1]. This approach might delay the overflow, but not completely prevent it.

1.2 The Fundamental lack of a global view

The BPF verifier operates under a limited scope during the verification process, which is supported by certain assumptions about the execution environment that BPF programs interact with. However, these assumptions and the verifier's limited scope prove [9] to be insufficient when dealing with the nesting of BPF programs.

In the kernel, multiple BPF programs can be nested through helper functions. For example, as shown in Figure 1, BPF program 1 can call a helper function, which then calls a function to which BPF program 2 is attached. The verifier's limited scope and assumptions do not adequately consider the complexities introduced by these nesting scenarios. In the stack example this leads to an overflow, as the verifier fails to account for the cumulative stack usage resulting from nested program executions. We next explore how helper-rooted callgraphs can help the verifier in achieving a global scope.

[1]Discussed in detail in Section 3

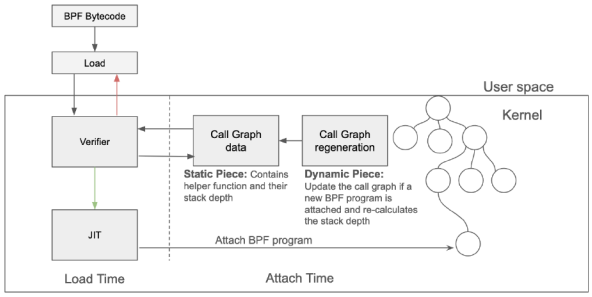

Figure 2: Proposed design overview

2 CALLGRAPH BASED SOLUTION

In this section, we define helper-rooted callgraphs, describe how they will be utilized by the BPF verifier to solve stack overflow issue and provide some hypothesis on using these callgraphs in the context of design.

2.1 Helper-rooted Callgraphs

Helper-rooted callgraphs consist of helper functions as root nodes and functions callable from these helpers as nodes in the graph. This callgraph can be used to identify nesting scenarios at load time. For example, if a BPF program is attached to a helper function or any of the functions called by that helper (callee's), then the callgraph data will be updated with the stack depths utilized by the BPF program during attachment time. The verifier will use this updated information to determine whether a BPF program is attaching directly to a helper or to a function inside these helpers, representing a nesting scenario, as shown in Figure 2.

The callgraph design consists of two components: static and dynamic. The static component refers to all the helper functions and their respective stack sizes, which are defined by the kernel code. The dynamic component refers to updating the call graph when a new BPF program is attached to a helper function's call graph, as this can change the maximum stack size when that helper function is used.

2.2 Discussion on generated callgraphs

Generating the call graph for the entire kernel is a challenging task. However, generating a call graph for a subset of functions, such as helper functions, should be relatively straightforward. There are challenges in handling both the static and dynamic pieces. The static piece has to handle indirect calls and loops within the graph. While loops are beyond the scope of this project, we propose addressing indirect calls using type matching [6], which predicts potential targets for indirect call sites. This method improves the accuracy of our call graph by providing a clearer picture of possible function calls and maximum stack depth at runtime. For the dynamic piece, the callgraph should be recalculated based on runtime attachments. Multiple cases of nesting are possible in BPF, such as a BPF program being stacked on top of other BPF programs, plugged in the middle between the programs, or as a leaf. The dynamic piece has to handle all three possible nesting scenarios and recalculate the stack depths accordingly.

Among the 278 helper functions in the kernel, some are particularly complex. For example, bpf_sys_bpf has deep call depths, a large number of loops and indirect calls, and accesses many kernel panic interfaces. This complexity can lead to overestimation of the calculated stack depth. However, we hypothesize that there is a subset of helper functions for which the generated call graph is straightforward and accurately represented, which may be sufficient for some popular classes of BPF extensions. For this subset, our proposed call graph solution will be effective.

3 RELATED WORKS

Providing a safe way to extend Operating Systems has been a pivotal area of research for over two decades. A number of approaches have been proposed to provide safety by using type-safe languages [1], software fault isolation [10, 12], interpretive languages [7], proof-carrying code [5, 8], and hardware-based protection mechanisms [3] but they do not address how multiple extensions interact and co-exist safely with other extensions.

Similar to stack management in kernel extensions, embedded systems must work within tight resource constraints. In work proposed by Biswas et al. [2], they studied how to protect embedded system stacks from overflow. To address the problem, they proposed reusing dead space in the heap and compressing live data if additional stack memory is required. This kind of approach for providing runtime safety will not work if there is a possibility of infinite nesting. A similar approach has been recently discussed on the Linux mailing list [11] to solve nesting issues to facilitate a scheduling usecase. The proposed design is to allocate stack variables in the heap or per-cpu pre-allocated memory so that multiple-level nested BPF programs cannot easily overflow the stack. Even this proposed design has the same problem of potential runtime overflow with deeper nested call chains.

We have suggested that with a call graph, the verifier will have the ability to prevent unsafe nesting. Some related work has been done in this area to evaluate the code coverage of system calls to gain insight into the effectiveness of fuzzers with Function Call Graphs [4]. However fuzzers will not cover all execution paths of the program and are therefore ineffective for estimating stack sizes statically.

4 SUMMARY

In this extended abstract, we presented the fundamental issue of the verifier's lack of a global view on BPF programs' interactions leading to problems like kernel stack overflows. We then provided a helper-rooted callgraph-based approach that can fix this problem at verification time. Finally, we provided thoughts on why the generated callgraph solution has to be limited to a subset of helper functions.

ACKNOWLEDGEMENT

This work was funded in part by NSF grant CNS-2236966 and by a grant from 4-VA, a collaborative partnership for advancing the Commonwealth of Virginia.

ETHICAL CONCERNS

This work raises no ethical concerns.

REFERENCES

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. 1995. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 267–283. https://doi.org/10.1145/224056.224077

[2] Surupa Biswas, Thomas Carley, Matthew Simpson, Bhuvan Middha, and Rajeev Barua. 2006. Memory overflow protection for embedded systems using run-time checks, reuse, and compression. *ACM Transactions on Embedded Computing Systems (TECS)* 5, 4 (2006), 719–752.

[3] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. 1999. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 140–153. https://doi.org/10.1145/319151.319161

[4] Mingi Cho, Hoyong Jin, Dohyeon An, and Taekyoung Kwon. 2021. Evaluating Code Coverage for Kernel Fuzzers via Function Call Graph. *IEEE Access* 9 (2021), 157267–157277. https://api.semanticscholar.org/CorpusID:244372100

[5] Daniel Kästner and Christian Ferdinand. 2014. Proving the Absence of Stack Overflows. In *Computer Safety, Reliability, and Security*, Andrea Bondavalli and Felicita Di Giandomenico (Eds.). Springer International Publishing, Cham, 202–213.

[6] Kangjie Lu. 2023. Practical Program Modularization with Type-Based Dependence Analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1256–1270. https://doi.org/10.1109/SP46215.2023.10179412

[7] Steven McCanne and Van Jacobson. 1993. The BSD packet filter: a new architecture for user-level packet capture *(USENIX'93)*. USENIX Association, USA, 2.

[8] George C. Necula and Peter Lee. 1996. Safe kernel extensions without run-time checking. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*. Association for Computing Machinery, New York, NY, USA, 229–243. https://doi.org/10.1145/238721.238781

[9] Siddharth Chintamaneni Sai Roop Somaraju and Dan Williams. 2023. Overflowing the Kernel Stack with BPF. In *Linux Plumbers Conference 2023*. https://lpc.events/event/17/contributions/1595/

[10] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. 1996. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *USENIX 2nd Symposium on OS Design and Implementation (OSDI 96)*. USENIX Association, Seattle, WA. https://www.usenix.org/conference/osdi-96/dealing-disaster-surviving-misbehaved-kernel-extensions

[11] Yonghong Song. 2024. Segmented Stacks for BPF Programs. https://lore.kernel.org/bpf/g2eynf5qrku2y5g433syeftgp3l2yg2sqawmvcee37ygezkslx@tklh2vnevwhx/T/. (2024). Accessed: 2024-02-14.

[12] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635