# VEP: A Two-stage Verification Toolchain for Full eBPF Programmability

Xiwei Wu[1], Yueyang Feng*[1], Tianyi Huang*[1], Xiaoyang Lu*[1], Shengkai Lin[1], Lihan Xie[1], Shizhen Zhao†[1], Qinxiang Cao†[1]

[1]Shanghai Jiao Tong University

## Abstract

Extended Berkely Package Filter (eBPF) is a revolutionary technology that can safely and efficiently extend kernel capabilities. It has been widely used in networking, tracing, security, and more. However, existing eBPF verifiers impose strict constraints, often requiring repeated modifications to eBPF programs to pass verification. To enhance programmability, we introduce VEP, an annotation-guided eBPF program verification toolchain. VEP consists of three components: VEP-C, a verifier for annotated eBPF-C programs; VEP-compiler, a compiler targeting annotated eBPF bytecode; and VEP-eBPF, a lightweight bytecode-level proof checker. VEP allows users to verify the correctness of their programs with appropriate annotations, thus enabling full programmability. Our experimental results demonstrate that VEP addresses the limitations of existing verifiers, i.e. the Linux verifier and PREVAIL, and provides a more flexible and automated approach to kernel security.

## 1 Introduction

Recently, eBPF [15] has gained significant popularity as a versatile technique. This innovative technology enables users to load programs into the Linux kernel dynamically, proving to be highly advantageous across various domains, including networking [2, 6, 29, 34, 46, 49],tracing [4, 43], security [3, 23, 47], storage [7, 51, 52], and more. In contrast to traditional Linux kernel modules, eBPF programs offer enhanced stability, as each program undergoes a rigorous verification process before attachment to the kernel. This verifier meticulously screens programs for unsafe behaviors, such as infinite loops and out-of-bound memory access, thus preventing potential crashes.

Existing eBPF verifiers (e.g., the Linux verifier [19] and PREVAIL [26]) often compromise programmability for security, sometimes rejecting safe eBPF programs. The Linux verifier [19] uses *register value tracking* to simulate the execution paths but limits programmability by imposing constraints such as maximum program size and loop complexity to prevent path explosion. PREVAIL [26] addresses this issue using *abstract interpretation* to merge paths, allowing variable-sized loops in eBPF programs. However, when loops involve complex, dynamic behaviors—such as termination conditions dependent on runtime data—PREVAIL's analysis could become imprecise, leading to safe programs being incorrectly rejected.

The eBPF maintainers have acknowledged this issue and have developed methods to mitigate these restrictions on program size and loop complexity. Since kernel version 3.18, the Linux verifier has employed *pruning* to eliminate redundant code paths [18, 19]. Kernel version 5.2 raised the instruction limit from 4096 to 1 million, and version 5.3 added support for bounded loops [16]. In version 5.13, map iterators were introduced to allow iteration through map elements [17], as loops are always unrolled by the verifier, which can lead to excessive code lengths when iterating over large maps. However, map iterators require a function pointer, which, while suited for functional languages, is cumbersome in C. Despite all these ad-hoc features, some safe programs remain impossible to pass the Linux verifier.

State-of-the-art verification tools have the potential to achieve full programmability. We broadly categorize modern verification tools into three types based on their design approaches when dealing with intricate algorithms, complex program properties, and the need for

*These authors contributed equally to this work.

†Corresponding authors.

high levels of automation: (1) Fully automatic verifiers. CBMC [12], a bounded model checker, verifies array bounds, pointer safety, and other properties in C programs by unrolling loops a fixed number of times. Infer [10], a static analysis tool developed by Meta, automatically checks memory safety in mobile and server-side code. These tools can automatically handle millions of lines of source codes. But their users have to tolerate false positives and negatives, especially when verifying complex programs with intricate algorithms and data structures. (2) Verifiers based on interactive theorem provers. This method can verify complex programs, but it requires significant manual effort in developing machine-checkable *proof code*. VST [24], as a representative of this type of tools, verifies properties of programs in the theorem prover Coq [48] by requiring users to describe the desired properties and manually write proof code to complete the proof. This approach lacks automation, and the time and memory consumption can be significant [1]. (3) Annotation verifiers based on SMT solver. This approach relies heavily on the SMT solver. VeriFast [31] allows users to provide assertions within the program that describe relevant properties, which are then evaluated by the SMT solver for correctness. Similarly, Dafny [33] leverages the Z3 [8] SMT solver to automate the verification of annotated programs. Unfortunately, even powerful and industry-standard SMT solvers such as Z3 and cvc5 [5] still encounter numerous unsolvable or incorrectly solved problems. Moreover, these SMT solvers are often of considerable scale, requiring substantial amounts of time and memory. In short, it is possible for verification tools to verify complex programs without posing any limitation (i.e. achieve full programmability) but with an additional cost such as requiring users to write annotations. Meanwhile, the tools must make trade-offs among efficiency, resource consumption, and potential false negatives and false positives (see Table 1).

For eBPF verification, the verifier must operate within the kernel, necessitating high efficiency, minimal resource usage, and the absence of false negatives. These requirements imply several constraints: First, despite the advanced capabilities of modern SMT solvers, their overhead and risk of false results make them unsuitable for kernel verification. Second, most eBPF programs are compiled with unverified compilers such as LLVM or GCC. We aim to minimize our Trusted Computing Base (TCB), avoiding including such monolithic components. Finally, verification tools must be highly automated and user-friendly to encourage widespread adoption by eBPF developers in practical settings.

Inspired by these considerations, we propose **v**erification toolchain for **e**BPF **p**rograms (VEP) based on an annotation-guided verification approach. The VEP toolchain comprises a C-level verifier, an annotation-aware compiler, and a bytecode-level proof checker. First, verification is conducted on the annotated C programs. To ensure they are free of undefined behavior(UB) per the C standard and only use limited resources. This phase also generates additional annotations and proof terms for later bytecode-level verification. Next, the verified C program is transformed into an annotated bytecode program by an annotation-aware compiler. In this step, the code, the annotations, and the proof terms are all converted to the bytecode level. Finally, a bytecode proof checker re-evaluates the proofs and the annotated bytecode to ensure compliance with eBPF standards, thereby completing the verification process. Our two-stage verification framework distributes substantial time and memory overhead to user space, while the kernel space contains only a proof checker with minimal time and space requirements.

In summary, we have developed a verification tool VEP for annotated eBPF programs, which offers the following advantages:

1. VEP achieves full programmability for eBPF programs, going beyond memory safety to verify the functional correctness of eBPF programs.
2. In user space, the verifier is highly automatic.
3. In kernel space, the proof checker is secure and efficient with lower time and memory consumption.
4. Our TCB is small, which includes only the proof checker in kernel space, excluding the C verifier and the compiler.

The structure of this paper is as follows: In Section 2, we describe the design of VEP, including the general framework and underlying design principles. In particular, we explain the design necessity of the two-stage verification scheme. In Section 3, 4 and 5, we introduce the detailed design of the three components of VEP respectively. In Section 6, we show our experiments and the performance of the proposed method. Section 7 and 8 introduce our future works and make a conclusion.

## 2 Overall Architecture of VEP

In this section, we will introduce the overall design of VEP and discuss the design choices of the two-stage

[1] The resource requirements of theorem-prover-based tools, such as VST, vary with implementation. Tools built entirely on theorem provers such as Coq and Isabelle [40] tend to have high time and memory demands, while implementations in languages such as C or Python such as VST-A [53] can greatly reduce these costs.

| Tool | Full Programability | Automation | Small TCB | Low Time/Memory Cost |
|---|---|---|---|---|
| CBMC/Infer | - | +++ | - - - | +++ |
| VST | + | - - | +++ | depend on implementation |
| VeriFast | + | + | - - - | + |

**Table 1:** Comparison of different verification tools.

verification framework.

## 2.1 Two-stage Verification

We have developed the verification tool VEP for annotated programs to achieve full programmability, enabling users to write any safe and valid program. Specifically, the VEP toolchain verifies whether an eBPF program (1) is free from aborting behaviors such as null-pointer dereferences and division by zero and (2) adheres to strict memory safety properties, including program-owned memory (e.g., stack, eBPF maps) and kernel-owned memory (e.g., pointers passed via helper functions or context structures).

VEP begins with a C-level verifier, VEP-C, which performs initial checks on user-annotated C programs. Developers can iteratively refine their code using VEP-C until it passes verification. In this process, VEP-C applies symbolic execution to compute the strongest postcondition for each program statement and leverages an SMT solver to derive assertions and generate corresponding proofs automatically. Here, postconditions define the conditions that must hold after executing a program segment, while preconditions specify the assumptions made before execution begins. Symbolic execution systematically computes the strongest postcondition based on the given preconditions, ensuring that the program adheres to its safety and resource constraints.

```
/*@ 0 ⩽ x < 100 */
x = x + 1;
/*@ 0 ⩽ x ⩽ 100 */
```

In this example, the precondition is $P \triangleq 0 \leqslant x < 100$, and the postcondition to be checked is $Q \triangleq 0 \leqslant x \leqslant 100$. We need to verify whether, after executing $c \triangleq$`x=x+1` under the condition $P$, the postcondition $Q$ holds. The first step is to compute the *strongest* postcondition; that is, the best we can tell about the program state after executing $c$ given the initial condition $P$. In this case, the strongest postcondition is $\exists x_0, x = x_0 + 1 \wedge 0 \leqslant x_0 < 100$, where intuitively $x_0$ represents the value of $x$ before the increment operation. During symbolic execution, such computation of strongest postconditions has its logic foundation in Hoare logic rules [28, 41]. The time complexity of the strongest postcondition computation is approximately linear with assertion length. So in general, the time complexity of symbolic execution is approximately linear with the product of the program length and the average assertion length.

Next, we use an SMT solver to determine whether $\exists x_0, x = x_0 + 1 \wedge 0 \leqslant x_0 < 100$ implies $0 \leqslant x \leqslant 100$. Although this example is simple, determining the validity of assertion derivations can be complex in practice. In fact, the decision problem is undecidable in general. Fortunately, existing research on SMT solvers has shown that modern solvers can produce correct results in many practical cases. VEP's built-in SMT solver is required not only to verify whether the derivations hold but also to provide (cvc5-style) proofs. Each proof consists of a list of proof steps. Every proof step derives a new conclusion from known propositions (including assumptions and conclusions proved earlier). Formally, a proof step contains one proof rule name, the associated parameters $t_i$, the premises $\varphi_i$, the resulting conclusion $\psi$, and its side condition $C$.

$$\text{RULENAME} : \frac{\varphi_1, \ldots, \varphi_n \mid t_1, \ldots, t_m}{\psi} \text{ if } C$$

In the example of $(\exists x_0, x = x_0 + 1 \wedge 0 \leqslant x_0 < 100) \Rightarrow 0 \leqslant x \leqslant 100$, the VEP's built-in SMT solver produces a 'YES' with a 138-line proof. It means that the code has been successfully verified by VEP-C, with both the corresponding proof and strongest postcondition generated.

Computed strongest postconditions and SMT-generated proofs are inserted back into the original annotated program. The annotation-aware compiler (VEP-compiler) then compiles the elaborated C program into annotated eBPF bytecode. Unlike traditional compilers, relevant assertions and proofs will also undergo corresponding compilation passes.

Finally, VEP uses a bytecode-level proof checker (VEP-eBPF) to perform a final verification. VEP-eBPF similarly uses symbolic execution to compute the strongest postcondition for each statement and then checks the corresponding proofs to derive the assertions, but without requiring SMT solvers: it only checks the proofs based on basic logic rules, using a minimal proof checker. If the verification passes, the user-written program has successfully undergone VEP's verification process. In the end, the eBPF program will be loaded in the kernel, and its execution is guaranteed to be safe. This security assurance is achieved solely by verifying that the programs and proofs checked by VEP-eBPF are consistent. Therefore, the entire TCB of VEP is effectively reduced to the lightweight proof checker, VEP-eBPF.

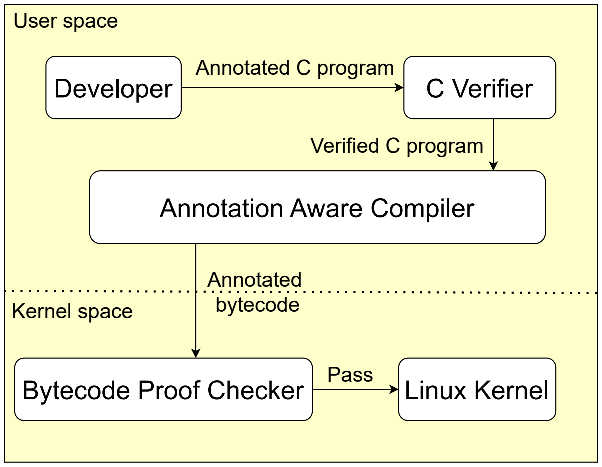

**Figure 1:** The framework of VEP.

All components of VEP were developed independently, without relying on existing tools. We aim for VEP to replace the current eBPF verification mechanism in Linux. Figure 1 shows the whole workflow of VEP. Our two-stage verification addressed the three requirements mentioned in Section 1.

- **Small and Efficient Kernel Integration.**
  Our two-stage verification process takes advantage of this by offloading complex SMT solver calls to VEP-C in user space while keeping a lightweight proof checker, VEP-eBPF, in kernel space. VEP-eBPF performs only straightforward logical deductions without complex SMT operations, ensuring the kernel component remains simple and efficient.

- **Minimal Final Trusted Computing Base (TCB).**
  The safety of loading eBPF programs depends solely on the correct implementation of VEP-eBPF. By isolating the critical verification logic in this small component, we minimize the TCB, thereby reducing risks associated with complex verification algorithms and enhancing overall reliability.

- **Highly Automated and User-Friendly.**
  While users must provide assertions to prove program safety, our approach leverages lessons from tools like VeriFast and VST by offering an annotation syntax that closely resembles C. This familiar syntax reduces the learning curve, making it easier for users to write and understand assertions.

## 2.2 Verification Capabilities

VEP is designed to verify critical properties of eBPF programs, ensuring they adhere to both safety and correctness guarantees. Here, safety refers to kernel safety, while correctness pertains to functional correctness verification—the assurance that program implementations precisely match their specified behaviors.

**Safety Guarantees:** Under the classification framework of KFlex [14], kernel safety consists of two core components: *kernel-interface compliance* and *extension correctness*. Since eBPF programs extend kernel functionality, *extension correctness* ensures safe memory accesses and guarantees program termination. Additionally, as eBPF programs often invoke kernel functions to perform specific tasks, *kernel-interface compliance* is equally critical. This requires that all kernel function calls are valid and that any resources acquired through these calls are properly released, preventing resource leaks.

VEP not only ensures *extension correctness* but also enforces *kernel-interface compliance*. VEP features an annotation language that enables users to formally specify program properties for verifying the following critical properties:

1. Enforcing memory access permissions for load/store operations and function calls.
2. Guaranteeing that all programs terminate within a finite number of steps.
3. Ensuring proper acquire-release ordering discipline for synchronization primitives (e.g., locks and semaphores).
4. Preventing resource leaks through comprehensive lifecycle tracking.

**Correctness Guarantees:** While the annotation requirements for functional correctness verification are indeed significantly more complex than those for safety verification, our annotation language has been specifically designed to facilitate the precise description of these complex properties. Since functional correctness verification fundamentally builds upon the foundation of safety verification, VEP enforces the requirement that all user-provided annotations must satisfy the four fundamental properties of safety guarantees. This ensures that verified programs maintain both safety guarantees and correctness guarantees throughout the verification process.

## 2.3 Discussion

In short, we believe an eBPF verifier should be powerful enough to reject unsafe programs and accept correct complex programs, and still be simple enough to be built in the kernel.

**Why not use only the C verifier?** In a single-phase C verification approach, users need to place a high level of trust in not only the C verifier but also the compiler. But even widely-used compilers such as GCC and LLVM continue to receive numerous bug reports daily. For instance, the LLVM bug tracker has numerous reports of bugs affecting various components of the compiler [22,50], such as the front end, the optimizer, and the code generator. These reports illustrate the necessity for users to trust the compiler while being aware of potential issues. Instead, we have a trustworthy bytecode verifier to ensure the final validation before loading, which is our only TCB.

**Why not use only the bytecode verifier?** Most importantly, if there is only one bytecode verifier, then it needs a strong SMT solver, which contradicts the simplicity goal of the whole verifier discussed above.

Second, the C verifier provides better computer-human interaction. An optimizing compiler may dramatically change the structure of a C program, making it impossible to convert verification results at the bytecode level back to the source level. If we only report failures at the bytecode level, users must understand how the whole toolchain works only to debug their code. Instead, the C verifier can directly generate C-level feedback, which is more understandable for developers.

A further reason is that writing an annotated bytecode program directly is challenging. Bytecode is very hard to comprehend, and thus it is even more difficult to write suitable assertions for it. Therefore, VEP provides a transformation from C assertions to bytecode assertions through an annotation-aware compiler, which enables users to write more readable assertions based on C code.

**Unsafe C programs may be compiled to a safe bytecode program. Can such programs pass VEP?** As the compiler is a black box for eBPF developers, we believe VEP should reject undefined behavior (UB) in C at the C verification level. For example, Figure 2 shows an unsafe eBPF program. Due to the insufficient space (size of 20) in *buffer* to store the information of comm, the Linux verifier rejects the function. However, a C program (shown in Figure 3) has out-of-bound array access. After being compiled into bytecode, that access may be a valid albeit unintended location, say, *buffer2*. The C standard does not require *buffer2* to be placed right after *buffer*. Compilers could choose to swap them for optimization. Advanced compilers may even detect this undefined behavior (UB) and aggressively exploit this fact by eliminating the whole C function. This program is accepted by version 5.10 of the Linux Kernel but is rejected by version 5.15. We believe that these programs need to be rejected as early as possible. VEP-C will reject all such programs at the C level. Even if such a program were to bypass VEP-C, VEP-eBPF would still reject it during bytecode verification due to the absence of required proofs generated from the C-level assertions.

```
int badhelpercall()
{   char buffer[1];
    return bpf_get_current_comm(buffer, 20); }
```

**Figure 2:** Bad helper call A from PREVAIL benchmark.

```
int badhelpercall()
{   char buffer[1];
    char buffer2[20];
    return bpf_get_current_comm(buffer, 20); }
```

**Figure 3:** Bad helper call B from PREVAIL benchmark.

**How much additional efforts are needed for writing annotations?** Typically, annotation-based verification tools require users to provide at least function specifications and loop invariants. For complex assertion derivations, users might also need to supply additional assertions to elaborate on the derivation steps. VEP offers some automation support for generating specifications and loop invariants, and we have found that, with this basic support, users can often complete program verification without *any* additional annotations. However, for complex programs, manual input is still required, particularly for verifying functional correctness.

**If one manually modifies the generated annotated bytecode, is it possible to trick the VEP-eBPF checker and break the safety guarantee?** Simply put: no. It is like you cannot modify the proof of $0 = 0$ to prove $0 = 1$.

In scenarios where a user modifies the annotated bytecode and/or its corresponding proofs, VEP-eBPF still meticulously verifies the alignment between them. If the modifications result in a correct match, meaning that the proof still validates the safety and correctness of the altered bytecode, VEP-eBPF will permit the bytecode to be loaded into the kernel. This ensures that even after modifications, as long as the integrity of the proofs is maintained, the program remains secure and is considered safe for kernel execution.

Conversely, if a user alters the annotated bytecode and introduces incorrect or inconsistent proofs, VEP-eBPF will detect these discrepancies during the verification process. The proof checker is designed to ensure that only bytecode with valid, accurate proofs can be executed within the kernel. When erroneous proofs are encountered—those that fail to substantiate the safety or correctness of the bytecode—VEP-eBPF will reject the bytecode, preventing it from being loaded into the kernel.

**How can we make sure that the pre/postconditions correctly describe the properties that we care?** Regarding whether the modified code functions as the user intends, additional proofs related to functional correctness can be provided by the user to ensure this aspect. VEP-eBPF's primary focus is to guarantee that any code it approves is safe to execute. However, the tool will not automatically verify that the modified code behaves as desired; it will only ensure that the code can be safely run without introducing security vulnerabilities. Therefore, it is up to the user to include further assertions and proofs to confirm that the program's functionality aligns with their expectations.

## 2.4 Related Work

To the best of our knowledge, VEP is the first annotation-guided eBPF verification toolchain. In theory, for any correct eBPF program written by programmers, there exists a way to add annotations to pass VEP.

Most existing eBPF verifiers, including the Linux verifier [19], PREVAIL [26], etc., are automatic verifiers. Admittedly, automatic verifiers are easy to use. But in theory, checking whether an eBPF program can be safely executed and terminated is an undecidable problem. Indeed, both the Linux verifier and PREVAIL encounter problems when handling loops with complex data structures and may generate false positive results for certain eBPF programs, which limits eBPF programs from achieving full programmability.

We have identified ExoBPF [38] and Serval [37], which necessitate users to add specifications at the bytecode level. These tools conduct symbolic execution utilizing theorem provers and subsequently employ an SMT solver to resolve constraints. Specifically, ExoBPF utilizes Lean, while Serval leverages Rosette. Although these tools are indeed powerful, they require users to be familiar with eBPF bytecode and theorem provers to effectively write bytecode specifications.

Previous research in annotation-based verification has laid important theoretical foundations. Necula established a proof-carrying code (PCC) framework where code producers generate safety proofs that can be efficiently verified by the code consumer [35, 36]. While these PCC concepts have inspired later verification systems, existing annotation-based tools like Vale [9] and Ironclad Apps [27] demonstrate limitations when adapted for eBPF verification. Vale leverages Dafny's Z3-based verification to analyze annotated assembly code for cryptographic implementations, while Ironclad Apps streamlined Z3's capabilities for OS-level verification. Though effective for assembly code verification in specific domains, these approaches impose substantial learning overhead for eBPF developers who must reason about low-level memory semantics and instruction sets. Our methodology builds upon PCC's foundational principles but shifts the verification abstraction layer - rather than requiring assembly-level annotations or specialized proof generators, we enable developers to annotate directly at the C source level. This creates an end-to-end verification chain from high-level semantics to eBPF bytecode, crucially mediated by a lightweight verification engine optimized for kernel integration.

# 3 Detailed Design of VEP-C

VEP-C is an annotation-based verifier for annotated C programs, implemented based on traditional ***symbolic execution*** algorithm with an ***entailment solver*** to check the validity of the verification conditions. It uses separation logic assertions to represent program states and performs symbolic execution. The entailment solver is based on a separation logic elimination solver and an SMT solver. During the verification process, it generates proofs for assertion derivations. In this section, we will detail the design of VEP-C, focusing on the assertion syntax, the symbolic execution process, and the verified program.

## 3.1 Verification Process of VEP-C

Throughout this subsection, we use the annotated program *memset* in Figure 4 as an example[2]. In this example, we intentionally write "`i = 0; for (; i < n;) { ...; i ++ }`" (respectively in line 5, 8, and 19) rather than the commonly used "`for (i = 0; i < n; ++ i) { ... }`" to illustrate the verification steps more conveniently. This is not a requirement or restriction in realistic

[2] To facilitate the reader's understanding, we present a version of the annotated program that leans more towards functional correctness. Assertions in actual code are more straightforward.

verification tools.

```
1  void memset(char *p1, __u32 n, char v)
2  /*@ With l1
3      Require chars(p1,n,l1)
4      Ensure ∃ l2, chars(p1,n,l2) */
5  { __u32 i = 0;
6   /*@ i == 0 && chars(p1,n,l1) */
7   /*@ Inv: ∃ l2, 0 ⩽ i ⩽ n && chars(p1,n,l2) */
8   for (; i < n; ) {
9   /*@ ∃ l2, 0 ⩽ i < n && chars(p1,n,l2) */
10  p1[i] = v;
11  /*@ ∃ l3 l2, 0 ⩽ i < n &&
12       l3[0:i] == l2[0:i] &&
13       l3[i] == v &&
14       l3[i+1:n] == l2[i+1:n] &&
15       chars(p1,n,l3) */
16  i++;
17  /*@ ∃ l3 l2, 0 ⩽ i - 1 < n &&
18       l3[0:i-1] == l2[0:i-1] &&
19       l3[i-1] == v &&
20       l3[i:n] == l2[i:n] &&
21       chars(p1,n,l3) */ }
22  /*@ ∃ l2, i == n && chars(p1,n,l2) */
23  return ; }
```

**Figure 4:** Annotated memset.

***Line 2-4**:* In the beginning, the user needs to provide a function specification, which indicates the condition that the arguments together with the initial program state need to satisfy when entering the function and the condition that needs to be satisfied when exiting the function. A function specification consists of a **With** clause, a **Require** clause, and an **Ensure** clause. The **Require** clause indicates the function precondition; the **Ensure** clause indicates the function postcondition; and the **With** clause indicates the list of logical variables mentioned in precondition which will be used in the whole program assertions. In this example, the precondition is `chars(p1, n,l1)` (line 3), which indicates that `p1` is an array of length `n`, with its data `l1` being a list of characters. The postcondition provided in line 4 specifies that the array stored at `p1` will be modified to a new character list `l2`.

***Line 5-6**:* To verify the program in Figure 4, VEP-C initiates symbolic execution based on the preconditions of the function. For instance, after processing the variable declaration and initialization at line 5, the strongest postcondition in line 6 is computed. In this example, assertions generated through symbolic execution are highlighted in red, distinguishing themselves from the assertions provided by the user, which are displayed in blue.

***Line 7-22**:* Before entering the loop, the user needs to provide a loop invariant, which indicates the property that the program state needs to satisfy at the beginning and end of each loop iteration. VEP-C will check whether the assertion before entering the function (lines 6) implies the loop invariant (line 7). If the above check passes, VEP-C will continue to symbolically execute the loop condition testing from the loop invariant. Thus, in its strongest postcondition (line 9), an additional proposition $i < n$ is added, comparing to the `Inv`. Moreover, symbolic execution will generate the strongest postcondition of the assignment statement (lines 11-15), and the strongest postcondition of the incremental step (lines 17-21). VEP-C will check whether the assertion entails the loop invariant `Inv`. The check is successful, thereby proving the correctness of the loop invariant. Subsequently, we can derive the strongest postcondition at the end of the loop, which is the strongest postcondition of `Inv` when the loop condition evaluates to false (line 22).

***Line 23**:* Finally, we perform symbolic execution on the return statement. Unlike other statements, the return statement requires the calculation of the return value (although there is no return value in this example), followed by the deallocation of all declared local variables and corresponding updates of the assertions. In this case, the variable `i` is deallocated, and the assertion is updated accordingly into `∃ l2, chars(p1,n,l2)`. The final step is to check whether the updated assertion implies the postcondition of the function. The proof here is straightforward, thus completing the verification of this example.

## 3.2 Assertion Language of VEP-C

In the example of Figure 4, we used first-order logic to describe program states. For some C programs, such assertions are sufficient to capture time costs and the changes made to memory. However, most programs manipulate more than one data structure (array, linked list, tree, etc). These data structures are stored in disjoint memory locations and such disjointness is critical in verification. For example, the C standard says that *strncpy* copies strings from source to destination, while the source and destination memory space are disjoint [30, Section 7.24.2.4]. Therefore, some naive specification like the one in Figure 5 does not correctly describes strncpy's behavior.

```
void strncpy (char *p1, char *p2, __u32 n)
/*@ With l1 l2
    Require chars(p1,n,l1) && chars(p2,n,l2)
    Ensure chars(p1,n,l2) && chars(p2,n,l2) */;
```

**Figure 5:** A naive strncpy specification.

This specification appears to be very concise, but it overlooks a critical issue: whether the two arrays, `p1` and `p2`, overlap. If there is an overlap, the behavior of the program could differ significantly, leading to potential unintended side effects that are not accounted for by this simple specification. Therefore, merely using first-order logic is insufficient to meet our needs.

State-of-the-art research has provided us with new insights, particularly through the introduction of separation logic into assertions. Tools such as VST, VeriFast, and Hip/Sleek [39] have demonstrated that using separation logic is an effective method for clearly describing properties and facilitating symbolic execution, especially in proofs related to memory properties.

Separation logic [41] uses a new connective *separating conjunction* to ensure that different names duplicate no identical addresses. The separating conjunction $P * Q$ represents the existence of two disjoint portions of the state, one that satisfies $P$ and one that satisfies $Q$. Specifically,

$$m \models P * Q \Leftrightarrow \exists m_1, m_2.\, m = m_1 \uplus m_2 \wedge m_1 \models P \wedge m_2 \models Q.$$

Here $\uplus$ means the disjoint union[3] and $m \models P$ means $m$ satisfies $P$. A distinction between $*$ and boolean conjunction $\&\&$ is that $P * P \neq P$ where $P \&\& P = P$. In particular, if $\textbf{store}(p, v)$ means that the value $v$ is stored at address $p$, $\textbf{store}(p, v) * \textbf{store}(q, u)$ implies $p \neq q$, and thus $\textbf{store}(p, v) * \textbf{store}(p, v)$ is always false: there is no way to divide a heap that a cell $p$ goes to both partitions.

Separation logic predicates can also be used to represent kernel resources [13,41], like semaphores and locks. Intuitively, when a helper function is used to gain a piece of resource, the corresponding predicate will be added to the assertion. When a helper function is used to release a piece of resource, the predicate are required to appear beforehand (which will be checked by VEP) and will then be removed. In separation logic, those resources act like memory. We show an example in Appendix E.

VEP-C uses separation-logic assertions in the canonical form of a symbolic heap. A symbolic heap is in the form $\exists \overrightarrow{x}.(P_1 \wedge \cdots \wedge P_n \wedge Q_1 * \cdots * Q_m)$, where the pure part $P$ describes memory-irrelevant properties between terms (e.g. `e1`, `e2`), and the spatial part $Q$ is a separating conjunction of spatial predicates. For example, `e1==e2` and `e1>e2` can appear as memory-irrelevant conjuncts; the empty heap predicate `emp` and the points-to predicate `store(e1,e2)` can appear as spatial conjuncts. Additionally, users can define their own predicates according to their specific needs. For example, a user can define a string array predicate such as `chars(a,e1,e2)`.

With the help of separation logic, we can write the specification shown in Figure 6, which describes the scenario stipulated by the C standard where `p1` and `p2` do not overlap in memory.

```
void strncpy (char *p1, char *p2, __u32 n)
/*@ With l1 l2
    Require chars(p1,n,l1) * chars(p2,n,l2)
    Ensure ∃ l3,
            chars(p1,n,l3) * chars(p2,n,l2) */;
```

**Figure 6:** A correct strncpy specification.

Traditionally, these terms (`e1`, `e2`, etc) above in pure parts and spatial parts should be memory-irrelevant expressions. For example, if `x` is a C variable of a struct type, then `&(x.tail)` is a memory-irrelevant expression, because computing its value does not include a load from memory. Many verification tools use this setting internally because this assertion language prevents a lot of ambiguity. For example, it may be unclear whether the predicate `store(*x, 0)` only claims the memory permission at address `*x`, or it claims the memory permission at addresses `*x` and `x` – if we take the address of storing variable `x` into consideration, it becomes even more complicated. VEP-C extends traditional symbolic heap and allows users to use memory-related expressions, which makes assertions more concise. VEP-C automatically transforms the expression to the traditional symbolic heap. In this way, users avoid a large portion of boilerplate[4]. For example, VEP-C allows users to write `*y==x→tail`, which is equivalent to:

```
∃ v yp,
store((field_addr(x,tail),v) *
store(y, yp) * store(yp, v)
```

Through the application of separation logic assertions, users gain the capability to formally specify function properties while simultaneously verifying both the fundamental safety guarantees (Properties 1 and 2 as detailed in Section 2.2) and comprehensive functional correctness properties within a unified verification framework.

**Verification when helper functions involve in** The acquisition and release of kernel resources of eBPF programs relies on helper functions, making the symbolic execution of helper function calls a critical component in the verification process. The VEP framework incorporates predefined specifications for most helper functions, formally characterizing their interactions with memory subsystems and kernel resource management. These specifications are derived from eBPF's official documentation [1]. Figure 7 presents the specifications for `bpf_sk_lookup_udp` and `bpf_sk_release`, while Figure 8 illustrates the symbolic execution behaviors of these two helper functions. The complete program is provided

[3] $A \uplus B = A \cup B$ if $A \cap B = \varnothing$, Otherwise, $A \uplus B$ is undefined. For example, $\{1,2,3,5\} = \{1,3\} \uplus \{2,5\}$, but $\{1,3\} \uplus \{1,2\}$ is undefined.

[4] The detailed syntax and transformation algorithm can be found in Appendix A and B.

in Appendix E.

```
struct bpf_sock * bpf_sk_lookup_udp
    (void *ctx, struct bpf_sock_tuple *tuple,
     __u32 tuple_size, __u64 netns, __u64 flags)
/*@ xdp_version
    Require flags == 0 &&
      valid_size(tuple_size, tuple) &&
      store_xdp(ctx) * store_sock_tuple(tuple)
    Ensure store_sock(__return) * store_xdp(ctx) *
      store_sock_tuple(tuple) */;
```

```
int bpf_sk_release (void * sock)
/*@ sock_version
    Require sock != NULL && store_sock(sock)
    Ensure ∃ v, __return == v */;
```

**Figure 7:** The specification of `bpf_sk_lookup_udp` and `bpf_sk_release`

In Figure 7, the `valid_size` predicate ensures that the value of `tuple_size` correctly corresponds to `tuple`. Specifically, if `tuple` represents an IPv4 socket, then `tuple_size` must be `sizeof(tuple→ipv4)`, and similarly for IPv6. The `store_sock(sk)` predicate ensures that if `sk` is a null pointer, it corresponds to an empty memory region. Otherwise, if `sk` is non-null, it provides access to a valid socket. This definition guarantees that memory operations on `sk` can only be performed after checking whether `sk` is null.

```
1  /*[auto generated pre]
2      ipv4(tuple) && store_xdp(ctx) *
3      store_sock_tuple(tup) */
4  sk = bpf_sk_lookup_udp(ctx, &tup,
5          sizeof(tup.ipv4), 0, 0);
6  /*[auto generated]
7      store_xdp(ctx) * store_sock_tuple(tup) *
8      store_sock(sk) */
9  ...
10 /*[auto generated pre]
11     sk != NULL && store_xdp(ctx) *
12     store_sock_tuple(tup) * store_sock(sk) */
13 bpf_sk_release(sk);
14 /*[auto generated]
15     store_xdp(ctx) * store_sock_tuple(tup) */
```

**Figure 8:** The symbolic execution result of `bpf_sk_lookup_udp` and `bpf_sk_release`

In Figure 8, we assume that `tup` is an IPv4 socket before the helper function call in line 4. Therefore, the function call on line 4 satisfies the function specification of `xdp_version`, allowing us to obtain a `store_sock(sk)`. When execution reaches line 12, we know that `sk` is non-null. Consequently, the function call on line 12 satisfies the function specification of `sock_version`, enabling us to properly reclaim the allocated socket memory.

By leveraging these built-in helper function specifications, VEP automatically enforces the verification of safety guarantee Properties 3 and 4 (as specified in Section 2.2) during symbolic execution. The full list of helper function specifications can be found in our GitHub repository.

### 3.3 Output of VEP-C

After the symbolic execution and entailment solver processes, VEP-C completes the verification of the input program. Based on the verification process, VEP-C generates a verified C program, which is then passed to the VEP-compiler to be compiled into bytecode. Figure 9 illustrates the verified C program corresponding to the program in Figure 4. Due to space constraints, we have omitted the specific assertions and proof content.

```
void memset(char *p1, __u32 n, char v)
/*@ With l1
    Require chars(p1,n,l1)
    Ensure chars(p1,n,repeat(v,n)) */
{ __u32 i = 0;
 /*@ Assertion\_1 with Proof\_1 */
 /*@ Inv with Proof\_2 and Proof\_6 */
 for (; i < n; ) {
 /*@ Assertion\_2 with Proof\_3 */
 p1[i] = v;
 /*@ Assertion\_3 with Proof\_4 */
 i++;
 /*@ Assertion\_4 with Proof\_5 */ }
 /*@ Assertion\_5 with Proof\_7 */
 return ; /*@ Proof\_8 */ }
```

**Figure 9:** Verified memset.

In this example, `Proof_1`, `Proof_3`, `Proof_4`, `Proof_5`, and `Proof_7` pertain to the verification of various safety checks during symbolic execution. These include range checks for array writes, as well as checks to ensure that there are no undefined behaviors during assignments. `Proof_2` and `Proof_6` are associated with the verification of the validity of loop invariants. `Proof_8` ensures that the function postcondition is satisfied upon completion of the function.

## 4 Detailed Design of VEP-compiler

An annotated C program undergoes several compilation passes before being converted into an annotated eBPF bytecode. While most of these passes are standard, modifications to assertions and proofs are necessary throughout the process. It's worth noting that the VEP-compiler contains only a few optimization passes at present, primarily for simplicity, which may not generate the most efficient bytecode. How to incorporate additional compilation optimizations while ensuring accurate transformation of assertions and proofs is a future work.

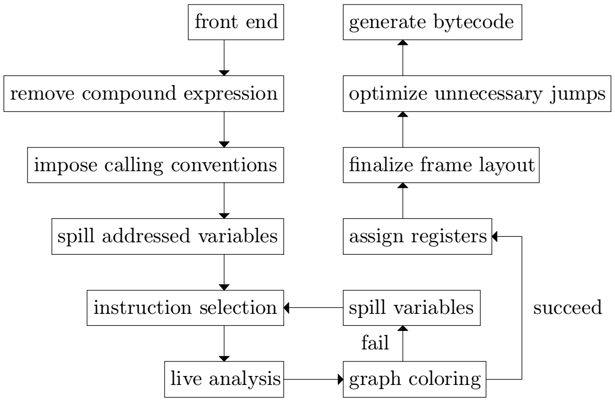

**Figure 10:** The nanopass-style compilation process

```
   //@ ∃ v, store((field_addr(x,tail),v)
                   ⇓ compile
         //@ ∃ v, store(x + 8,v)
```

**Figure 11:** IR Generation example

Figure 10 shows the whole compilation process. Every pass has a single simple and clear purpose, as in the nanopass framework [44]. We discuss the transformation during relevant passes in the rest of this section.

## 4.1 IR Generation

In our intermediate representation, we replace structure-member access by dereferencing the address of the structure plus the offset of the member. We do the same for assertions, as illustrated in Figure 11.

## 4.2 Calling Conventions

The BPF calling convention is defined as follows.

1. R0 stores the return value.
2. R1 to R5 are used to pass arguments.
3. R6 to R9 are callee saved. Others are caller-saved.

All the helper functions obey the suggested calling convention. We adopt the same convention in compiling in-program functions, so the compiler modifies function specifications to reflect the calling convention as follows.

1. Return value __return is replaced by R0.
2. Function parameters are replaced by R1 to R5.
3. State that R6 to R9 are preserved.

The generated code saves used callee-saved registers on the stack at the beginning of a procedure and restores them before returning. To convince the symbolic executor that callee-saved registers are indeed unchanged, we also modify each assertion in the procedure. They state that certain portions of the stack contain the original values of callee-saved registers.

```
strncpy:
/*@ With l1 l2 _R1 _R2 _R3 _R6 _R7 _R8 _R9
    Require
      _R1 == R1 &&
      _R2 == R2 &&
      _R3 == R3 &&
      _R6 == R6 && _R7 == R7 &&
      _R8 == R8 && _R9 == R9 &&
      chars(R1, R3, l1) * chars(R2, R3, l2)
    Ensure
      _R6 == R6 && _R7 == R7 &&
      _R8 == R8 && _R9 == R9 &&
      chars(_R1,_R3,l2) * chars(_R2,_R3,l2) */
```

**Figure 12:** Transformation of the function specification for strncpy. p1, p2, n and __return are replaced by R1, R2, R3, and R0, respectively.

Figure 12 shows the specification in Figure 5 produced by this pass. Besides the substitution defined previously, the compiler introduces auxiliary logic variables _Ri-s to relate the register values at function entry and those at the return point.

## 4.3 Register Allocation

We adopt a standard "iterated register coalescing" algorithm [25]. Most of the time, simply substituting variables in assertions with their corresponding registers (or their location on the stack, if it is spilled) is enough. The only exception is when the variable is not live (the value of the variable is not used later) at that point. In that case, the register does not necessarily hold the value of the variable: maybe another variable is using it. To make the assertion valid while retaining information, the variable should be substituted by an existentially quantified logic variable. Figure 13 shows a complete example. The third line in the source code takes the address of x, so x is spilled to the stack. p and q are assigned to the same register R1 because they are unused.

In annotated code, register allocation can sometimes yield interesting results. Figure 14 shows one example.

In the previous example, the C code clearly fails VEP-C verification, but the compiler generates valid eBPF bytecode for it. This is due to liveness analysis determining that the value of x is not live at that point. We believe rejecting such code at the VEP-C level is reasonable and does not compromise programmability. Even if one only uses VEP-compiler and VEP-eBPF, he/she can still trust the safety of the C code, because (1) the precondition is unchanged, and (2) propositions related to resources (time, memory) are unaffected.

```
int x, y, *p, *q;
x = 0; y = 1;
p = &x; q = 0;
//@ y == x + 1 && p != q
return y;
```

⇓ compile

```
*(R10 - 4) = 0
 R0 = 1
 R1 = R10
 R1 -= 4
 R1 = 0 // p and q are not used
 /*@ R1 == *(R10-4) + 1 &&
     ∃ _p _q, _p != _q */
 ret
```

**Figure 13:** An assertion after register allocation.

```
int x = 0;          R0 = 0
//@ x == 1     ⇒     //@ ∃ _x, _x == 1
return;             ret
```

**Figure 14:** Example of register allocation.

### 4.4 Frame Layout

In annotated C programs, a variable is available after we declare it. But in bytecode, there are no such declarations. So the frame—which contains spilled scalar variables, structures, and so on—should be specified in the precondition so that the symbolic executor knows which addresses are valid. Knowing how much stack space a procedure consumes is also necessary to verify that the whole program does not exceed the stack space limit.

Figure 15 shows a complete example. In the example, we assume the compiler places `l` at `R10-16`. Consequently, `R10-16` stores an integer, and `R10-8` stores a pointer to a `struct list`. The underscores indicate that their actual values are not needed.

## 5 Detailed Design of VEP-eBPF and Proof Check

VEP-eBPF processes the annotated bytecode produced by the VEP-compiler and performs symbolic execution. During this execution, when it encounters assertion entailment or safety checks typically requiring an entailment solver, VEP-eBPF distinguishes itself from VEP-C by not invoking an SMT solver. Instead, it verifies whether the proofs, generated by the SMT solver and transformed by the VEP-compiler, correctly establish the required entailments. If all entailments can be validated through their corresponding proofs, it indicates that the bytecode is safe, allowing the program to be loaded into the kernel.

VEP-eBPF's proof language is designed to include both spatial parts derivation proofs and pure propositional parts derivation proofs. The spatial parts proofs are primarily syntactic transformations of the entailment using established separation logic properties. These properties have been formalized and proved within the Coq proof assistant, ensuring their correctness. On the other hand, the pure parts proofs are based on a proof language inspired by cvc5, a well-known SMT solver. In this process, no SMT solving is required. The spatial syntax transformation and pure proof checking are handled by highly efficient algorithms.

```
struct list {
  int x;
  struct list *next;
};
int f()
/*@ Require emp
   Ensure __return == 0 */
{ struct list l;
  l.head = 0;
  return l.head; }
```

⇓ compile

```
f: /*@ Require
     *(R10-16) == _ *
     *(R10-8)  == _
   Ensure R0 == 0 */
...
```

**Figure 15:** A precondition decorated with frame information. Assume `l` is located at `R10-16`, then the compiler guarantees that `R10-16` stores an integer and `R10-8` stores a pointer to a `struct list`. The underscores indicate that they are uninitialized.

As a result, VEP-eBPF is an efficient, low-memory proof checker ideally suited for kernel-space deployment. The lightweight nature of these algorithms contributes to the overall performance, making VEP-eBPF a robust tool for verifying the safety of eBPF bytecode without the overhead of traditional SMT-based methods.

## 6 Evaluation

We evaluate the time cost, memory efficiency, and verification accuracy of VEP compared to the Linux verifier (Kernel 5.15) and PREVAIL (August 2024 version). Our benchmark selected a total of 41 programs across four categories of programs: Linux kernel samples, PREVAIL test cases, C standard library string functions, and unsafe programs with known vulnerabilities, such as out-of-bounds memory accesses and invalid function call. Detailed per-program results are provided in Appendix C (Tables 4 and 5), and all data is publicly available in our

GitHub repository[5].

### 6.1 Performance Analysis

The Linux verifier demonstrates superior stability with consistent sub-millisecond verification times (avg. 0.94 ms) and low memory consumption (<5.2 MB) across all benchmarks (Table 2). In contrast, VEP-eBPF achieves 3–5× faster verification than both PREVAIL (avg. 13.88 ms) and VEP-C (avg. 39.46 ms), while maintaining predictable memory usage of 2–3 MB – a 75% reduction compared to VEP-C's 8.5–32.6 MB footprint. This efficiency gain stems from our novel design choice discussed in Section 2.

### 6.2 Verification Accuracy

While all tools successfully reject 100% of unsafe programs (10/10 PR), critical deficiencies emerge in their handling of safe implementations. The Linux verifier demonstrates particularly severe limitations, accepting only 60% of safe programs (6/10 PR) with a 40% rejection rate - primarily stemming from inadequate loop verification (e.g., StringLib functions). PREVAIL exhibits even more erratic behavior, achieving wildly inconsistent acceptance rates from 10% (1/10 PR) on StringLib to 80% (8/10 PR) for its own test suite, a consequence of its SMT solver dependency introducing non-determinism.

This version-induced instability compounds the problem - 30% of PREVAIL samples and 20% Linux kernel programs that previously passed verification get rejected after updates, exposing fundamental flaws in their architectural design. Such brittleness severely undermines their reliability for production deployment.

In stark contrast, VEP achieves perfect verification outcomes across all benchmarks: 100% acceptance (10/10 PR) for properly annotated safe programs and 100% rejection of unsafe implementations. As long as the user provides correct annotations, VEP ensures there are no false positives and no false negatives. This deterministic behavior persists across kernel versions, demonstrating VEP's architectural superiority in balancing verification rigor with practical stability.

### 6.3 Annotation Efficiency

Tables 2 and 3 present the total lines of code, the number of user-written assertions, and the corresponding proof obligations generated by VEP. The comparison highlights that VEP requires a relatively modest number of assertions to verify programs effectively. For example, in the Linux samples with 618 lines of code, only 76 assertions were needed, resulting in 64,840 proof obligations. This indicates that a small number of high-level assertions can generate comprehensive proof coverage. Although the generated proofs are extensive, the verification process remains efficient because our proof checker is highly lightweight, ensuring minimal verification time.

Regarding the effort required to write annotated programs, we observe that adding annotations does not demand extensive modifications to the original code. Most assertions involve standard properties such as memory safety, resource bounds, and function preconditions, which can be specified with basic knowledge of program verification concepts. The expertise required is comparable to writing simple function contracts or preconditions in languages with contract support.

Furthermore, VEP supports the generation of specifications for several functions through lightweight methods, reducing manual effort. As VEP evolves, we aim to automate more of the assertion generation process, further minimizing the manual effort required and lowering the expertise barrier for new users.

### 6.4 Case Study : Key_Connection

The *Key_Connection* example (last row in Tables 2-3) demonstrates VEP's ability to verify non-eBPF network functions – specifically, an L7 filter mapping string keys to backend servers. Detailed code and the annotations for this example can be found in Appendix D. For this program, VEP generates 350 lines of annotated bytecode and 5,800 lines of proof.

This capability addresses critical limitations in current eBPF frameworks like Cilium [11], which must offload complex L7 logic to external proxies (Envoy [21]) due to verification challenges. By enabling verifiable L7 processing, VEP could eliminate proxy dependencies while maintaining safety guarantees.

## 7 Future Work

In advancing eBPF program verification, several critical areas offer promising research opportunities. This section outlines key directions for future work designed to enhance and expand the capabilities of our verification framework, aiming to make it more robust and applicable to real-world scenarios.

[5] https://github.com/yashen32768/NSDI25-VEP-535

| Programs | Total Code | Linux verifier | | | | PREVAIL | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | PR | MaxT | AvgT | MaxM | PR | MaxT | AvgT | MaxM |
| Source | Lines | | (ms) | (ms) | (KB) | | (ms) | (ms) | (KB) |
| Linux samples | 618 | 9/10 | 1.13 | 0.94 | 4196 | 10/10 | 48.37 | 12.48 | 7918 |
| PREVAIL samples | 252 | 6/10 | 1.08 | 0.71 | 4200 | 8/10 | 74.67 | 13.88 | 5279 |
| StringLib | 321 | 3/10 | 2.69 | 1.83 | 5168 | 1/10 | 39.89 | 39.89 | 7267 |
| Unsafe Programs | 195 | 10/10 | 0.051 | 0.038 | 4340 | 10/10 | 34.88 | 6.05 | 5312 |
| Key_Connection | 63 | 0/1 | 0.025 | 0.025 | 4304 | 0/1 | 2.312 | 2.312 | 5004 |

**Table 2:** Evaluation results for Linux verifier and PREVAIL.

| Programs | Total Assertion | Total Proofs | VEP-C | | | | compiler | VEP-eBPF | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | PR | MaxT | AvgT | MaxM | AvgT | MaxT | AvgT | MaxM |
| Source | Lines | Lines | | (ms) | (ms) | (KB) | (ms) | (ms) | (ms) | (KB) |
| Linux samples | 76 | 64840 | 10/10 | 158.12 | 39.46 | 32569 | 1.73 | 21.69 | 8.42 | 8034 |
| PREVAIL samples | 49 | 41931 | 10/10 | 40.23 | 9.38 | 12422 | 0.47 | 4.40 | 2.76 | 3047 |
| StringLib | 112 | 56117 | 10/10 | 36.60 | 13.47 | 12612 | 0.66 | 3.54 | 2.63 | 2970 |
| Unsafe Programs | 32 | - | 10/10 | 5.36 | 3.32 | 2955 | - | - | - | - |
| Key_Connection | 17 | 5819 | 1/1 | 16.24 | 16.24 | 8534 | 0.56 | 2.48 | 2.48 | 2440 |

**Table 3:** Evaluation results for VEP. 'PR' refers to the acceptance rate for safe programs and the rejection rate for unsafe programs.

**Towards Functional Correctness** As our demands for programs increase, memory safety alone no longer meets our development requirements. We aim to go further by supporting the verification of the functional correctness of eBPF programs. From this perspective, VEP needs to enable users to directly write proofs within C annotations or provide an interface to incorporate external proofs.

**Towards Less Annotations** Given that VEP currently requires users to provide necessary C annotations, including function preconditions, postconditions, and loop invariants, we aim to introduce additional tools to automate the generation of these annotations and reduce the user's burden. Traditional verification tools [20, 32] have already made progress in this area, and recent advancements have been achieved by integrating large language models [42, 45]. In the future, VEP will be able to adopt similar approaches to automate the generation of some annotations, thereby minimizing the user's workload.

**Towards Compilation Optimization** Advanced optimization passes may dramatically change the control flow and data flow of a program, and assertion annotations should be modified accordingly. In our current compiler, one such compilation pass is register allocation, in which multiple variables may be represented by one register (as long as their lifetime does not overlap) — we translate all these variables to the register in assertions. In the future, optimization passes may be added to eliminate some redundant instructions or redundant variables. Then, corresponding algorithms need to be designed to compile assertion annotations.

## 8 Conclusion

Since its proposal, eBPF is widely used in various domains while its verification often comes with a trade-off between safety and programmability. In this paper, we propose a two-stage automatic formal framework that addresses this issue between verification and programmability. It empowers eBPF developers to write annotations regarding the memory to guide automatic verification. As far as we are concerned, this is the first approach that could verify the safety and resource constraints of eBPF programs without compromising programmability. Specifically, a safe eBPF program can always pass VEP's check, provided that it includes sufficient annotations.

Based on this, we implement a prototype toolchain VEP. Our evaluation further demonstrates that VEP can verify the safety of complex programs with moderate overhead, which highlights the potential of the two-stage framework as a practical solution for kernel security. We believe this will enable eBPF to support more flexible and powerful programs across a wider range of domains.

**Acknowledgements** This work was supported by the NSF China (No. 62272292 and 62472274). We also thank our shepherd Rishabh Iyer and the NSDI reviewers.

## References

[1] ebpf docs. https://docs.ebpf.io/linux/helper-function/.

[2] The Cilium Authors. Cilium. https://cilium.io/.

[3] The Falco Authors. falco. https://falco.org/.

[4] The Pixie Authors. pixie. https://pixielabs.ai/.

[5] Haniel Barbosa, Andrew Reynolds, Tim King, Mudathir Mohamed, Andres Noetzli, Andrew V. Sutherland, Cesare Tinelli, Clark W. Barrett, Morgan Deters, and Mathias Preiner. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[6] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110, 2018.

[7] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, 2019. USENIX Association.

[8] Nikolaj Bjørner and Leonardo de Moura. Proofs and refutations, and z3. In *Proc. 7th International Workshop on Implementation of Logics (IWIL), CEUR Workshop Proc*, volume 418, pages 123–132, 2008.

[9] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying High-Performance cryptographic assembly code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 917–934, Vancouver, BC, August 2017. USENIX Association.

[10] Cristiano Calcagno, Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Moving fast with software verification. In *NASA Formal Methods*, pages 3–11. Springer, 2015.

[11] Cilium. Cilium Service Mesh. https://cilium.io/use-cases/service-mesh, 2024. Accessed: 2024.

[12] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[13] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, page 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 249–264, New York, NY, USA, 2024. Association for Computing Machinery.

[15] eBPF Development Team. ebpf. https://ebpf.io/, 2022.

[16] The eBPF Development Team. Bounded loops in bpf for the 5.3 kernel. https://lwn.net/Articles/794934/.

[17] The eBPF Development Team. bpf: add bpf_for_each_map_elem helper. https://lwn.net/Articles/846504/.

[18] The eBPF Development Team. bpf: add search pruning optimization and tests. https://lwn.net/Articles/614226/.

[19] The eBPF Development Team. ebpf verifier. https://docs.kernel.org/bpf/verifier.html.

[20] Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. Ilinva: Using abduction to generate loop invariants. In Andreas Herzig and Andrei Popescu, editors, *Frontiers of Combining Systems - 12th International Symposium, FroCoS 2019, London, UK, September 4-6, 2019, Proceedings*, volume 11715 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2019.

[21] Envoy. Envoy is an open source edge and service proxy, designed for cloud-native applications. https://www.envoyproxy.io, 2024. Accessed: 2024.

[22] Yuyou Fan and John Regehr. High-throughput, formal-methods-assisted fuzzing for llvm. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 349–358, 2024.

[23] William Findlay, Anil Somayaji, and David Barrera. Bpfbox: Simple precise process confinement with ebpf. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 91–103, 2020.

[24] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[25] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, may 1996.

[26] 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 *PLDI*, pages 1069–1084, 2019.

[27] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-End security via automated Full-System verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, Broomfield, CO, October 2014. USENIX Association.

[28] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[29] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *CoNEXT*, pages 54–66, 2018.

[30] International Organization for Standardization. *ISO/IEC 9899:2018: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, 2018.

[31] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 41–55, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[32] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. SLING: using dynamic analysis to infer program invariants in separation logic. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 788–801. ACM, 2019.

[33] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[34] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[35] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[36] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, page 229–243, New York, NY, USA, 1996. Association for Computing Machinery.

[37] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with serval. In *SOSP*, 2019.

[38] Luke Nelson, Xi Wang, and Emina Torlak. A proof-carrying approach to building correct and flexible in-kernel verifiers. https://lpc.events/event/

11/contributions/944/attachments/893/1707/2021-09-23-lpc21.pdf.

[39] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 251–266. Springer, 2007.

[40] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[41] Peter O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

[42] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[43] Alastair Robertson. bpftrace. https://github.com/iovisor/bpftrace.

[44] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. *SIGPLAN Not.*, 39(9):201–212, sep 2004.

[45] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 7762–7773, Red Hook, NY, USA, 2018. Curran Associates Inc.

[46] Calico Development Team. Calico. https://www.tigera.io/project-calico/.

[47] The Aqua Tracee Development Team. tracee. https://www.aquasec.com/products/tracee/.

[48] The Coq Development Team. Coq. https://coq.inria.fr/.

[49] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. revisiting the open vswitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 245–257, 2021.

[50] Qiushi Wu, Zhongshu Gu, Hani Jamjoom, and Kangjie Lu. Gnnic: Finding long-lost sibling functions with abstract similarity. *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[51] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. {XRP}:{In-Kernel} storage functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, 2022.

[52] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. Bpf for storage: an exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 128–135, 2021.

[53] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. Vst-a: A foundationally sound annotation verifier, 2023.

# A Syntax of VEP-C Assertions

In this section, we discuss the syntax of VEP-C assertions, which are divided into two main categories: user assertions and internal assertions. User assertions enable developers to annotate C code with specific directives, whereas internal assertions support the tool's symbolic execution and reasoning processes.

## A.1 Syntax of User Assertions

The primary design principle of user assertions is to align as closely as possible with the habits of C programmers, while also ensuring sufficient expressiveness for formal reasoning at the logical level. We begin by defining the expressions used in user assertions, followed by an explanation of how to construct a complete assertion. Lastly, we will discuss the structure of function specifications within this framework.

The expressions are inductively defined in Figure 16. The only thing unfamiliar is custom functions. They are functions in the *logic* world. For example, we provide a built-in function `rev`$(l)$ that reverses a list (a list of values, not a data structure in C).

A user assertion can consist of several branches, each of which is divided into a section that describes pure facts and another that describes memory aspects. The syntax tree is defined in Figure 17. The semantics of user assertions is a little different from textbook separation logics, because of its HCI nature. To convince yourself that our transformation explained later makes sense, imagine we normalize every memory-related expression to a form `*`$l$. We assume two $l$s are different simply by their syntax. Then, each proposition—whether pure or spatial—describes such memory locations in addition to their spatial part, if any. When they are connected through conjunction `&&` or separating conjunction `*`, we union these locations; when connected through `||`, we keep them local to each case.

Just like custom functions, users can introduce custom predicates. For example, our built-in predicate `Ebpf_map`$(m)$ denotes that an eBPF map is stored at a location $m$.

A function specification consists of three parts: logical variable list (**With** clause), precondition (**Require** clause), and postcondition (**Ensure** clause).

**With clause.** An implicit "∀" around the specification. They are sometimes essential to relate the program state at the function exit point to the one at the function entry. They can even appear in annotations inside the function so that developers have convenient access to the initial state. In general, their values should be "determinable" from the precondition.

**Require clause.** The precondition describes properties that the program state at function entry should satisfy.

**Ensure clause.** The postcondition describes properties that the program states when the function exits should satisfy. We use the keyword `__return` to refer to the return value, if any.

The syntax of function specifications is formally defined in Figure 18:

## A.2 Syntax of Internal Assertions

We use a canonical form of assertions, internal assertions, actual symbolic execution, constraint solving, and everything. They have a simpler structure and clearer semantics (paradoxically, user-friendly assertions have *complex* semantics). Each internal assertion consists of several branches; each of which includes four parts: **Exist**, **Local**, **Prop**, and **Sep**. **Exist** is a list of existential variables, either explicitly written by developers or automatically generated during transformation (explained later). **Local** describes the relations between program variables and the logic world. For example, in C, `&`$v = e$ means that the address of the program variable $v$ is expression $e$; in eBPF bytecode, $r = e$ means the register $r$ stores value $e$. **Prop** stores pure facts. **Sep** stores memory-related facts. The syntax of internal assertions is defined in Figure 19.

We will use an example to clearly illustrate the distinction between user assertions and internal assertions. Consider the following annotation in a C program:

```
exists v, y = &x && x == 2 * v &&
          forall n, v != n * n
```

An equivalent internal assertion looks like this:

```
Ex    : v vx vy
Local : &x = px,  &y = py
Prop  : vy == px && vx == 2 * v &&
        forall n, n != n * n
Sep   : store(px, int, vx)
      * store(py, int, vy)
```

In this example, the user assertion describes an integer variable $x$, whose value is an even number that is not a perfect square; $y$ is a pointer to $x$. The internal assertion introduces two new existential variables $vx$ and $vy$ to represent the current values of the program variables $x$

| | | | | |
|---|---|---|---|---|
| Expression | $e$ | ::= | $z \in \mathbb{Z}_{64}$ | integer literal |
| | | $\mid$ | $v_p \mid v_l$ | C variable, logic variable |
| | | $\mid$ | $\texttt{\&}e \mid \texttt{*}e$ | address, dereference |
| | | $\mid$ | $\odot e \mid e_1 \oplus e_2$ | unary and binary operation |
| | | $\mid$ | $e_1\texttt{[}e_2\texttt{]}$ | array indexing |
| | | $\mid$ | $e\texttt{.}m \mid e\texttt{->}m$ | member access |
| | | $\mid$ | $\texttt{sizeof}(t)$ | type size |
| | | $\mid$ | $f(e_1, \ldots, e_n)$ | custom function |
| Unary operator | $\odot$ | ::= | $\texttt{!} \mid \texttt{\~{}} \mid \texttt{-} \mid \texttt{+} \mid \ldots$ | |
| Binary operator | $\oplus$ | ::= | $\texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \ldots$ | |

**Figure 16:** Syntax of expressions.

| | | | | |
|---|---|---|---|---|
| Pure proposition | $P$ | ::= | $e_1 \oplus e_2$ | comparison |
| | | $\mid$ | $P_1 \ \texttt{\&\&}\ P_2$ | conjunction |
| | | $\mid$ | $P_1 \ \texttt{\|\|}\ P_2$ | disjunction |
| | | $\mid$ | $P_1 \ \texttt{->}\ P_2$ | implication |
| | | $\mid$ | $\texttt{!}P$ | negation |
| | | $\mid$ | $\texttt{exists}\ x, P$ | existential quantification |
| | | $\mid$ | $\texttt{forall}\ x, P$ | universal quantification |
| | | $\mid$ | $p(e_1, \ldots, e_n)$ | custom predicate |
| Spatial proposition, assertion | $S$ | ::= | $P$ | |
| | | $\mid$ | $P \ \texttt{\&\&}\ S$ | conjunction |
| | | $\mid$ | $S_1 \ \texttt{*}\ S_2$ | separating conjunction |
| | | $\mid$ | $S_1 \ \texttt{\|\|}\ S_2$ | disjunction |
| | | $\mid$ | $\texttt{exists}\ x, S$ | existential quantification |
| | | $\mid$ | $p(e_1, \ldots, e_n)$ | custom predicate |
| Comparison operator | $\oplus$ | ::= | $\texttt{>} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{!=}$ | |

**Figure 17:** Syntax of user assertions.

| | | | |
|---|---|---|---|
| Function specification | $F$ | ::= | $(\forall v_1, \ldots, v_n . S_{pre}, S_{post})$ |

**Figure 18:** Syntax of function specification.

| | | | | |
|---|---|---|---|---|
| Assertion | $A$ | ::= | $H$ | singleton |
| | | $\mid$ | $A \vee H$ | disjunction |
| Heap | $H$ | ::= | $\exists x_1, \ldots, x_n . L \wedge P \star S$ | |
| Local | $L$ | ::= | $\bullet$ | empty |
| | | $\mid$ | $\texttt{\&}v = e, L$ | address |
| Pure proposition | $P$ | ::= | $e_1 \oplus e_2$ | comparison |
| | | $\mid$ | $\neg P \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \mid P_1 \rightarrow P_2$ | connective |
| | | $\mid$ | $\exists x . P \mid \forall x . P$ | quantification |
| | | $\mid$ | $p(e_1, \ldots, e_n)$ | custom predicate |
| Spatial proposition | $S$ | ::= | $\bullet$ | empty heap |
| | | $\mid$ | $S_1 \star S_2$ | separating conjunction |
| | | $\mid$ | $e_1 \mapsto e_2\ \mathsf{type}\, t$ | store |
| | | $\mid$ | $e_1 \mapsto -\ \mathsf{type}\, t$ | uninitialized store |
| | | $\mid$ | $p(e_1, \ldots, e_n)$ | custom predicate |
| Expression | $e$ | ::= | $z \in \mathbb{Z}$ | integer constant |
| | | $\mid$ | $x$ | logic variable |
| | | $\mid$ | $\odot e \mid e_1 \oplus e_2$ | arithmetic |
| | | $\mid$ | $\&e_1 \rightarrow m$ | member offset |
| | | $\mid$ | $f(e_1, \ldots, e_n)$ | custom function |

**Figure 19:** (Abstract) Syntax of C internal assertions.

$$
\begin{array}{llll}
\text{Normal form} & N & ::= & \overrightarrow{H} \\
\text{Heap} & G, H & ::= & \exists \overrightarrow{x}.P \,\&\&\, \overrightarrow{B} \\
\text{Body} & B & ::= & p(e_1, \ldots, )
\end{array}
$$

**Figure 20:** Syntax of DNF.

and $y$, respectively. It is more structured and facilitates symbolic execution more conveniently.

## B Transformation To Internal Assertion

In this section, we show how user assertions are transformed into internal assertions.

### B.1 Step 1: Normalization

This phase puts an assertion in a sort of disjunctive normal form (DNF). Such a normal form is defined in Figure 20. The normalization procedure is described in Figure 21.

### B.2 Step 2: Convert Expressions

In this phase, we convert expressions in user assertions to expressions in internal assertions, at the same time generating auxiliary **Local** and **Sep**. The procedure is defined in Figure 22. The judgement $\mathcal{V}(e, L, S, x) = (e', L', S', x')$ means that under context $L, S, x$, the value of $e$ is $e'$ and the context is updated to $L', S', x'$. $\mathcal{A}$ is similar, computing addresses (if valid) instead of values. The conversion is naturally extended to pure propositions $\mathcal{T}(P)$ and spatial propositions $\mathcal{T}(B)$, replacing each expression $e$ in them with $\mathcal{V}(e)$, which we omit here.

### B.3 Step 3: Sanity Check

Our pure propositions $P$ can contain spatial expressions, so they are not really pure. Consequently, some user assertions cannot be transformed into internal ones. For example, the following user assertion is invalid.

$$\texttt{forall}\ i,\ 0 <= i \ \texttt{\&\&}\ i < 4 => a\texttt{[}i\texttt{]} == 0$$

A simple scope checking on internal assertions is enough to exclude such invalid cases. That is, every logic variable appearing in $S$ should be bound by existential variables introduced in `exists` $x, S$.

## C Full Evaulations

Full test results of previous work and VEP are shown in Table 4 and Table 5, respectively.

## D Case Study : Key_connection

Figure 23 presents the complete Key_connection program, though only the assertions used to verify memory safety are shown here. In this program, `connt→conn_list` is a linked list of servers to connect to. To describe the memory structure related to this linked list, we introduce two predicates: `ConnListrep` and `ConnListseg`. These predicates are defined as follows:

```
ConnListrep(x) := x == 0 && emp ||
    ∃ n l num mark_v length,
      0 ⩽ n && n < 128 && l[n] == 0 &&
      x → num_packet == num &&
      x → mark == mark_v &&
      x → lengthsofar == length &&
      store_char_array(&(x→key), 128, l) *
      ConnListrep(x → next)

ConnListseg(x,y) := x == y && emp ||
    ∃ n l num mark_v length,
      0 ⩽ n && n < 128 && l[n] == 0 &&
      x → num_packet == num &&
      x → mark == mark_v &&
      x → lengthsofar == length &&
      store_char_array(&(x→key), 128, l) *
      ConnListseg(x → next, y)
```

In this definition, `store_char_array` corresponds to `chars` in Figure 4, representing a character array. This is purely a recursive definition describing the server list.

## E Case Study : Acquire/Release example

Figure 24 presents an example that uses helper functions for acquiring and releasing resources. In this example, line 13 invokes the helper function `bpf_sk_lookup_udp` to obtain a socket. The success of this operation is checked at line 17, and if successful, the socket is released at line 19 using the helper function `bpf_sk_release`. As a result, this program maintains kernel-interface compliance level safety.

In this example, the user only needs to provide the annotations in lines 3 to 5 to successfully verify the program. VEP automatically generates the assertions in lines 13-15 and 19-20. Suppose the user omits the null check for sk at line 16 and directly returns. In that case, VEP will report an error due to the presence of an unverified

$$norm(P) = P \qquad norm(p(e_1,\ldots,e_n)) = p(e_1,\ldots,e_n) \qquad \frac{norm(S) = \exists\overrightarrow{x_1}.P_1 \texttt{\&\&} \overrightarrow{B_1},\ldots,\exists\overrightarrow{x_n}.P_n \texttt{\&\&} \overrightarrow{B_n}}{norm(\exists x,S) = \exists\overrightarrow{x,x_1}.P_1 \texttt{\&\&} \overrightarrow{B_1},\ldots,\exists\overrightarrow{x,x_n}.P_n \texttt{\&\&} \overrightarrow{B_n}}$$

$$\frac{norm(S) = \exists\overrightarrow{x_1}.P_1 \texttt{\&\&} \overrightarrow{B_1},\ldots,\exists\overrightarrow{x_n}.P_n \texttt{\&\&} \overrightarrow{B_n}}{norm(P \;\texttt{\&\&}\; S) = \exists\overrightarrow{x_1}.(P \texttt{\&\&} P_1) \texttt{\&\&} \overrightarrow{B_1},\ldots,\exists\overrightarrow{x_n}.(P \texttt{\&\&} P_n) \texttt{\&\&} \overrightarrow{B_n}} \qquad \frac{norm(S_1) = \overrightarrow{H_1} \qquad norm(S_2) = \overrightarrow{H_2}}{norm(S_1 \;\texttt{||}\; S_2) = \overrightarrow{H_1 H_2}}$$

$$\frac{norm(S_1) = H_1,\ldots,H_n \qquad norm(S_2) = G_1,\ldots,G_m}{norm(S_1 \;\texttt{*}\; S_2) = merge(H_1,G_1),\ldots,merge(H_1,G_m),\ldots,merge(H_n,G_m)}$$

$$merge(\exists\overrightarrow{x_1}.P_1 \texttt{\&\&} \overrightarrow{B_1}, \exists\overrightarrow{x_2}.P_2 \texttt{\&\&} \overrightarrow{B_2}) = \exists\overrightarrow{x_1 x_2}.(P_1 \texttt{\&\&} P_2) \texttt{\&\&} \overrightarrow{B_1 B_2}$$

**Figure 21:** Normalization procedure.

$$\mathcal{V}(z,C) = (z,C) \qquad \mathcal{V}(v_l,C) = (v_l,C) \qquad \mathcal{V}(\texttt{sizeof}(t),C) = (\mathsf{sizeof}(t),C) \qquad \mathcal{V}(\texttt{\&}e,C) = \mathcal{A}(e,C)$$

$$\frac{\mathcal{V}(e,C) = (e',C')}{\mathcal{V}(\odot e,C) = (\odot e',C')} \qquad \frac{\mathcal{V}(e_1,C) = (e'_1,C') \qquad \mathcal{V}(e_2,C') = (e'_2,C'')}{\mathcal{V}(e_1 \oplus e_2,C) = (e'_1 \oplus e'_2,C'')}$$

$$\frac{\mathcal{V}(e_1,C) = (e'_1,C_1) \qquad \cdots \qquad \mathcal{V}(e_n,C_{n-1}) = (e'_n,C_n)}{\mathcal{V}(f(e_1,\ldots,e_n),C) = (f(e'_1,\ldots,e'_n),C_n)} \qquad \frac{\mathcal{A}(e,C) = (a,L,S,x) \qquad a \in \mathrm{dom}(S)}{\mathcal{V}(e,C) = (S(a),L,S,x)}$$

$$\frac{\mathcal{A}(e,C) = (a,L,S,x) \qquad a \notin \mathrm{dom}(S) \qquad v \text{ fresh}}{\mathcal{V}(e,C) = (v,L,S\{a \mapsto v\},x \cup \{v\})} \qquad \mathcal{A}(\texttt{*}e,C) = \mathcal{V}(e,C) \qquad \mathcal{A}(e_1\texttt{[}e_2\texttt{]},C) = \mathcal{V}(e_1\texttt{+}e_2,C)$$

$$\frac{\mathcal{A}(e,C) = (e',C')}{\mathcal{A}(e\,\texttt{.}m,C) = (e' + \mathsf{offset}(m),C')} \qquad \frac{\mathcal{V}(e,C) = (e',C')}{\mathcal{A}(e\texttt{->}m,C) = (e' + \mathsf{offset}(m),C')} \qquad \frac{v_p \in \mathrm{dom}(L)}{\mathcal{A}(v_p,L,S,x) = (L(v_p),L,S,x)}$$

$$\frac{v_p \notin \mathrm{dom}(L) \qquad a \text{ fresh}}{\mathcal{A}(v_p,L,S,x) = (a,L\{v_p \mapsto a\},S,x \cup \{a\})}$$

$$\frac{\mathcal{T}(P,\{\},\{\},\{\}) = (P',C) \qquad \mathcal{T}(\overrightarrow{B},C) = (\overrightarrow{B'},L,S,y)}{\mathcal{T}(\exists\overrightarrow{x}.P \texttt{\&\&} \overrightarrow{B}) = \exists x_1,\ldots,x_n,y_1,\ldots,y_m.L \wedge P' \circledast_i B'_i \circledast_{a\in\mathrm{dom}(S)} a \mapsto S(a)} \qquad \mathcal{T}(\overrightarrow{H}) = \bigvee \overrightarrow{\mathcal{T}(H)}$$

**Figure 22:** Expression conversion.

```
#include "bpf.h"

struct connection{
    char key[128];
    unsigned int num_packet;
    unsigned int mark;
    unsigned int lengthsofar;
    struct connection * next;
};

struct identifier{
    int mark;
    char  name[32];
    char  pattern[512];
};

struct conntrack{
    int queuenum;
    int iden_num;
    struct identifier *iden_array;
    struct connection *conn_list;
};

struct connection * get_connection
    (struct conntrack * connt, char * key)
/*@ With n0 m0 l0 Connlist
    Require 0 ⩽ n0 && l0[n0] == 0 && n0 < m0 &&
        connt → conn_list == Connlist &&
        store_char_array(key, m0, l0) *
        ConnListrep(Connlist)
    Ensure ∃ v, __return == v && TT
*/
{
    struct connection * p ;
    if(connt == NULL || key == NULL)
        return NULL;
    p = connt→conn_list;
    /*@ Inv
        0 ⩽ n0 && l0[n0] == 0 && n0 < m0 &&
        connt → conn_list == Connlist &&
        store_char_array(key, m0, l0) *
        ConnListrep(p) * ConnListseg(Connlist, p)
    */
    while(p != NULL){
        if(strcmp(p→key , key) == 0)
            break;
        p = p→next;
    }
    if(p == NULL)
        return NULL;
    else
        return p;
}
```

**Figure 23:** Annotated Key_connection

memory region store_sock(sk), whose validity is uncertain. Similarly, if the user fails to release sk at line 18, VEP will also raise an error at the final return statement.

```
1   SEC("xdp")
2   int prog(struct xdp_md *ctx)
3   /*@ Require store_xdp(ctx)
4       Ensure ∃ v, __return == v && store_xdp(ctx)
5   */
6   {
7     struct bpf_sock_tuple tup;
8     struct bpf_sock *sk;
9     if (!check_ipv4_udp(ctx)) return XDP_DROP;
10    init_sock_tuple(ctx, &tup);
11    sk = bpf_sk_lookup_udp(ctx, &tup,
12          sizeof(tup.ipv4), 0, 0);
13    /*[auto generated]
14      store_xdp(ctx) * store_sock_tuple(tup) *
15          store_sock(sk) */
16    if (!sk) return XDP_DROP;
17
18    bpf_sk_release(sk);
19    /*[auto generated]
20      store_xdp(ctx) * store_sock_tuple(tup) */
21    return XDP_PASS;
22  }
```

**Figure 24:** An example for Acquire/Release by helper function

| **Programs** | | Code Lines | Linux verifier | | PREVAIL | |
|---|---|---|---|---|---|---|
| | | | Time(ms) | Memory(KB) | Time(ms) | Memory(KB) |
| Linux Samples | sockex1_kern | 29 | 1.03 | 4194 | 2.05 | 4978 |
| | syscall_tp_kern(enter) | 38 | 1.03 | 4194 | 3.17 | 4931 |
| | cpustat_kern(frequency) | 93 | 1.13 | 4190 | 11.30 | 6377 |
| | cpustat_kern(idle) | 116 | 1.11 | 4196 | 23.52 | 7918 |
| | xdp_adjust_tail_kern | 47 | 0.64 | 4196 | 6.69 | 5539 |
| | syscall_tp_kern(exit) | 35 | 0.99 | 4190 | 3.02 | 4927 |
| | lathist_kern(on) | 77 | 1.09 | 4144 | 10.18 | 6025 |
| | trace_event_kern | 65 | fail | - | 48.37 | 6730 |
| | tcp_iw_kern | 68 | 0.65 | 4152 | 9.82 | 5664 |
| | tcp_rwnd_kern | 50 | 0.77 | 4155 | 6.69 | 5404 |
| PREVAIL Samples | twomaps | 34 | fail | - | 2.43 | 5056 |
| | twotypes | 33 | fail | - | 3.80 | 5278 |
| | map_in_map | 36 | fail | - | 5.17 | 5054 |
| | stackok | 13 | 1.02 | 4114 | 74.67 | 5279 |
| | loop | 21 | fail | - | fail | - |
| | packet_start_ok | 14 | 1.08 | 4200 | 1.19 | 4942 |
| | twostackvars | 47 | 0.53 | 4140 | 20.9 | 5267 |
| | packet_access | 28 | 0.58 | 4153 | 2.74 | 5216 |
| | bpf2bpf | 13 | 0.51 | 4154 | 0.16 | 4157 |
| | dependent_read | 13 | 0.55 | 4154 | fail | - |
| Stringlib | strcpy | 34 | fail | - | fail | - |
| | strncpy | 34 | 1.65 | 4212 | fail | - |
| | strcat | 44 | fail | - | fail | - |
| | strncat | 43 | fail | - | fail | - |
| | strlen | 19 | fail | - | fail | - |
| | strncmp | 31 | 2.69 | 4316 | fail | - |
| | strcmp | 32 | fail | - | fail | - |
| | memset | 28 | 1.14 | 5168 | 39.89 | 7267 |
| | strchr | 28 | fail | - | fail | - |
| | memchr | 28 | fail | - | fail | - |
| Unsafe Program | badhelpercall | 6 | reject | - | reject | - |
| | badmapptr | 24 | reject | - | reject | - |
| | badrelo | 20 | reject | - | reject | - |
| | ctxoffset | 21 | reject | - | reject | - |
| | nullmapref | 23 | reject | - | reject | - |
| | badhelpercall2 | 22 | reject | - | reject | - |
| | packet_overflow | 14 | reject | - | reject | - |
| | wronghelper | 20 | reject | - | reject | - |
| | mapunderflow | 23 | reject | - | reject | - |
| | packet_reallocate | 22 | reject | - | reject | - |
| Key_connection | | 63 | fail | - | fail | - |

**Table 4:** Time cost and memory usage of samples by Linux verifier and PREVAIL

| **Programs** | | Assertion | Proof | VEP-C | | VEP-compiler | VEP-eBPF | |
|---|---|---|---|---|---|---|---|---|
| | | Lines | Lines | Time(ms) | Memory(KB) | Time(ms) | Time(ms) | Memory(KB) |
| Linux Samples | sockex1_kern | 3 | 1140 | 4.06 | 5882 | 0.35 | 2.53 | 2284 |
| | syscall_tp_kern(enter) | 7 | 2253 | 5.33 | 6391 | 0.37 | 2.57 | 2396 |
| | cpustat_kern(frequency) | 11 | 8357 | 50.33 | 15887 | 2.62 | 7.89 | 4292 |
| | cpustat_kern(idle) | 11 | 19390 | 158.12 | 32569 | 6.09 | 21.32 | 7167 |
| | xdp_adjust_tail_kern | 10 | 739 | 4.27 | 5852 | 0.32 | 2.41 | 2236 |
| | syscall_tp_kern(exit) | 7 | 2253 | 5.28 | 6396 | 0.35 | 2.47 | 2379 |
| | lathist_kern(on) | 9 | 4180 | 23.47 | 18502 | 2.47 | 21.69 | 8034 |
| | trace_event_kern | 6 | 7136 | 67.33 | 18841 | 1.15 | 5.79 | 3353 |
| | tcp_iw_kern | 6 | 161 | 12.57 | 12150 | 1.75 | 9.11 | 4182 |
| | tcp_rwnd_kern | 6 | 19231 | 63.88 | 19154 | 1.84 | 8.44 | 4342 |
| PREVAIL Samples | twomaps | 2 | 2310 | 6.07 | 7119 | 0.46 | 2.77 | 2521 |
| | twotypes | 4 | 9449 | 15.49 | 8288 | 0.73 | 4.40 | 2665 |
| | map_in_map | 3 | 261 | 2.97 | 5995 | 0.39 | 2.31 | 2348 |
| | stackok | 4 | 1848 | 4.32 | 5223 | 0.19 | 1.96 | 2028 |
| | loop | 8 | 4042 | 10.02 | 7682 | 0.55 | 2.84 | 2453 |
| | packet_start_ok | 3 | 1245 | 2.64 | 5174 | 0.21 | 2.03 | 2126 |
| | twostackvars | 12 | 18446 | 40.23 | 12422 | 1.43 | 3.96 | 3047 |
| | packet_access | 3 | 3669 | 8.85 | 7075 | 0.50 | 3.42 | 2659 |
| | bpf2bpf | 7 | 13 | 0.70 | 4436 | 0.11 | 1.76 | 1932 |
| | dependent_read | 3 | 648 | 2.56 | 4920 | 0.19 | 2.14 | 2106 |
| Stringlib | strcpy | 9 | 6051 | 12.09 | 8172 | 0.66 | 2.67 | 2510 |
| | strncpy | 11 | 4242 | 8.69 | 7888 | 0.59 | 2.61 | 2557 |
| | strcat | 17 | 12820 | 31.17 | 11778 | 1.39 | 3.41 | 2907 |
| | strncat | 17 | 14254 | 36.60 | 12612 | 1.16 | 3.54 | 2970 |
| | strlen | 9 | 2035 | 5.02 | 5946 | 0.29 | 2.23 | 2236 |
| | strncmp | 11 | 3976 | 8.45 | 7994 | 0.54 | 2.58 | 2470 |
| | strcmp | 9 | 5934 | 14.27 | 8528 | 0.76 | 2.25 | 2574 |
| | memset | 11 | 1925 | 3.78 | 6271 | 0.35 | 2.18 | 2281 |
| | strchr | 9 | 2329 | 6.94 | 6962 | 0.40 | 1.66 | 2463 |
| | memchr | 9 | 2551 | 7.64 | 6683 | 0.42 | 3.19 | 2517 |
| Unsafe Program | badhelpercall | 4 | reject | 2.82 | 1618 | - | - | - |
| | badmapptr | 3 | reject | 3.05 | 1701 | - | - | - |
| | badrelo | 3 | reject | 2.60 | 1575 | - | - | - |
| | ctxoffset | 3 | reject | 2.70 | 1436 | - | - | - |
| | nullmapref | 3 | reject | 4.38 | 2275 | - | - | - |
| | badhelpercall2 | 4 | reject | 2.46 | 1402 | - | - | - |
| | packet_overflow | 3 | reject | 4.31 | 2115 | - | - | - |
| | wronghelper | 3 | reject | 2.73 | 1652 | - | - | - |
| | mapunderflow | 3 | reject | 2.78 | 1781 | - | - | - |
| | packet_reallocate | 3 | reject | 5.36 | 2955 | - | - | - |
| Key_connection | | 17 | 5819 | 16.24 | 8534 | 0.56 | 2.48 | 2440 |

**Table 5:** Time cost and memory usage of samples by VEP