Research Report: Enhanced eBPF Verification and eBPF-based Runtime Safety Protection

Guang Jin
Trusted Science & Technology, Inc.
Rockville, Maryland, USA
guang@trustedst.com

Jason Li
Trusted Science & Technology, Inc.
Rockville, Maryland, USA
jason@trustedst.com

Greg Briskin
Trusted Science & Technology, Inc.
Rockville, Maryland, USA
greg@trustedst.com

***Abstract*—The *extended Berkeley Packet Filter* (eBPF) technology has been extending the capabilities of current *Operating Systems* (OSs) rapidly in recent years. The eBPF community is well-aware of using formal verification methods to ensure the security of eBPF programs. However, each of the two primary kinds of formal methods, namely abstract interpretation and symbolic execution, comes with their own set of pros and cons. This research report presents our formal eBPF verification approach, which combines the merits of both types of formal methods to ensure soundness, completeness, precision and recall for our solution. This solid security foundation makes eBPF-based applications particularly appealing in the field of cybersecurity. In addition, this research report describes our eBPF-based solution to enhance the runtime security for prebuilt user-space programs. Grounded in a formally provable security foundation, our eBPF-based runtime safety monitoring solution avoids introducing new errors, offers customization to counter various vulnerabilities, and eliminates the need for offline instrumentation.**

Keywords*—*cybersecurity, formal verification, eBPF, runtime verification

I. Introduction

The *extended Berkeley Packet Filter* (eBPF) is a subsystem that allows modern *Operating Systems* (OSs) (including both Linux [1] and Windows [2]) to safely execute untrusted user-defined extensions inside the OS kernel. Figure 1 illustrates the eBPF architecture used in a modern OS, such as Linux. The eBPF technology defines a register-based *Virtual Machine* (VM) using a 64-bit reduced instruction set. Using a *Just-In-Time* (JIT) compiler and the *Compile Once, Run Everywhere* (CORE) features, eBPF bytecodes can be translated into host native machine code thus extending program capabilities without modifying kernel source codes or writing kernel modules. While the original *Berkeley Packet Filter* (BPF) programs were mainly used in network-related applications (via processing network packets), the eBPF technology has been substantially expanded to support various system introspection capabilities and now become suitable for performing various cyber-security related functions, such as observability, monitoring, tracing, and profiling.

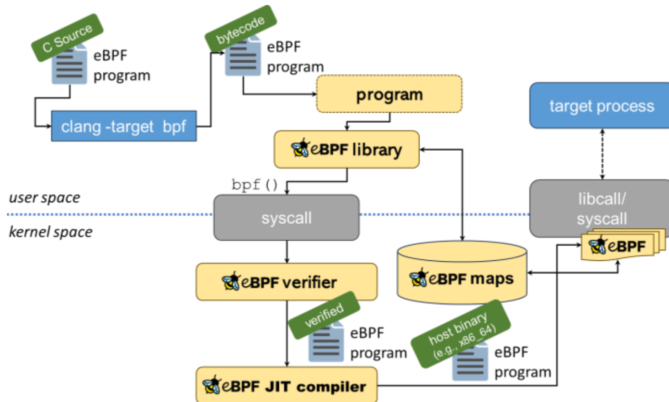


Figure 1: eBPF architecture

To protect the kernel against potentially buggy and malicious extensions, eBPF relies on static analysis performed by the in-kernel verifier. However, the current verifier became a major barrier for developers due to its manifested limitations, such as high rate of false positives, lack of both soundness and completeness, as well as poor scalability. In our opinion, formal verification of the eBPF program code is essential to ensure the safety of eBPF execution inside the kernel.

This research report presents an overview of our ongoing efforts at *Trusted Science & Technology, Inc.* (TrustedST) to apply formal methods on eBPF verification, and to build an eBPF-based runtime safety protection framework.

II. Formal eBPF Verification

A. Background and Problem Statement

The eBPF technology has several features that are appealing to the formal verification community. First, the eBPF VM, as well as the eBPF assembly and C languages are extremely restricted. For example, the current eBPF VM does not support floating-point number operations, and each eBPF program can only use up-to 4K instructions. Second, the verification concept does exist in the eBPF ecosystem in the form of in-kernel verifier which is supposed to check the safety of eBPF programs before translating/executing them in the kernel [1]. The state-of-the-art automatic formal verification methods can be divided into two categories, shown in Table 1.

Table 1: Abstract Interpretation v.s. Symbolic Execution

	Abstract Interpretation	Symbolic execution
Pros	Fast analysis with guaranteed termination	No false positives or negatives
Cons	Unavoidable false positives	Slow analysis and may not terminate

- *The abstract interpretation-based approaches* use abstract functions mapping concrete domains to abstract domains [3].

Applicable funding agency: DARPA.

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

The elements of concrete domain represent the concrete states of input programs and have the corresponding abstract elements within abstract domains. Analyses based on the abstract interpretation are conducted over abstract elements that over-approximate the concrete elements. PREVAIL, for example, applies abstract interpretation to statically analyze and verify eBPF programs over the abstract zone domain [4]. Another notable example is the Linux kernel's eBPF verifier, which is implicitly based on the abstract interpretation. However, the Linux eBPF verifier's heuristics-analysis is inherently unsound and may also introduce false negatives [5]. In general, the abstract interpretation-based approaches are fast. However, due to the over-approximation, false positives are unavoidable [3].

- For the *symbolic execution-based approaches,* symbolic values are fed to program inputs to explore all possible execution paths. Assertions are often instrumented to mark possible unsafe states of input programs. The path conditions are collected during a symbolic execution. The reachability of unsafe states is then verified by an automatic *Satisfiability Modulo Theories* (SMT) solver to solve the path constraints leading to the assertions. For example, Radius2 is a fast symbolic execution and taint analysis framework covering different architectures and executable formats (including eBPF bytecodes) [6]. Another well-known binary analysis and symbolic execution framework, angr, also supports eBPF [7]. Exoverifier is a proof-carrying code framework for eBPF programs and uses the SMT-based symbolic execution for proof generation [8]. The SMT-based symbolic execution and model checking is usually accurate and precise. However, the SMT solver often becomes a bottleneck and may greatly impact the overall analysis speed. Since the Linux kernel v5.3 started supporting loops in eBPF programs [9], simple symbolic execution-based eBPF verification may become trapped in loops, leading to non-termination.

III. TRUSTEDST'S FORMAL EBPF VERIFICATION

A. *Our Approach*

As indicated in Table 1, both abstract interpretation and symbolic execution come with their distinct advantages and limitations. Verification based on the abstract interpretation theory usually runs fast at the cost of unavoidable false positives. Symbolic execution-based approaches can thoroughly explore all possible program states, but the associated computational cost renders them impractical in certain scenarios. In our solution, we aim at blending the merits of both approaches while circumventing their drawbacks.

Figure 2 illustrates the eBPF verification workflow of our solution. Like all eBPF verification approaches, we take eBPF bytecodes as input. The verification starts with *sound analysis* based on abstract interpretation to avoid false negatives. Symbolic execution is used here only to encode the path conditions leading to the detected safety violations. As sound analysis can guarantee that safety violations are not present in certain paths [3], these paths can be safely ignored for further symbolic execution-based analysis. An automatic SMT solver (e.g., Z3 [10]) is used to prove or disprove the violation conditions. In this way, we can minimize false positives while still maintaining the efficiency of verification.

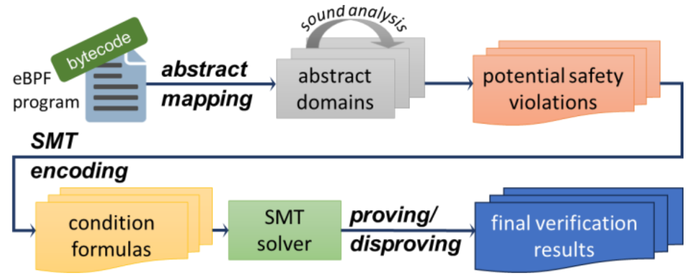


Figure 2: Our eBPF verification workflow

Figure 3 lists an eBPF code snippet that was incorrectly rejected by a Linux kernel eBPF verifier [8]. The current eBPF VM supports only integer registers and requires all registers to be properly initialized at function entries. Moreover, Linux kernel's eBPF verifiers only conducted analysis on non-relational abstract domains such as the interval domain. Assume, at $\mathcal{L}_1$, the intervals are found that `r1`$\in [0, U_{64}MAX]$ and `r2`$\in [0, U_{64}MAX]$. At $\mathcal{L}_3$, the interval of `r1` can be updated to $[0, 7]$, since $\mathcal{L}_3$ satisfies the `else` path condition of $\mathcal{L}_2$. Since the interval domain is non-relational, the abstract interpretation-based analysis does not trace the relationship between the registers. At $\mathcal{L}_3$, the interval of `r2` is still $[0, U_{64}MAX]$. Assume here `r0` points to an array or a network packet, at $\mathcal{L}_4$ then an out-of-bound error may be falsely detected. Analysis over relational abstract domains (such as the zone domain used in [4]) can reduce the false positives (e.g., the out-of-bound error shown in Figure 3). However, due to the over-approximation of abstract interpretation, even sound analysis may still produce false positives [3].

Location	Statement
	⋮
$\mathcal{L}_1$:	`r1 = r2`
$\mathcal{L}_2$:	`if r1 >= 8 goto out`
$\mathcal{L}_3$:	`r0 += r2`
$\mathcal{L}_4$:	`*(u8 *)(r0) = 0`
	⋮
	`out: exit`

Figure 3: A snippet of an eBPF program

When (potentially false or true) violations are inferred by the abstract interpretation analysis, our solution applies symbolic execution only over the paths leading to these inferred safety violations. We use the symbolic execution in a top-down fashion to start the analysis at the eBPF program entry points. In the example of Figure 3, our symbolic execution is only interested in the path conditions leading to the out-of-bound error found at $\mathcal{L}_4$. The path conditions can be found as r1=r2$\in [0, 7]$ by an automatic SMT solver. Using the described technique, we, therefore, can disprove this out-of-bound error and accept the given eBPF program code.

B. *Our Implementation*

We are actively developing our formal eBPF verification framework. Figure 4 illustrates our current prototype architecture. Our prototype is based on both PREVAIL [11] and Radius2 [6]. PREVAIL performs static analysis by converting

eBPF bytecodes into PREVAIL instructions, instrumenting assertion checks into the corresponding eBPF *Call Flow Graphs* (CFGs) and conducting the zone-domain analysis within the Crab framework [4]. Radius2 conducts the symbolic execution by proving the reachability and tracking the *Program Counter* (PC) states. Radius2 originally interfaces with Radare2 [12] and uses Boolector [13] as the automatic SMT solver. Radare2 acts as a disassembler converting binary codes into *Evaluable Strings Intermediate Language* (ESIL) [14]. Radius2 then interprets ESIL into PC states and uses the automatic SMT solver to prove the reachability. Using the original Radius2 and Radare2 can perform symbolic execution over the original non-instrumented eBPF programs. To formally verify the security of eBPF (or any other types of programs), however, instrumentation is needed to add assertions to check potential security violations.

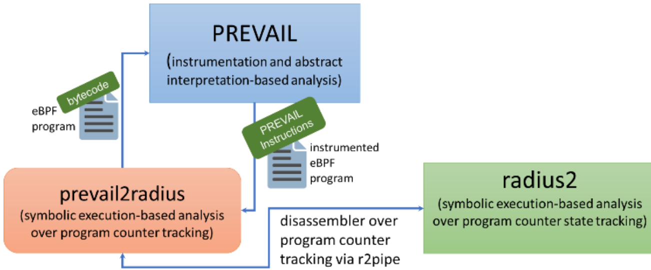


Figure 4: Our eBPF Verification Prototype Architecture

To unify the formal verification of both PREVAIL and Radius2, our verification prototype reuses PREVAIL's assertion instrumentation which is implemented as the Crab *Intermedia Representation* (IR). As shown in Figure 4, a major software tool in development for our prototype is named as `prevail2radius` which interfaces both symbolic execution and abstract interpretation. Our `prevail2radius` operates as a mediator between Radius2 and PREVAIL. Via the r2Pipe interfacing Radius2, `prevail2radius` works as a disassembler and provides ESIL outputs. Our `prevail2radius` does not directly disassemble eBPF bytecodes like Radare2 does. Instead, `prevail2radius` interacts with PREVAIL to convert and instrument eBPF bytecodes into the Crab IR. In PREVAIL, instrumented assertions are one type of instruction. When disassembling assertions, `prevail2radius` adds special PC states (i.e., assertion failures) via conditional branches (on checking security properties) to the original eBPF CFGs. Radius2 then uses the symbolic execution and Boolector to prove the reachability. As `prevail2radius` outputs modified CFGs via the added conditional branches, the reachability proofs are equivalent to the formal security proofs. In addition, `prevail2radius` can trim out the safe branches as identified by PREVAIL, and only conducts the additional symbolic execution over potentially unsafe branches via Radius2. Since PREVAIL conducts sound analysis and never produces false negatives, our eBPF verification prototype performs sound analysis as well and uses Radius2 to further fine-tune the formal verification. With the help of Boolector to automatically solve branch conditions, our eBPF verification prototype can find *concrete input values* to prove the reachability of security violations. These concrete inputs are essential to create effective exploits or to generate efficient security patches for vulnerable eBPF programs.

Figure 5: Memory Layout of eBPF Programs under Verification

Figure 5 shows the memory layout of eBPF programs under the verification in our prototype. As discussed above, eBPF programs are instrumented with PREVAIL assertions for security checks. Different from the original fixed 64-bit instruction length, after the `prevail2radius` translation, the original eBPF instructions and PREVAIL assertions are represented as variable-length instructions to Radius2 in our prototype. A single PREVAIL assertion is often translated by `prevail2radius` into multiple variable-length instructions to Radius2. For example, to verify the stack access assertion, four instructions are used (two instructions to `mov` the access addresses to temporary registers and two conditional branches to check the register contents against the lower and upper stack bounds). Our current prototype designates the initial 1K address space specifically for assertion failures, as depicted in Figure 5. To put it differently, we encode security violations within this 1K address range. When the PC reaches this lower 1K address, our eBPF verification prototype can demonstrate the presence of specific security violations using concrete inputs. Except for the code space, upper address ranges are used for data. Via parsing the *BPF Type Format* (BTF) [15], our prototype constructs virtual memory content within the data address range and assigns symbolic values to memory addresses with primitive data types. If an input argument (i.e., `r1` to `r5`) does not use the BTF pointer, the corresponding register is assigned with the symbolic value directly.

C. Initial Results

We are actively developing our eBPF verification framework. Our current prototype can already be used to verify some eBPF programs. Figure 6 shows a screenshot of one test. In this example, we define x as a 64-bit integer and assign `r1` as the symbolic value. When interacting with Radius2, our prototype can reuse instrumented PREVAIL assertions and translate the assertions into conditional branches for the symbolic execution to prove the reachability.

```
ebpf@ebpfx86:~/radare/radius/target/debug$ ./radius2 -p prevail:/home/ebpf/prism
.tmp.bpf -s x 64n -S r1 x 64
0x61c6af338b50      r7 = r1                                    |  r1,r7,=
0x61c6af338b58      r1 = 0                                     |  0,r1,=
0x61c6af338b60      r11 = r10                                  |  r10,r11,=
0x61c6af338b61      r11 -= 1                                   |  1,r11,-=
0x61c6af338b62      assert((r10 + -1) >= 0)                    |  0,r11,>=,?{,0x61
c6af338b63,}{,0x0,},pc,=
0x61c6af338b63      r11 += 1                                   |  1,r11,+=
0x61c6af338b64      assert((r10 - 1) <= eBPF_stack_size)       |  512,r11,<=,?{,0x
61c6af338b68,}{,0x1,},pc,=
0x61c6af338b68      *(u8 *)(r10 + -1) = r1                     |  r1,-1,r10,+,=[8]
```

Figure 6: A screenshot of using current prototype to verify an eBPF program

Figure 7 lists the assembly code of `ctxoffset.o`, one eBPF example included in PREVAIL [11]. At $\mathcal{L}_2$, an integer overflow may occur when `r1`$> U_{64}MAX - 8$. PREVAIL can detect this security issue via the abstract interpretation-based analysis. Our current prototype can also detect this issue by combining the abstract interpretation-based and symbolic

execution-based analyses. It is worth noting that symbolic execution can help us to find concrete input values to trigger specific security issues. These concrete inputs are particularly useful for end users to understand the detected security issues or to find remedies for their eBPF programs.

Location	Statement
$\mathcal{L}_0$	`r2 = 0x1`
$\mathcal{L}_1$:	`*(u32 *)(r10 - 0x4) = r2`
$\mathcal{L}_2$:	`r1 += 0x8`
$\mathcal{L}_3$:	`r3 = r10`
$\mathcal{L}_4$:	`r3 += -0x4`
$\mathcal{L}_5$	`r2 = 0x0 ll`
$\mathcal{L}_7$	`r4 = 0x0`
$\mathcal{L}_8$	`call 0x35`
$\mathcal{L}_9$	`exit`

Figure 7: Assembly code of `ctxoffset.o`

Figure 8 illustrates the screenshot of using our current prototype to verify `ctxoffset.o`. Here, `r1` is assigned as a symbolic 64-bit integer a .When iterating with Radius2, `prevail2radius` outputs modified CFGs with the added conditional branches for PREVAIL assertions. Radius2 uses the automatic SMT solver to find solutions to reach the assertion failure PC states. If PC goes to the lower 1K address range, Radius2 detects a crash and outputs the corresponding concrete input values as the proof to trigger the crash. As shown in Figure 8, $a = U_{64}MAX - 7$ is found to trigger the integer overflow.

```
ebpf@ebpfx86:~/radare/radius/target/debug$ ./radius2 -p prevail:/home/ebpf/ebpf-
verifier/ebpf-samples/build/ctxoffset.o --crash -s a 64n -S r1 a 64
0x5c5441ee00b0      r2 = 1                                  |  1,r2,=
0x5c5441ee00b8      r11 = r10                               |  r10,r11,=
0x5c5441ee00b9      r11 -= 4                                |  4,r11,-=
0x5c5441ee00ba      assert((r10 + -4) >= 0)                 |  0,r11,>=,?{,0x5c
5441ee00bb,}{,0x0,},pc,=
0x5c5441ee00bb      r11 += 4                                |  4,r11,+=
0x5c5441ee00bc      assert((r10 - 4) <= eBPF_stack_size)    |  512,r11,<=,?{,0x
5c5441ee00c0,}{,0x1,},pc,=
0x5c5441ee00c0      *(u32 *)(r10 + -4) = r2                 |  r2,-4,r10,+,=[32
]
0x5c5441ee00c8      ignore_assert(r1.type in {number, ctx, stack, packet, shared
}) |  r1,r1,=
0x5c5441ee00d0      r1 += 8                                 |  8,r1,+=
0x5c5441ee00d8      r3 = r10                                |  r10,r3,=
0x5c5441ee00e0      ignore_assert(r3.type in {number, ctx, stack, packet, shared
}) |  r3,r3,=
0x5c5441ee00e8      r3 += -4                                |  -4,r3,+=
0x5c5441ee00f0      load_map_fd(r2 = 1)                     |  1,r2,=
0x5c5441ee00f8      r4 = 0                                  |  0,r4,=
0x5c5441ee0100      ignore_assert(r1.type == ctx)           |  r1,r1,=
0x5c5441ee0108      assert(r1 != 0)                         |  0,r1,-,?{,0x5c54
41ee0110,}{,0x7,},pc,=

symbolic PC : (ite (= a #xfffffffffffffff8) #x0000000000000007 #x00005c5441ee011
0)

0x5c5441ee0110      ignore_assert(r2.type == map_fd)        |  r2,r2,=
0x00000007      context offset zero failure             |  7,TRAP
crash found
0x5c5441ee0118      ignore_assert(r3.type in {stack, packet}) |  r3,r3,=
0x5c5441ee0120      ignore_assert(within stack(r3:key_size(r2))) |  r11,r11,=
0x5c5441ee0128      ignore_assert(r4.type == number)        |  r4,r4,=
0x5c5441ee0130      call 53                                 |  pc,sp,=[8],8,sp,
-=,0x35,$
0x5c5441ee0138      ignore_assert(r0.type == number)        |  r0,r0,=
0x5c5441ee0140      exit                                    |  8,sp,+=,sp,[8],p
c,=
state.backtrace.is_empty():true new_flags.is_empty():true

  a : 0xfffffffffffffff8
```

Figure 8: A screenshot of using our current prototype to verify `ctxoffset.o`

It can also be found in Figure 8 that our current eBPF verification prototype ignores some PREVAIL assertions. Most of the ignored assertions pertain to type checking, which should not cause false positives. At TRUSTEDST, we are actively developing and improving our formal eBPF verification framework. The future version of our eBPF verification solution will use the symbolic execution-based analysis to find proofs for all assertions instrumented by the abstract interpretation-based analysis.

IV. Conclusion and Future Plan

This research report summarizes our on-going eBPF related efforts. We are using formal methods to ensure the security of eBPF programs. This paper details how our verification solution blends the benefits of both symbolic execution and abstract interpretation to achieve both effectiveness and efficiency. Using this solution allows us to formally verify eBPF programs and to lay down a solid security foundation for rich eBPF based applications. In addition, the appendix lists our efforts to develop a dynamic, extensible, and flexible eBPF-based solution to enhance the runtime security of prebuilt binary programs without offline instrumentation. We plan to mature and release our final solutions in the near future.

Acknowledgment

This work was partially supported by the DARPA *Verified Security and Performance Enhancement of Large Legacy Software* (V-SPELLS) program. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors or collaborators.

References

[1] "eBPF verifier," The kernel development community, [Online]. Available: https://docs.kernel.org/bpf/verifier.html. [Accessed January 2023].

[2] "GitHub Repository of ebpf-for-windows," Microsoft Corporation, [Online]. Available: https://github.com/microsoft/ebpf-for-windows. [Accessed January 2024].

[3] A. Miné, Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation, Now Foundations and Trends, 2017.

[4] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk and 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, pp. 1069-1084, June 2019.

[5] H. Vishwanathan, M. Shachnai, S. Narayana and S. Nagarakatte, "Verifying the Verifier: eBPF Range Analysis Verification," In Proceedings of 35th International Conference on Computer Aided Verification, pp. 226-251, July 2023.

[6] "Github Repository of radius2," The radius2 Workgroup, [Online]. Available: https://github.com/aemmitt-ns/radius2. [Accessed January 2024].

[7] "Github repository of angr's ebpf plugin," The angr workgroup, [Online]. Available: https://github.com/angr/angr-platforms/tree/master/angr_platforms/bpf. [Accessed January 2023].

[8] L. Nelson, X. Wang and E. Torlak, "A proof-carrying approach to building correct and flexible in-kernel verifiers," September 2021. [Online]. Available: https://homes.cs.washington.edu/~lukenels/slides/2021-09-23-lpc21.pdf.

[9] J. Corbet, "Bounded loops in BPF programs," 3 December 2018. [Online]. Available: https://lwn.net/Articles/773605/.

[10] "Z3: An efficient SMT solver," Microsoft, [Online]. Available: https://www.microsoft.com/en-us/research/project/z3-3/. [Accessed September 2023].

[11] "Github Repository of PREVAIL," The PREVAIL Workgroup, [Online]. Available: https://github.com/vbpf/ebpf-verifier. [Accessed January 2024].

[12] "Github Repository of Radare2," Radare.org, [Online]. Available: https://github.com/radareorg/radare2. [Accessed Feburary 2024].

[13] A. Niemetz, M. Preiner and A. Biere, "Boolector 2.0 system description," Journal on Satisfiability, Boolean Modeling and Computation, vol. 9, pp. 53 - 58, 2014 (published 2015).

[14] "ESIL," Radare.org, [Online]. Available: https://book.rada.re/disassembling/esil.html. [Accessed Feburary 2023].

[15] "BPF Type Format (BTF)," The kernel development community. [Online]. [Accessed Feburary 2024].

[16] J. Keniston, P. S. Panchamukhi and M. Hiramatsu, "Kernel Probes (Kprobes)," [Online]. Available: https://docs.kernel.org/trace/kprobes.html. [Accessed January 2024].

[17] S. Dronamraju, "Uprobe-tracer: Uprobe-based Event Tracing," [Online]. Available: https://docs.kernel.org/trace/uprobetracer.html. [Accessed January 2024].

[18] "Official Website of Cilium," Isovalent, Inc., [Online]. Available: https://cilium.io/. [Accessed January 2024].

[19] D. Thompson and L. Yan, "Kernel analysis using eBPF," [Online]. Available: https://elinux.org/images/d/dc/Kernel-Analysis-Using-eBPF-Daniel-Thompson-Linaro.pdf. [Accessed January 2024].

[20] Q. Monnet, "Github Repository of rbpf," [Online]. Available: https://github.com/qmonnet/rbpf. [Accessed January 2024].

[21] "Github Repository of ubpf," The IO Visor Project, [Online]. Available: https://github.com/iovisor/ubpf. [Accessed January 2024].

[22] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai and A. Quinn, "bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions," 14 November 2023. [Online]. Available: https://arxiv.org/abs/2311.07923.

[23] "Github Repository of bpftime," The eunomia-bpf Project, [Online]. Available: https://github.com/eunomia-bpf/bpftime. [Accessed January 2024].

[24] T. Liu, C. Curtsinger and E. D. Berger, "DoubleTake: fast and precise error detection via evidence-based dynamic analysis," In Proceedings of the 38th International Conference on Software Engineering, pp. 911 - 922, May 2016.

[25] A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts, 10th ed., Wiley, 2018.

[26] M. Kerrisk, "bpf-helpers(7) — Linux manual page," 22 December 2023. [Online]. Available: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[27] "Dynamic, context-aware access control for G Suite now generally available," Google, 29 October 2019. [Online]. Available: https://workspaceupdates.googleblog.com/2019/10/context-aware-access-available.html.

[28] "Dynamic Access Control: Scenario Overview," Microsoft, 27 September 2022. [Online]. Available: https://learn.microsoft.com/en-us/windows-server/identity/solution-guides/dynamic-access-control-overview.

APPENDIX: EBPF BASED RUNTIME SAFETY PROTECTION

A. *In-Kernel and User-space eBPF*

Current eBPF frameworks enable a flexible hooking mechanism through which customized eBPF programs can be dynamically attached at the entry and exit points of target routines. For example, `kprobe` and `kretprobe` are used for hooking the OS kernel [16], while `uprobe` and `uretprobe` are for user space *library calls* (libcalls) [17]. The cilium project is a well-known framework which uses eBPF hooks to provide networking, security, and management capabilities for container-based clouds [18].

As illustrated by Figure 1, eBPF programs are typically inserted into and run within the kernel-space. After intercepting the runtime activities of a target process (or a group of processes within a container), in-kernel eBPF programs store these activities into eBPF maps of different data types including hash, array, and bloom filter. In a typical eBPF application scenario, a user-space program is required to perform additional analysis over the information stored in eBPF maps. The frequent context-switching between kernel- and user- spaces dominates the runtime overhead of eBPF applications [19].

Several recent projects (e.g., *rbpf* [20], *ubpf* [21] and *bpftime* [22, 23]) move the in-kernel eBPF execution into the user-space. As illustrated by Figure 9, the emerging user-space eBPF architecture injects natively compiled eBPF programs into a target user-space program/process dynamically at the runtime. Compared to its in-kernel counterpart, the user-space eBPF architecture does not require the costly context-switching. Via its verifier capabilities (e.g., PREVAIL is used by bpftime [23]), the user-space eBPF architecture provides the same security guarantees as the in-kernel eBPF does. By supporting eBPF maps (e.g., reimplementing eBPF maps using shared memory [22]), user-space eBPF implementations support existing eBPF toolchains and even pre-built eBPF bytecodes. Although the user-space eBPF architecture only allows hooking over libcalls via `uprobe` and `uretprobe`, the user-space eBPF can still provide a foundation for secure, effective, and efficient runtime safety protection for our solution.

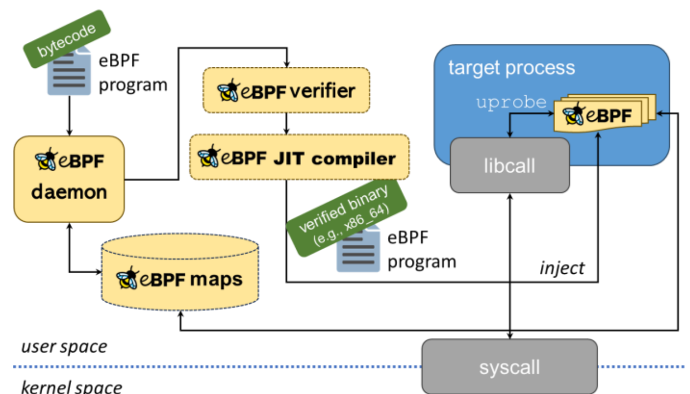


Figure 9: User-space eBPF architecture

B. *TRUSTEDST'S Runtime Safety Protection*

We are currently designing and implementing eBPF based runtime safety protection capabilities to detect security

violations, errors, or failures and to provide extra security enhancement for prebuilt programs without offline instrumentation. As shown in Figure 10, we are designing and implementing eBPF programs for specific safety needs. These eBPF programs/hooks are dynamically injected into the target processes to store their runtime activities into eBPF maps. The security guarantee of eBPF ensures that these eBPF hooks do not decrease the security of target programs. We are also creating an extensible verification server that runs as a daemon process and supports customizable plugins to analyze real-time data from eBPF maps for various safety concerns. In addition, we plan to support a web-based front-end to visualize the safety analysis results and manage the runtime safety protection for non-expert users.

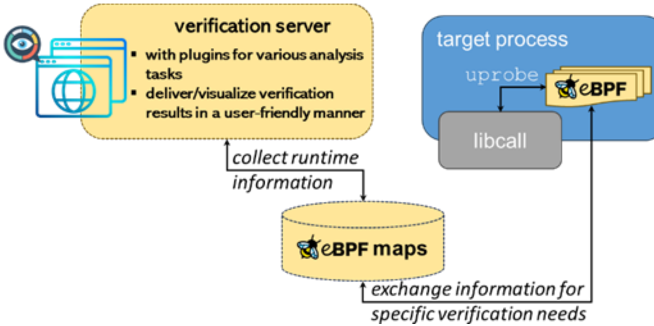


Figure 10: Our eBPF based runtime verification architecture

Our solution is mainly based on the user-space eBPF technology since it provides better extensibility, flexibility, efficiency, and security. The following examples demonstrate the versatility of our eBPF-based approach.

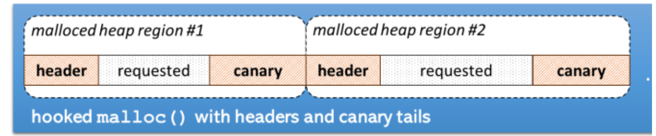
1) Example 1: Heap Profiler/Analyzer

Table 2: eBPF hooks for heap profiler/analyzer

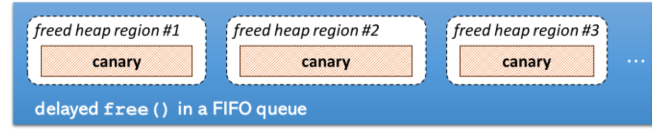
uprobe/uretprobe points	runtime information to collect
`malloc()` and `free()` of `libc.so`	▪ `pid` and `tid` of callers ▪ size and virtual address of allocated heap space

As summarized by Table 2, in order to profile and analyze heaps, eBPF hooks are attached to the entry and exit points of both `malloc()` and `free()` of `libc`. During the runtime of target programs/processes, the eBPF hooks collect the caller's `pid` and `tid` of heap-related functions, and store heap-related information into eBPF maps. Our heap analyzer runs as a plugin within the verification server and monitors the heap usage of the targets to detect memory-corruption errors (e.g., double-frees). More eBPF code can be developed to hook file-related functions (e.g., `open()` and `close()`), while similar analysis methods can be used to detect double-close errors.

Like the native library interposition approach [24], the eBPF hooks can add extra data fields to heap regions to detect more types of memory-corruption errors. As shown by Figure 11.a, the eBPF hooks can, for example, add extra headers and canary tails over allocated heap regions to help detecting heap-leak errors, while the corrupted canaries can help to detect the heap overflows [24]. The eBPF hooks can delay the `free()` operations, and put the heap regions into a *First-In-First-Out* queue, as illustrated by Figure 11.b. By filling the queued heap



(a) eBPF hooks to add protection data against heap-leaks and heap-overflows



(b) eBPF hooks to delay `free()` operations and detect use-after-frees

Figure 11: eBPF hooks to detect extra heap related errors

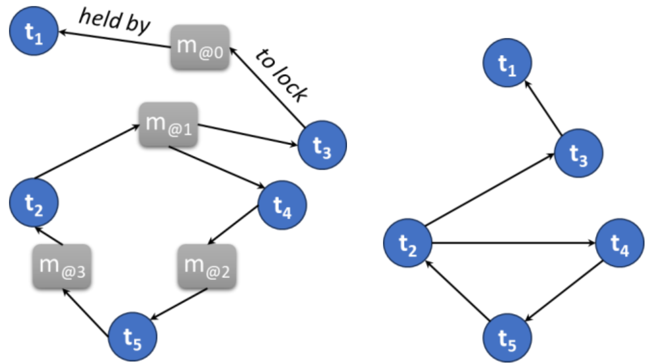
regions with canaries, use-after-free errors can be discovered when corruption of canaries is detected. As inspired by [24], another eBPF program can be attached to a target process to periodically analyze the trapped heap operations and report detected memory-corruption errors to the verification server.

2) Example 2: Mutual Exclusion Lock Analyzer

Table 3: eBPF hooks for mutual exclusion locks

uprobe/uretprobe points	runtime information to collect
`mtx_init()`, `mtx_destroy()`, `mtx_lock()`, `mtx_unlock()`, and other `mtx_`-related functios in `libc.so`	▪ `pid` and `tid` of callers ▪ invocation time of related functions ▪ addresses of in-use locks

Table 3 lists the eBPF hooks for analyzing *mutual exclusion* (mutex) locks. By monitoring the target program's runtime invocations of mutex related functions, these eBPF hooks can be used to detect race conditions.



(a) lock-allocation graph (b) wait-for graph

Figure 12: Dead-lock detection based on wait-by graph

By analyzing the runtime mutex operations of a target program, a lock-allocation graph can be constructed. Figure 12.a illustrates an example, where $t \rightarrow m$ indicates a thread t trying to lock on a mutex m, while $m \rightarrow t$ means a mutex m is being held by a thread t. By removing the lock nodes and collapsing adjacent edges from the local-allocation graph, the corresponding wait-for graph can be obtained. Dead-lock incidents can be detected by analyzing a wait-for graph and finding loops within it [25]. For example, Figure 12.b shows that the threads t_2, t_4 and t_5 form a dead-lock error. This mutex

analysis can be easily extended to detect other lock starvation issues. For example, live-locks can be detected if multiple threads are competing and trying to acquire the same mutex lock frequently.

These mutex-related eBPF hooks can also be applied to other resources (e.g., spinlocks, files, pipes etc.). Similar analysis algorithms can be used to detect general resource-related race conditions. In addition, the resource-allocation and wait-for graphs can be rendered to non-expert end-users to visualize the runtime safety results.

3) Example 3: Dynamic Access Control

Table 4: eBPF hooks for dynamic access control

uprobe/uretprobe points	**runtime information to collect**
`open()`, `close()`, `socket()`, `sendto()`, `sendmsg()`, `recv()`, `recvmsg()` and `shutdown()` of `libc.so`	▪ `pid` and `tid` of callers ▪ path names and security levels of opened files ▪ protocols and IP addresses of live sockets

Table 4 outlines more eBPF hooks used to enable and enforce a dynamic access control for prebuilt binary targets. The eBPF hooks monitor opened files and live network sockets of running targets to determine the runtime contextual information. Based on the runtime context of target programs/processes and predefined security policies, eBPF programs can grant or deny the access (of files or network operations) via the eBPF helper function, bpf_override_return() [26], as shown by Figure 13.

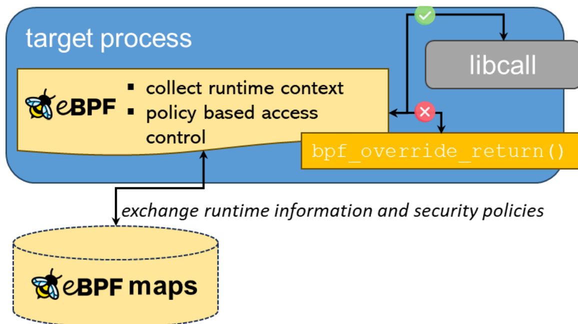


Figure 13: Context-aware dynamic access control via eBPF hooks

Like the existing dynamic access control solutions [27, 28], our eBPF based approach monitors and enforces the access control by capturing the dynamic runtime contextual information. For example, if a target process opens a secret file (with classification information tagged by additional tools), its on-going network operations must be dropped to prevent potential information leaks. While many existing solutions rely on program developers to follow certain *Application Programming Interfaces* (APIs) to enforce access control, our eBPF-based approach enables a runtime out-of-band control of any prebuilt binaries.

4) Example 4: Customized Signal Handler

Portable Operating System Interface (POSIX) signals are used to notify programs about certain runtime errors and other non-critical events. Table 5 lists eBPF based POSIX signal handlers used by our solution. The inserted eBPF hooks are invoked when POSIX signals are delivered to the target

Table 5: eBPF hooks for handling POSIX signals

uprobe/uretprobe points	**runtime information to collect**
`sigvec()` of `libc.so`	▪ `pid` and `tid` of callers ▪ `sig`: signal number ▪ `ctx`: context on which the target program/process runs ▪ stack traces

processes and before targets' own handlers are invoked. Additional runtime information including the signal numbers and stack traces (retrievable via `bpf_get_stack()` [26]) can be collected into eBPF maps to investigate the root causes.

Our runtime safety monitoring framework is currently under active development. As pointed out by the above examples, our highly extensible and flexible solution allows deployment of various eBPF hooks to monitor the runtime activities of prebuilt binary targets and enhance their security without the need for any offline rewriting or instrumentation.

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)