A Flow-Sensitive Refinement Type System for Verifying eBPF Programs

AMEER HAMZA, Florida State University, USA
LUCAS ZAVALÍA, Florida State University, USA
ARIE GURFINKEL, University of Waterloo, Canada
JORGE A. NAVAS, Certora Inc, USA
GRIGORY FEDYUKOVICH, Florida State University, USA

The Extended Berkeley Packet Filter (eBPF) subsystem within an operating system's kernel enables userspace programs to extend kernel functionality dynamically. Due to the security risks associated with runtime modification of the operating system, eBPF requires all programs to be verified before deploying them within the kernel. Existing approaches to eBPF verification are monolithic, requiring their entire analysis to be done in a secure environment, resulting in the need for extensive trusted codebases. We present a type-based verification approach that automatically infers proof certificates in userspace, thus reducing the size and complexity of the trusted codebase. At the same time, only the proof-checking component needs to be deployed in a secure environment. Moreover, compared to previous techniques, our type system enhances the debuggability of the programs for users through ergonomic type annotations when verification fails. We implemented our type inference algorithm in a tool called VeRefine and evaluated it against an existing eBPF verifier, Prevail. VeRefine outperformed Prevail on most of the industrial benchmarks.

CCS Concepts: • **Theory of computation** → **Type theory**; **Abstraction**; **Program verification**; • **Networks** → *Network monitoring*; • **Security and privacy** → **Information flow control**; *Operating systems security*.

Additional Key Words and Phrases: Flow-Sensitivity, Refinement Types, Type Inference, Memory Safety, Information Flow Safety, Automated Reasoning about Low-Level Code

ACM Reference Format:
Ameer Hamza, Lucas Zavalía, Arie Gurfinkel, Jorge A. Navas, and Grigory Fedyukovich. 2025. A Flow-Sensitive Refinement Type System for Verifying eBPF Programs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 410 (October 2025), 28 pages. https://doi.org/10.1145/3763799

1 Introduction

The eBPF technology [42] allows userspace code to run in the privileged context of the kernel in Linux and Windows operating systems. Evolved from the Berkeley Packet Filter (BPF) project [34], which originally aimed to filter network traffic at a low level, it has substantially more capabilities, including network observability, process monitoring, and tracing now. For example, in Kubernetes systems, all pods in a given node run on the same kernel, allowing eBPF programs to observe processes in every container in every pod on the node. However, the kernel, being a secure environment, may be harmed by such untrusted programs. For example, it is hypothetically possible to manipulate pointers inside of eBPF such that the address of a pointer is sent to the userspace.

Authors' Contact Information: Ameer Hamza, Florida State University, Tallahassee, USA, ah18r@fsu.edu; Lucas Zavalía, Florida State University, Tallahassee, USA, lrzavalia@fsu.edu; Arie Gurfinkel, University of Waterloo, Waterloo, Canada, arie.gurfinkel@uwaterloo.ca; Jorge A. Navas, Certora Inc, Austin, USA, jorge@certora.com; Grigory Fedyukovich, Florida State University, Tallahassee, USA, grigory@cs.fsu.edu.

ACM 2475-1421/2025/10-ART410
https://doi.org/10.1145/3763799

This, in turn, could allow malicious agents to reveal critical information about the kernel's memory layout. Similarly, if an eBPF program were to crash, the kernel itself may crash as well. Thus, eBPF requires all programs to be verified to ensure that no vulnerabilities occur.

Although formal verification of software is challenging in general, some limitations on the expressiveness and complexity of eBPF programs make it more convenient to verify them. Specifically, eBPF programs cannot allocate memory dynamically; they can access only a certain number of memory regions, use a single thread only, and have an upper bound on the number of instructions [24]. Even then, there remain many aspects to eBPF programs that still present a challenge for verification, such as the extensive use of pointer arithmetic and register spilling [24].

Two common verifiers in use today are the Linux Verifier [1] and Prevail [24]. The former runs inside the kernel and verifies the safety of eBPF programs by exploring all possible execution paths while tracking register values and certain memory contents. However, its verification approach is conservative and heuristic-driven rather than being grounded in formal theory, which can lead to both unnecessary rejections and inefficiencies in certain cases, for example, path explosion in loops and unbounded state growth handling multiple execution paths. Prevail is based on Abstract Interpretation, and thus is more efficient than the Linux Verifier. However, verifying an eBPF program involves performing numerous computationally expensive checks, all of which must be conducted within a secure environment. The secure environment must trust the verification process, but the complexity and computational cost of these intensive verification procedures make this impractical. Neither approach is user-friendly: they generate either verbose or no verification logs, making it difficult for users to debug unsafe programs.

Our goal is to avoid having to deploy the entire verifier's codebase in a secure environment. Our approach, inspired by Proof-Carrying Code (PCC) [37], separates the process of generating a proof from the process of validating it. At a high level, the process of generating proof certificates is carried out in userspace, while a comparatively simpler process running in the secure environment uses the proof certificates and a validation algorithm to trust the code with which it was sent. *Type systems* are particularly well-suited for implementing PCC architectures. Powered by algorithms for both type inference (i.e., generating proof certificates) and type checking (i.e., validating proof certificates), they offer a reasonably efficient static verification method.

In this paper, we present a type system for eBPF and two safety properties, namely *Region Safety* and *Information-Flow Safety*. Region Safety ensures that no memory access happens outside the bounds of the program's designated memory regions. Information-Flow Safety ensures that no sensitive information (for example, pointer addresses) is exposed to the user. In order to conduct these safety checks, we use a low-level, flow-sensitive, refinement type system for bytecode representations of eBPF programs. Flow-sensitivity [5, 6] tracks the type of a variable based on its use in different branches/basic blocks, allowing a variable to have multiple types at various points in the program. Information-flow safety requires that pointers are never stored in a memory region accessible to the user. Flow-sensitivity enables determining the type of data stored at each store instruction. Refinement types [21, 43] enable fine-grained information about program states to be encoded as types using quantifier-free first-order predicates. Predicates could express, e.g., that a variable is a number whose value is within a certain range. Our analysis relies on this to check for region safety, i.e., by asserting that no memory access happens outside of a designated region and using refined types to guarantee that a pointer is within a certain range of possible values.

Our type-inference approach explores the control-flow graph (CFG) of an eBPF bytecode. It infers types for each instruction in each basic block that represent information about the program registers and memory cells. It automatically handles branching in the graph, inferring types at join points as the least upper bounds of the types from each path. When a type at a program location cannot be inferred, a type error is generated, stating that the program might be unsafe. The result is either a program fully annotated with types (in case of successful inference) or a partially annotated program (in case of failure). Type annotations improve debugging by allowing users to track information at each program point. They help users identify the root cause of verification failures and determine whether they deal with actual bugs or false positives. Users get a chance to learn about program behaviors without prior knowledge of the specifics of the underlying type system.

Once the types have been inferred and the information about the applied rules has been stored, a type-checking procedure could easily validate the types at each program location in linear time with respect to program locations.

We implemented our type inference algorithm in a new tool called VeRefine utilizing Prevail's parsing backend. The implementation of VeRefine is over 9000 lines of C++ code. We evaluated VeRefine against Prevail on a set of 420 publicly available benchmarks, including benchmarks from industry-based projects (i.e., Linux, Cilium, OVS, Falco, Suricata, Prototype-Kernel, and Beyla). VeRefine automatically verified a comparative number of benchmarks as Prevail. It took only a fraction of the time taken by Prevail and provided better debuggability capabilities to users.

To sum up, our work improves eBPF program verification in the following aspects:

(1) Our PCC-inspired infrastructure for eBPF region and information-flow safety verification requires only proof-checking to be done in a secure environment, while computationally intensive proof-generation is done in the userspace.
(2) Our flow-sensitive, refinement type system to check the safety of eBPF programs, along with an algorithm for inferring variable types, demonstrates successful adaptation of techniques developed in prior work for eBPF and other languages.
(3) The eBPF programs annotated with types improve the debugging capabilities without requiring them to understand the underlying type system to find the root causes for failures.
(4) The new VeRefine tool automatically handles eBPF benchmarks from industry-based projects faster than the state-of-the-art Prevail verifier.

2 Motivating Example

Fig. 1 gives an example bytecode with type annotations. We write `r1 : `**stk**[506] to mean that register `r1` contains a pointer to a memory region **stk** at offset 506. Here, **stk**[506] is a type annotation for `r1`. We write `stack[a-b]` to represent a stack cell at a valid address `a` with width `(b-a+1)`. Similar to type annotations for registers, a cell `stack[a-b]` can be annotated with a type `t`, stated as `stack[a-b] : t` that represents that cell `stack[a-b]` stores type `t`. Annotations are colored in red to distinguish them from the actual bytecode in Fig. 1. Initially known types are given before basic block `bb0` (lines 1 and 2). Variables s_i (where each i is a number) represent symbolic numeric values. Any relevant variable types updated or accessed by an instruction, but which cannot be shown with the instruction itself, are shown after it. All extra annotations not shown with the instructions are highlighted in yellow to distinguish them from the actual bytecode.

With these type annotations, the verifier reports an error at line 22: "`Map key size is not singleton`", a false positive result (the program is safe but reported unsafe). We only show the relevant parts of the bytecode for brevity. The cause of the error is that the second argument, `r2`, to `map_lookup_elem` should point to a memory location storing a known numeric value, but it does not. The error roots back, through a series of stack loads and stores and assignments to the `probe_read` function call at line 4. The `probe_read` function allows programs to safely read memory from an arbitrary user-space or kernel-space address pointed to by `r3` of size `r2`, and store the read contents in the memory location pointed to by `r1`. Since the memory address is arbitrary, it might not always be possible to model the location as a pointer in the analysis; hence, `r3` is safely

```
 1 r1 : stk[506], r2 : num⟨2⟩, r3 : num,
 2 r6 : map_1, r10 : stk[512]
 3 bb0:
 4   r0 : num = probe_read(
 5         r1 : stk[506], r2 : num⟨2⟩,
 6         r3 : num);
 7   stack[506-507] : num
 8   goto bb1;

10 bb1:
11   r1 : num = *(u16 *)(r10 - 6);
12   *(u32 *)(r10 - 44) = r1;
13   stack[468-471] : num
14   goto bb2;

15 bb2:
16   r1 : num = *(u32 *)(r10 - 44);
17   *(u32 *)(r10 - 4) = r1;
18   stack[508-511] : num
19   r2 : stk[512] = r10;
20   r2 : stk[508] += -4;
21   r1 : map_1 = r6;
22   r0 : num = map_lookup_elem(
23         r1 : map_1, r2 : stk[508]);
24   stack[508-511] : num
25   goto EXIT;
```

Fig. 1. Example type-annotated bytecode where the analysis produces a false positive result.

typed as a numeric value. The analysis cannot statically infer the contents read, which should be a known numeric value, and it later manifests as an error at line 22.

We explore the root cause of the error with the help of annotations produced by the type analysis. The type of `stack[508-511]` (annotation given at line 24) indicates that the issue relates to the stack store operation at line 17 and, further, to the load operation at line 16. Tracing back to line 12, we identify the store operation at the stack memory location `(r10 - 44)`, and further the load operation from location `(r10 - 6)` to be relevant, benefiting from the annotations. Finally, we find line 4 containing `probe_read` function call that stores a number at location `(r10 - 6)` (or offset `506` in stack), but without knowing an exact value. This helps us understand that the error is a false result due to a lack of precision when analyzing `probe_read`.

To sum up, this example demonstrates the utility of type annotations in debugging eBPF programs. The root cause and the location of the error often span across multiple basic blocks. Our type annotations help in tracing the error back to its root cause, which is difficult with only error messages and limited debugging information provided by existing tools. For comparison, in Sec. 7.1, Fig. 11 gives the debug logs returned by Prevail (so-called *invariants*) for the same example, which do not provide insights to the developers.

3 Background on eBPF

eBPF programs are typically written using a restricted dialect of C or Rust. The eBPF subsystem requires a very specific compilation process in which eBPF programs are compiled to an intermediate bytecode. The original BPF would run the bytecode in an interpreter inside the kernel; eBPF, on the other hand, has upgraded to a JIT compiler instead of an interpreter. A requirement for the eBPF ecosystem is that the eBPF programs must be formally verified to be safe before running in the kernel space.

To transmit data back to the user space, eBPF relies on BPF maps, which are key-value data structures accessible from both kernel and userspace. eBPF programs can interact with maps as well as various other eBPF data structures through a library of helper functions.

The eBPF instruction set is given in Fig. 2. The syntax closely follows the notation used in the documentation for eBPF instructions [2] with some differences involving memory (loads and stores) and jump/branching operations.

There are 10 general-purpose registers labeled `r0`,..., `r9`, and one specialized read-only register `r10`, which contains a stack frame pointer. Memory operations use the notation $:=_{sz}$ that show access with sz bytes. The `goto` and `assume` operations are not part of the standard eBPF instruction set. The `goto` with a single target mimics an unconditional jump to another basic block. The `goto` with two targets, together with the `assume` statements, mimic conditional jumps to two different basic blocks (a branch). The `assume` statements at the beginning of each block encode the condition under which the flow jumps to the block. Calls to eBPF helper functions are expressed in C-style notation, i.e., $f(C_0, \ldots, C_n)$.

$$
\begin{array}{rcl@{\qquad}rcl}
P & ::= & I \;;\; P \mid I & B & ::= & R \approx C \\
I & ::= & *A :=_{sz} C \mid R :=_{sz} *A & C & ::= & R \mid n \\
 & & \mid R := E \mid \texttt{assume}\ B \mid \texttt{goto}\ L & \oplus & \in & \{*, /, |, \&, <<, >>, >>>, \%, \hat{}\,\} \\
 & & \mid \texttt{goto}\ L, L \mid f(C_0, \ldots, C_n) & \sim & \in & \{\texttt{htobe}, \texttt{htole}, \texttt{bswap}, \texttt{neg}\} \\
R & ::= & \texttt{r0} \mid \texttt{r1} \mid \texttt{r2} \mid \texttt{r3} \mid \texttt{r4} \mid \texttt{r5} & \approx & \in & \{=, \neq, <, \leq, >, \geq\} \\
 & & \mid \texttt{r6} \mid \texttt{r7} \mid \texttt{r8} \mid \texttt{r9} \mid \texttt{r10} & n & \in & \mathbb{Z} \\
E & ::= & C \mid \sim(R) \mid \texttt{loadmap\_fd}\ n & sz & \in & \{1, 2, 4, 8\} \\
 & & \mid R \oplus C \mid R \pm C & L & \in & \mathit{Labels} \\
A & ::= & (R + n) \mid (R - n) & f & \in & \mathit{BpfHelperFunctions}
\end{array}
$$

Fig. 2. Grammar for eBPF instructions.

Every eBPF program can access a set of memory regions. The stack region has a fixed size, 512 bytes, and is used for register spilling, scratch memory, and transferring parameters to eBPF helper functions. The context region is fixed-size (size known at compile-time) and contains information passed to it by the point in the operating system where the program is attached. The precise layout of the context region depends on the specific attachment type specified in the eBPF program [42]. When the eBPF program involves networking, the context region contains important information about the network packet, namely pointers identifying the beginning of the packet, the end of the packet, and the beginning of the metadata associated with the packet. The size of the packet region is unknown at compile-time, hence the above packet pointers are needed for packet region bounds reasoning. Finally, a memory region is called shared if it is created in an eBPF program when either 1) a value from a map is looked up using a map key, or 2) there is an access to an implementation-specific variable (called platform variable) to provide runtime information to the eBPF programs [3]. Both map values and platform variables enable information flow between the kernel and userspace. As expected, multiple shared regions can be generated by an eBPF program.

4 Programming Model

This section introduces eBPF types and environments to be used in type rules in Sec. 5.

4.1 Types

The syntax for types in our type system is given in Fig. 3. Types can be assigned to both registers and memory cells. We collectively refer to these as *variables*. Types are broadly characterized into pointer types, numeric types, maps, and function types. In addition, the types ANY and NONE are the top and bottom types for the type system, respectively. The type **num** represents a value interpreted as an integer, while types ρ represent values interpreted as pointers to some memory region. These types are further refined using $\{v : \mathbf{num} \mid \varphi(v)\}$ and $\{v : \rho \mid \varphi(v)\}$, respectively. The variable v is called the value variable and must occur in formula φ. To accommodate the BPF map type, we use the notation $\mathbf{map}_i$, where i indicates a unique numeric index for the map. The eBPF helper functions are typed as $\tau_0 \times ... \times \tau_n \rightarrow \tau$.

Terms are broadly characterized as numeric terms t_n and pointer terms t_p. Numeric terms consist of integers n and symbolic variables s_i. Variables s_i are referred to as *slack* variables [12] and are considered global and static. We use slack variables to encode interval/range types as well as to track offsets for packet pointers. Pointer terms consist of additional symbols *meta*, *begin*, and *end*,

$$\begin{aligned}
\text{(Types) } \tau &::= \tau'_0 \times \cdots \times \tau'_i \to \tau' \mid \tau' \\
\text{(Types) } \tau' &::= r \mid \textsc{any} \mid \textsc{none} \mid \mathbf{map}_i \mid \{\nu : r \mid \varphi\} \\
\text{(Refinable Types) } r &::= \rho \mid \mathbf{num} \\
\text{(Region Types) } \rho &::= \mathbf{pkt} \mid \mathbf{stk} \mid \mathbf{ctx} \mid \mathbf{shared}_i \\
\text{(Constraints) } \varphi &::= \varphi \wedge \varphi \mid t_n \approx t_n \mid t_p \approx t_p \mid \nu = t_n \mid \mathit{offset}(\nu) = t_p \mid s_i \in [n, m] \mid \top \mid \bot \\
\text{(Numeric Terms) } t_n &::= t_n \pm t_n \mid n \mid s_i \\
\text{(Pointer Terms) } t_p &::= t_p \pm t_n \mid \mathit{meta} \mid \mathit{begin} \mid \mathit{end} \\
\text{(Integers) } n, m &\in \mathbb{Z}, \quad \text{(Naturals) } i \in \mathbb{N}, \quad \text{(Comparisons) } \approx \in \{=, \neq, <, \leq, >, \geq\}
\end{aligned}$$

Fig. 3. Grammar for eBPF types.

```
1  r1 : {pkt[begin + s0] | s0 ∈ [0,4]}
2  r2 : pkt[end]
3  bb0:
4    r3 : {pkt[begin + s0] | s0 ∈ [0,4]} := r1;
5    r3 : {pkt[begin + s0 + 8] | s0 ∈ [0,4]} += 8;
6    goto bb1;
```

```
6  bb1:
7    assume(r3 <= r2);
8    r1 : {pkt[begin + s0] | begin + s0 + 8 ≤ end ∧ s0 ∈ [0,4]}
9    r0 : num :=8 *(r1);
```

Fig. 4. Motivation for using slack variables.

which are used to represent offsets for the beginning of the metadata region before packet, the beginning of the packet, and the end of the packet region, respectively. Note that for all eBPF programs, we assume that $\mathit{meta} \leq \mathit{begin} \leq \mathit{end}$.

The language of constraints is built out of the language of terms. Specifically, it supports comparisons between numeric terms and comparisons between pointer terms. The possible values for slack variables can be restricted to be in a specific interval of integers, written $s_i \in [n, m]$. There are distinguished equality comparisons $\nu = t_n$ and $\mathit{offset}(\nu) = t_p$ that constrain the value variable for refined types; in practice, a valid refined type must have exactly one such constraint. We distinguish between ν and $\mathit{offset}(\nu)$ because numeric types refine the value they represent directly, but pointer types refine the offsets of pointers. Finally, constraints consist of conjunctions of other constraints or logic constants $\top$ and $\bot$. Refined types by constant $\bot$ are considered a type error and reflect scenarios where there is contradictory information about a piece of data.

It is convenient to introduce syntax sugar for common types and give auxiliary definitions:

$$\begin{aligned}
\mathbf{num}\langle t_n \rangle &\equiv \{\nu : \mathbf{num} \mid \nu = t_n\} \\
\{\mathbf{num}\langle t_n \rangle \mid \varphi\} &\equiv \{\nu : \mathbf{num} \mid \nu = t_n \wedge \varphi\} \text{ where } \nu \notin \mathit{Vars}(\varphi) \\
\rho[t_p] &\equiv \{\nu : \rho \mid \mathit{offset}(\nu) = t_p\} \\
\{\rho[t_p] \mid \varphi\} &\equiv \{\nu : \rho \mid \mathit{offset}(\nu) = t_p \wedge \varphi\} \text{ where } \nu \notin \mathit{Vars}(\varphi)
\end{aligned}$$

- the set of all types generated by the grammar in Fig. 3: $\mathcal{T} = L(\tau)$,
- the set of all instructions generated by the grammar in Fig. 2: $\mathcal{I} = L(P)$,
- the set of all registers: $\mathcal{V} = \{\texttt{r0}, \ldots, \texttt{r10}\}$,
- the set of all region types: $\mathfrak{R} = \{\mathbf{ctx}, \mathbf{stk}, \mathbf{pkt}\} \cup \{\mathbf{shared}_i \mid i = 0, 1, 2, \ldots\}$,
- the set of standard sizes: $\mathcal{S} = \{1, 2, 4, 8\}$.

To motivate slack variables as compared to intervals, Fig. 4 gives a snippet with the safe load operation at line 9. Because constraint $\mathit{begin} + s_0 + 8 \leq \mathit{end}$ holds at line 7, an access of 8 bytes at pointer location $\mathbf{pkt}[\mathit{begin} + s_0]$ is safe. By contrast, if intervals were used, the type of r3 after line 5 would be $\mathbf{pkt}[\mathit{begin} + [8, 12]]$. This type at line 7 implies constraint $\mathit{begin} + 8 \leq \mathit{end}$ because the intervals represent numeric values in multiple executions of a program. This causes the load operation at line 9 to fail as the type of `r1` is $\mathbf{pkt}[begin + [0, 4]]$. Since safety verification takes into account all possible executions, one of the executions might do an access at $begin + 4$ for 8 bytes, which fails given $begin + 8 \leq end$.

The slack variables provide no more information than intervals; however, their specific use case improves the analysis. Intuitively, the slack variables mimic the sense of a single execution out of multiple executions. They enable a pattern of "check" and "use", where slacks relate the "check" (`assume` applied to `r3` at line 7) to the "use" (`load` operation to `r1` at line 9). Sometimes, instead of using slack variables, we could directly relate registers (for example, at line 4, relate `r3` and `r1` through the equality relation). However, this is not guaranteed to work in the flow-sensitivity setting, as registers may get assigned at any point, and such relations might then be incorrect.

Such an analysis could also be done using a Single Static Assignment (SSA) encoding for the program. Since variables in SSA encoding only get assigned once, variables could directly be used inside the refining constraints, instead of using slack variables. However, slack variables target a very specific use, i.e., to ensure that packet accesses are within bounds. For whole eBPF programs or parts of eBPF programs not dealing with packet pointers, using an SSA encoding might be an expensive operation. Furthermore, eBPF programs are commonly, though not exclusively, distributed in compiled object file format, and as such, we cannot assume access to high-level intermediate representations like LLVM IR that are often used to obtain SSA forms. Thus, a complete SSA encoding must be generated from raw bytecode for every individual program, which is both impractical and unnecessary for our verification goals. Our slack variable approach thus provides a lightweight and on-demand substitute for SSA, applied only where needed to ensure memory safety without imposing a heavy transformation burden on the entire program.

4.2 Environments

Refinement types have been successfully applied for verification in the context of functional languages that are stateless and where mutation is not allowed. But in the context of a low-level imperative language, program variables may change, possibly invalidating the refinements in a type. Thus, in our type system, the judgments about the typing of instructions are relative to the instruction's position in the program. This means that it needs an ability to "update" and "access" the typing of a register. To capture such behaviors, we define the type environment Γ as follows:

Definition 1. *A* type environment $\Gamma \subseteq \mathcal{V} \times \mathcal{T}$ *is a mapping from the set of registers* $\mathcal{V}$ *to types* $\mathcal{T}$. *To update the type of a register* R *in* Γ, *we write* $\Gamma[R \mapsto \tau]$. *To access the type of a register* R *stored in* Γ, *we write* $\Gamma(R)$.

In addition to Γ, the type system needs an abstract representation for the memory regions of the eBPF programs. We model the memory regions as tuples containing a region type $\rho \in \mathfrak{R} \setminus \mathbf{pkt}$, a natural number $size_\rho$ representing the size of the region ρ, and a set of *cells* that contain the type/size/offset triplets representing the data stored in the region. The packet region is excluded from this definition, as its size is unknown at compile time.

Definition 2. *Let* $\mathcal{R} = (\rho, size_\rho, C_\rho)$ *be a tuple where* $\rho \in \mathfrak{R} \setminus \mathbf{pkt}$ *is a region type,* $size_\rho \in \mathbb{N}$ *is the size of the region, and* $C_\rho \subseteq \mathcal{T} \times \mathbb{N} \times \mathbb{N}$ *is the set of tuples representing memory cells. Then,* $\mathcal{R}$ *is a region if* $\forall(\tau, sz, i) \in C_\rho, i + sz \leq size_\rho$.

In this context, we abuse notation slightly by using the region type identified by ρ as a label to identify regions so as to minimize the symbol usage and because each eBPF program has a well-defined number of regions. The eBPF memory environment constitutes the set of memory regions, defined by the context region (labeled **ctx**), the stack region (labeled **stk**), and possibly multiple shared regions (labeled $\mathbf{shared}_i$ for some index i).

Definition 3. *A set Δ of memory regions is an eBPF* memory environment *if it meets three conditions: (1) there is exactly one region $\mathcal{R} \in \Delta$ with $\rho = \mathbf{ctx}$, (2) there is exactly one region $\mathcal{R} \in \Delta$ with $\rho = \mathbf{stk}$, and (3) there are zero or more regions $\mathcal{R} \in \Delta$ of the form $\rho = \mathbf{shared}_i$ each with a unique index $i \in \mathbb{N}$.*

eBPF programs begin in a well-defined initial state. Below we give some intuition for how the initial state is encoded in our abstractions of eBPF programs (written Γ_{init} and Δ_{init}) and how these evolve throughout the analysis of the program. The registers of all eBPF programs begin in the same state, while the initial state of the memory regions, specifically the context region, is determined by the eBPF program's attachment hook inside the operating system.

(1) Initially, `r1` always points to the context region at offset 0, and `r10` points to the top of the stack. $\Gamma_{init}(\texttt{r1}) = \mathbf{ctx}[0]$, $\Gamma_{init}(\texttt{r10}) = \mathbf{stk}[512]$. All other registers are set to ANY in Γ_{init}.
(2) The stack region is initially empty ($C_{\mathbf{stk}} = \emptyset$), and Δ_{init} contains an entry $(\mathbf{stk}, 512, \emptyset)$.
(3) The contents of the context region, $C_{\mathbf{ctx}}$, are deduced using a tuple of four natural numbers called the context descriptor $\mathcal{D} = \{size_{\mathbf{ctx}}, m, b, e\}$, where $size_{\mathbf{ctx}}$ represents the size of the context region, and m, b, e numbers represent the locations inside the context where packet pointers $\mathbf{pkt}[meta]$, $\mathbf{pkt}[begin]$, and $\mathbf{pkt}[end]$ are stored, resp. C_{ctx} is defined as follows:

$$C_{ctx} = \begin{cases} \emptyset & \text{if } m = b = e = -1 \\ \{(\mathbf{pkt}[meta], 4, m), (\mathbf{pkt}[begin], 4, b), (\mathbf{pkt}[end], 4, e)\} & \text{otherwise} \end{cases}$$

Δ_{init} contains an entry $(\mathbf{ctx}, size_{ctx}, C_{\mathbf{ctx}})$. The packet pointers are stored as 4 bytes.
(4) For each shared region labeled $\mathbf{shared}_i$, its $size_{\mathbf{shared}_i}$ is known at compile time, and its content is initially empty. Hence Δ_{init} contains entries $(\mathbf{shared}_i, size_{\mathbf{shared}_i}, \emptyset)$.
(5) We allow "strong updates" through stack pointers due to their role in register spilling and keeping track of parameters for eBPF helper functions. The contents of the stack region, $C_{\mathbf{stk}}$, are tracked throughout the analysis. The contents of the other regions are not needed for our analysis, and we do not track the strong updates through their pointers to keep the analysis simpler. Once their initial states are defined, no more updates happen. However, we always care about the region safety and information-flow safety for all regions when load and store operations are analyzed. We discuss this in more detail in Sec. 5.

When performing the load and store operations on memory regions, an extra "overlap" function is required. For any region ρ, given an offset n and width sz of a memory operation, it checks whether any cells are already present in the region in the offset range $[n, n + sz - 1]$.

Definition 4. *Let Δ be an eBPF memory environment, $n \in \mathbb{N}$ be an offset, and $sz \in \mathcal{S}$ be the width. Then, the set of overlapping cells $\textsc{Overlap}_\rho(sz, n)$:*

$$\textsc{Overlap}_\rho(sz, n) = \{(\tau', sz', i) \in C_\rho \mid [i, i + sz' - 1] \cap [n, n + sz - 1] \neq \emptyset\}$$

When updating a memory region ρ with a cell (τ, sz, n), intuitively, we exclude all cells in C_ρ that overlap with (τ, sz, n), and then insert (τ, sz, n) in C_ρ.

Definition 5. *Let Δ be a memory environment, $\tau \in \mathcal{T}$ be a type, $n \in \mathbb{N}$ be an offset, and $sz \in \mathcal{S}$ be a width. Then, an update $\Delta_\rho[\tau, sz, n]$ to a memory region $\rho \in \Delta$:*

$$\Delta_\rho[\tau, sz, n] = \Delta \setminus \{(\rho, size_\rho, C_\rho)\} \cup \{(\rho, size_\rho, C_\rho \setminus \textsc{Overlap}_\rho(sz, n) \cup \{(\tau, sz, n)\})\} \quad (1)$$

As discussed before, we only allow updates to the stack region, we only use $\Delta_\rho[\cdot, \cdot, \cdot]$ for $\rho = \mathbf{stk}$.

$$\frac{\Delta, \Gamma \vdash E : \tau}{\Delta, \Gamma \vdash R := E \dashv \Delta, \Gamma[R \mapsto \tau]}\ \textsc{T-Assign} \qquad \frac{n \in [-2^{63}, 2^{63} - 1]}{\Delta, \Gamma \vdash n : \mathbf{num}\langle n \rangle}\ \textsc{T-Num} \qquad \frac{\Gamma(R) = \tau}{\Delta, \Gamma \vdash R : \tau}\ \textsc{T-Reg}$$

$$\frac{\Delta, \Gamma, x_0 : \tau_0, \ldots, x_n : \tau_n \vdash f : \tau_0 \times \ldots \times \tau_n \to \tau}{\Delta, \Gamma \vdash f(x_0, \ldots, x_n) \dashv \Delta', \Gamma[\texttt{r0} \mapsto \tau, \texttt{r1} \mapsto \textsc{any}, \ldots, \texttt{r5} \mapsto \textsc{any}]}\ \textsc{T-Fun}$$

$$\frac{}{\Delta, \Gamma \vdash \texttt{goto}\ \ L_0, L_1 \dashv \Delta, \Gamma}\ \textsc{T-Goto2} \qquad \frac{}{\Delta, \Gamma \vdash \texttt{goto}\ \ L \dashv \Delta, \Gamma}\ \textsc{T-Goto1}$$

$$\frac{\Delta, \Gamma \vdash \iota_1 \dashv \Delta', \Gamma' \qquad \Delta', \Gamma' \vdash \iota_2 \dashv \Delta'', \Gamma''}{\Delta, \Gamma \vdash \iota_1; \iota_2 \dashv \Delta'', \Gamma''}\ \textsc{T-Sequence} \qquad \frac{\begin{array}{c}\Delta, \Gamma \vdash R : \{\mathbf{num}\langle t_n \rangle \mid \varphi\} \\ \Delta, \Gamma \models \tau = \textsc{IntervalizeU}(\sim,\ \{\mathbf{num}\langle t_n \rangle \mid \varphi\})\end{array}}{\Delta, \Gamma \vdash {\sim}R : \tau}\ \textsc{T-Unary}$$

$$\frac{\Delta, \Gamma \vdash C_0 : \{\mathbf{num}\langle t_n \rangle \mid \varphi\} \qquad \Delta, \Gamma \vdash C_1 : \{\mathbf{num}\langle t'_n \rangle \mid \varphi'\}}{\Delta, \Gamma \vdash C_0 \pm C_1 : \{\mathbf{num}\langle t_n \pm t'_n \rangle \mid \varphi \wedge \varphi'\}}\ \textsc{T-Binary}$$

$$\frac{\begin{array}{c}\Delta, \Gamma \vdash C_0 : \{\mathbf{num}\langle t_n \rangle \mid \varphi\} \qquad \Delta, \Gamma \vdash C_1 : \{\mathbf{num}\langle t'_n \rangle \mid \varphi'\} \\ \Delta, \Gamma \models \tau = \textsc{IntervalizeB}(\oplus,\ \{\mathbf{num}\langle t_n \rangle \mid \varphi\},\ \{\mathbf{num}\langle t'_n \rangle \mid \varphi'\})\end{array}}{\Delta, \Gamma \vdash C_0 \oplus C_1 : \tau}\ \textsc{T-BinAlt}$$

$$\frac{\Delta, \Gamma \vdash R : \{\rho[t_p] \mid \varphi\} \qquad \Delta, \Gamma \vdash C : \{\mathbf{num}\langle t_n \rangle \mid \varphi'\}}{\Delta, \Gamma \vdash R \pm C : \{\rho[t_p \pm t_n] \mid \varphi \wedge \varphi'\}}\ \textsc{T-PtrNum}$$

$$\frac{\Delta, \Gamma \vdash R_1 : \{\rho[t_p] \mid \varphi\} \qquad \Delta, \Gamma \vdash R_2 : \{\rho[t'_p] \mid \varphi'\}}{\Delta, \Gamma \vdash R_1 - R_2 : \{\mathbf{num}\langle t_p - t'_p \rangle \mid \varphi_1 \wedge \varphi_2\}}\ \textsc{T-PtrSub}$$

$$\frac{\Delta, \Gamma \vdash R_1 : \{\mathbf{num}\langle t_n \rangle \mid \varphi\} \qquad \Delta, \Gamma \vdash R_2 : \{\mathbf{num}\langle t'_n \rangle \mid \varphi'\}}{\Delta, \Gamma \vdash \mathit{assume}(R_1 \approx R_2) \dashv \Delta, \Gamma[R_1 \mapsto \{\mathbf{num}\langle t_n \rangle \mid t_n \approx t'_n \wedge \varphi \wedge \varphi'\}]}\ \textsc{T-AssumeNum}$$

$$\frac{\Delta, \Gamma \vdash R_1 : \{\rho[t_p] \mid \varphi\} \qquad \Delta, \Gamma \vdash R_2 : \{\rho[t'_p] \mid \varphi'\}}{\Delta, \Gamma \vdash \mathit{assume}(R_1 \approx R_2) \dashv \Delta, \Gamma[R_1 \mapsto \{\rho[t_p] \mid t_p \approx t'_p \wedge \varphi \wedge \varphi'\}]}\ \textsc{T-AssumePtr}$$

$$\frac{n \in [0,\ 2^{32}]}{\Delta, \Gamma \vdash \texttt{loadmap\_fd}\ \ n : \mathbf{map}_n}\ \textsc{T-Map} \qquad \frac{\begin{array}{cc}\Delta, \Gamma \vdash R_1 : \{\nu : r \mid \varphi\} & \nu \notin \mathrm{Vars}(\psi) \\ \Delta, \Gamma \vdash R_2 : \{\nu : r \mid \varphi'\} & \vdash \varphi \Rightarrow \psi\end{array}}{\Delta, \Gamma \vdash R_2 : \{\nu : r \mid \varphi' \wedge \psi\}}\ \textsc{T-Prop}$$

Fig. 5. Type rules for control, arithmetic, and logic instructions.

Finally, an eBPF memory environment Δ together with a type environment Γ can make the following judgments: type assignment ($\Delta, \Gamma \vdash x : \tau$), type equality ($\Delta, \Gamma \models \tau = \tau'$), subtyping ($\Delta, \Gamma \models \tau <: \tau'$), and type/memory environment update ($\Delta, \Gamma \vdash \iota \dashv \Delta', \Gamma'$).

5 Type Rules

In this section, we explore the rules that make up our type system.

5.1 Arithmetic, Control and Logic Instructions

We begin with the rules for arithmetic, logic, and control flow instructions. The rules are given in figure 5. For brevity, we use the abbreviations $\oplus$ and $\approx$ as defined in Fig. 3. Rule T-Assign handles an assignment of one register to another, updating the type of the destination register to that of the source register. The rule T-Num is used to type an immediate value in an expression, while rule T-Reg is used to type a register in an expression. All numeric values are integers in

Algorithm 1: IntervalizeB: Procedure for simplifying numeric types

```
Input: An operation $\oplus$, a type $\tau_0 = \{\mathbf{num}\langle t_n \rangle \mid \varphi\}$, a type $\tau_1 = \{\mathbf{num}\langle t'_n \rangle \mid \varphi'\}$
Output: A type $\{\mathbf{num}\langle s_j \rangle \mid s_j \in [a_j, b_j]\}$
1 $(s_{i_0} + \cdots + s_{k_0} + n_0,\ s_{i_0} \in [a_{i_0}, b_{i_0}] \wedge \cdots \wedge s_{k_0} \in [a_{k_0}, b_{k_0}]) \leftarrow \textsc{normalize}(\tau_0)$
2 $(s_{i_1} + \cdots + s_{k_1} + n_1,\ s_{i_1} \in [a_{i_1}, b_{i_1}] \wedge \cdots \wedge s_{k_1} \in [a_{k_1}, b_{k_1}]) \leftarrow \textsc{normalize}(\tau_1)$
3 $[\ell_0, u_0] \leftarrow [a_{i_0} + \cdots + a_{k_0} + n_0,\ b_{i_0} + \cdots + b_{k_0} + n_0]$
4 $[\ell_1, u_1] \leftarrow [a_{i_1} + \cdots + a_{k_1} + n_1,\ b_{i_1} + \cdots + b_{k_1} + n_1]$
5 $a_j \leftarrow \min(\{u \oplus v \mid (u, v) \in [\ell_0, u_0] \times [\ell_1, u_1]\})$
6 $b_j \leftarrow \max(\{u \oplus v \mid (u, v) \in [\ell_0, u_0] \times [\ell_1, u_1]\})$
7 return $\{\mathbf{num}\langle s_j \rangle \mid s_j \in [a_j, b_j]\}$
```

Algorithm 2: IntervalizeU: Procedure for simplifying numeric types

```
Input: An operation $\sim$, a type $\tau = \{\mathbf{num}\langle t_n \rangle \mid \varphi\}$
Output: A type $\{\mathbf{num}\langle s_j \rangle \mid s_j \in [a_j, b_j]\}$
1 $(s_i + \cdots + s_k + n,\ s_i \in [a_i, b_i] \wedge \cdots \wedge s_k \in [a_k, b_k]) \leftarrow \textsc{normalize}(\tau)$
2 $[\ell, u] \leftarrow [a_i + \cdots + a_k + n,\ b_i + \cdots + b_k + n]$
3 $a_j \leftarrow \min(\{\sim u \mid u \in [\ell, u]\})$
4 $b_j \leftarrow \max(\{\sim u \mid u \in [\ell, u]\})$
5 return $\{\mathbf{num}\langle s_j \rangle \mid s_j \in [a_j, b_j]\}$
```

the range $[-2^{63}, 2^{63} - 1]$. Rule T-Sequence allows the sequential composing of instructions. Rule T-Fun represents an invocation of a BPF helper function and checks that all arguments passed to the function match with the function's declared type. Γ is updated according to the eBPF calling convention, the return value is stored in the register `r0`, the registers `r1` through `r5` are used for passing arguments. Thus, their values are not preserved by the function call itself, and they get type Any. Rules T-Goto1 and T-Goto2 handle branch/jump instructions. Such instructions are always considered well-typed as branch conditions are instead encoded in the `assume` instruction.

For the arithmetic rules, we carefully restrict the arithmetic that pointers engage in. It is possible to subtract or add a number to a pointer, updating its annotated offset (T-PtrNum). Rule T-PtrSub allows us to subtract pointers resulting in a numerical value of the signed integer type (i.e., `ptrdiff_t` in C/C++). Rule T-Binary is used for addition and subtraction of numeric registers. Since the grammar for numeric types only allows addition and subtraction of terms (to keep types simple), we use rule T-BinAlt for all operations other than addition and subtraction. Note that operations of the form $\oplus$= are typed by applying an arithmetic rule and then T-Assign, so we do not enumerate rules for such instructions. If we have numeric types τ_1 and τ_2 and an operation $\oplus$ to apply, the idea is to convert both τ_1 and τ_2 to an equivalent representation with no slacks, that is, only intervals. We then apply the operation $\oplus$ to the computed intervals to get the result. This way, no $\oplus$ operator appears in the types. For this purpose, we define an auxiliary function IntervalizeB (Alg. 1) and construct a new resultant type of the form $\{\mathbf{num}\langle s_j \rangle \mid s_j \in [a_j, b_j]\}$, where $[a_j, b_j]$ is the computed interval. Similarly, rule T-Unary uses IntervalizeU (Alg. 2), which takes as input a single numeric type τ and a unary operation $\sim$, converts τ to an only-interval representation, and applies operation $\sim$. Both auxiliary functions make use of the normalize function, which transforms the types $\{\mathbf{num}\langle t \rangle \mid \varphi'\}$ or $\{\rho[t] \mid \varphi'\}$ into a tuple (t, φ), where φ is a weaker formula than φ' only containing the definitions of slacks while skipping any other constraints. In the term t, subtraction can be written as addition without loss of generality. We do not apply the normalize function to non-refined types.

Instruction `loadmap_fd` takes a 32-bit number n for a map and constructs a $\mathbf{map}_n$ type. In eBPF, all other interactions with maps must utilize BPF helper functions; thus, T-Map is only used to introduce maps. Rules T-AssumeNum and T-AssumePtr introduce a new relationship between two data types and store this information in the constraints of one of the input registers. Rule T-Prop is used for transferring constraints between types and is used when a register requires a constraint that refines another register. This rule can be applied non-deterministically many times; hence, it is only used in a restrictive manner, explained in Sec. 5.4.

5.2 Region Safety and Information-Flow Safety

Before the load and store operations (to follow in Sec. 5.3), we need to check whether a certain load or store operation is allowed. In this section, we introduce important terms and checks used for these operations. Our analysis is designed to verify two safety claims about load/store instructions:

(1) Region Safety: Every load or store instruction must only access data within the boundaries of a memory region.
(2) Information-Flow Safety: No load or store instruction may expose sensitive information, for example, bits of a pointer to the userspace.

To check region safety, we consider the specific memory region for the desired load/store. For the context, stack, and shared regions, we define a predicate Bounds : $\mathbb{N} \times \mathcal{T} \to \{\top, \bot\}$ that guarantees that an offset is between zero and the size of the region and is defined according to the following inequality:

$$\textsc{Bounds}(sz, \{\rho[t_p] \mid \varphi\}) \;=\; 0 \le lb(t_p,\ \varphi) \le ub(t_p,\ \varphi) + sz \le size_\rho \tag{2}$$

$$\begin{aligned} \text{where}\quad lb(t_p + t_p',\ \varphi) &= lb(t_p,\ \varphi) + lb(t_p',\ \varphi) & ub(t_p + t_p',\ \varphi) &= ub(t_p,\ \varphi) + ub(t_p',\ \varphi) \\ lb(t_p - t_p',\ \varphi) &= lb(t_p,\ \varphi) - ub(t_p',\ \varphi) & ub(t_p - t_p',\ \varphi) &= ub(t_p,\ \varphi) - lb(t_p',\ \varphi) \\ lb(s_i,\ \varphi) &= m \text{ for } \varphi \Rightarrow s_i \in [m,n], n, m \in \mathbb{Z} & ub(s_i,\ \varphi) &= n \text{ for } \varphi \Rightarrow s_i \in [m,n], n, m \in \mathbb{Z} \\ lb(n,\ \varphi) &= n & ub(n,\ \varphi) &= n, n \in \mathbb{Z} \end{aligned}$$

The check in Eq. (2) is not sufficient for determining the region safety of memory accesses to the packet region because the exact size of the packet region is not known at compile time. Note that we do not define the lb and ub operations for the offset symbols $meta$, $begin$, and end since the Bounds predicate is not intended to be applied to packet pointers. To properly check packet memory accesses for region safety, we require reasoning using constraints present in refinement types of packet pointers. Consider a packet pointer refined by a formula φ written as $\{\mathbf{pkt}[t_p] \mid \varphi\}$. To check that access of sz bytes at offset t_p is safe, the following checks are required:

$$\varphi \Rightarrow meta \le t_p \qquad \varphi \Rightarrow t_p + sz \le end$$

Recall that $meta$, $begin$, and end are offset symbols that refer to the metadata, beginning, and end of the packet region. Intuitively, the refinement φ must be strong enough to show that the lower bound of t_p is at least $meta$ and that the sum of the upper bound of t_p and sz is no greater than end. Given these two conditions, we formally state the definition for region safety:

Definition 6. *Let Γ, Δ be a type/memory environment and let ι be a load or store instruction (i.e., ι has either the form $R :=_{sz} *A$ or $*A :=_{sz} C$). We say ι is region safe if*

(1) *either $\Delta, \Gamma \vdash A : \{\rho[t_p] \mid \varphi\}$ and $\rho \ne \mathbf{pkt}$ and $\textsc{Bounds}(sz, \{\rho[t_p] \mid \varphi\})$ holds*
(2) *or $\Delta, \Gamma \vdash A : \{\mathbf{pkt}[t_p] \mid \varphi\}$ and $\varphi \Rightarrow meta \le t_p \wedge t_p + sz \le end$*

We have shown how the region safety is enforced for different regions. We now discuss how the information-flow safety is enforced. The context, packet, and shared regions might be a cause of leaking sensitive information from the program to the userspace; hence, storing pointers to these regions is disallowed. However, numeric values are allowed to be stored. As stated earlier,

$$\frac{\Delta,\Gamma \vdash A : \{\mathbf{pkt}[t_p] \mid \varphi\} \qquad \Delta,\Gamma \vdash E : \mathbf{num}\langle t_n\rangle \qquad sz \in \mathcal{S} \qquad \vdash \varphi \Rightarrow \mathit{meta} \leq t_p \qquad \vdash \varphi \Rightarrow t_p + sz \leq \mathit{end}}{\Delta,\Gamma \vdash *A :=_{sz} E \dashv \Delta,\Gamma}\ \textrm{T-P}\textsc{kt}\textrm{ST}$$

$$\frac{\Delta,\Gamma \vdash A : \mathbf{shared}_i[n] \qquad \Delta,\Gamma \vdash E : \mathbf{num}\langle t_n\rangle \qquad sz \in \mathcal{S} \qquad \textsc{Bounds}(sz, \mathbf{shared}_i[n])}{\Delta,\Gamma \vdash *A :=_{sz} E \dashv \Delta,\Gamma}\ \textrm{T-S}\textsc{hared}\textrm{ST}$$

$$\frac{\Delta,\Gamma \vdash A : \mathbf{ctx}[n] \qquad \Delta,\Gamma \vdash E : \mathbf{num}\langle t_n\rangle \qquad sz \in \mathcal{S} \qquad \textsc{Bounds}(sz, \mathbf{ctx}[n]) \qquad \textsc{Overlap}_{\mathbf{ctx}}(sz, n) = \emptyset}{\Delta,\Gamma \vdash *A :=_{sz} E \dashv \Delta,\Gamma}\ \textrm{T-C}\textsc{tx}\textrm{ST}$$

$$\frac{\Delta,\Gamma \vdash A : \mathbf{stk}[n] \qquad \Delta,\Gamma \vdash E : \tau \qquad sz \in \mathcal{S} \qquad \textsc{Bounds}(sz, \mathbf{stk}[n])}{\Delta,\Gamma \vdash *A :=_{sz} E \dashv \Delta_{\mathbf{stk}}[\tau, sz, n],\Gamma}\ \textrm{T-S}\textsc{tk}\textrm{ST}$$

$$\frac{\Delta,\Gamma \vdash A : \{\mathbf{pkt}[t_p] \mid \varphi\} \qquad sz \in \mathcal{S} \qquad \vdash \varphi \Rightarrow \mathit{meta} \leq t_p \qquad \vdash \varphi \Rightarrow t_p + sz \leq \mathit{end}}{\Delta,\Gamma \vdash R :=_{sz} * A \dashv \Delta,\Gamma[R \mapsto \mathbf{num}]}\ \textrm{T-P}\textsc{kt}\textrm{LD}$$

$$\frac{\Delta,\Gamma \vdash A : \mathbf{shared}_i[n] \qquad sz \in \mathcal{S} \qquad \textsc{Bounds}(sz, \mathbf{shared}_i[n])}{\Delta,\Gamma \vdash R :=_{sz} * A \dashv \Delta,\Gamma[R \mapsto \mathbf{num}]}\ \textrm{T-S}\textsc{hared}\textrm{LD}$$

$$\frac{\Delta,\Gamma \vdash A : \mathbf{ctx}[n] \qquad sz \in \mathcal{S} \qquad \Delta,\Gamma \models \Delta(\mathbf{ctx}, sz, n) = \tau \qquad \Delta,\Gamma \models \tau \neq \textsc{none} \qquad \textsc{Bounds}(sz, \mathbf{ctx}[n])}{\Delta,\Gamma \vdash R :=_{sz} * A \dashv \Delta,\Gamma[R \mapsto \tau]}\ \textrm{T-C}\textsc{tx}\textrm{LD}$$

$$\frac{\Delta,\Gamma \vdash A : \mathbf{stk}[n] \qquad sz \in \mathcal{S} \qquad \Delta,\Gamma \models \Delta(\mathbf{stk}, sz, n) = \tau \qquad \Delta,\Gamma \models \tau \neq \textsc{none} \qquad \textsc{Bounds}(sz, \mathbf{stk}[n])}{\Delta,\Gamma \vdash R :=_{sz} * A \dashv \Delta,\Gamma[R \mapsto \tau]}\ \textrm{T-S}\textsc{tk}\textrm{LD}$$

Fig. 6. Typing rules for load and store operations on various memory regions.

we assume that any numeric values stored in these regions are not important for the analysis, hence we never update the state of context and shared regions in Δ ($\Delta_\rho[\cdot,\cdot,\cdot]$, defined in Eq. (1), is never called for these regions). Similarly, when loading from these regions, the loaded values are assumed to be numbers. Note that an exception to this rule is context region loads when loading the pre-defined packet pointers, which is allowed when loading exact pointers (same offset and size), otherwise, the load is disallowed.

The stack region does not have the same restrictions, since the stack region is used by eBPF programs for register spilling. Therefore, loads and stores to the stack region must be handled accordingly. When storing in stack region, we use the update procedure $\Delta_{\mathbf{stk}}[\cdot,\cdot,\cdot]$ (Eq. (1)) which makes sure that overlapping cells are also removed. This is important for the information-flow safety, as the overlapping cells are assumed to be information that has gone out of scope, which may also contain pointers. If overlapping cells are not removed, it may cause bits of out-of-scope pointers to be read from stack, which is considered information leakage. Similarly, when loading from stack region, it is assumed that only cells that are present in the $C_{\mathbf{stk}}$ are loaded. Loading a value that is already not in $C_{\mathbf{stk}}$ might cause a pointer to be read, which violates information-flow safety. We defer the formal definition of information-flow safety to the subsection on load and store operations, as it requires a formalization for such operations.

5.3 Load and Store Operations

Type rules for the load and store operations on memory regions are given in figure ??. We first discuss the store instructions. Each type rule for them needs four general ingredients: a pointer A to some memory region to perform the store at, an expression E being stored, a natural number sz representing the number of bytes the data must occupy, and a condition guaranteeing safety. For Rules T-PktST, T-SharedST, and T-CtxST, E must be of type **num**. Rule T-CtxST needs to additionally check that there are no overlapping cells that contain pointer information, i.e.,

($\textsc{Overlap}_{\mathbf{ctx}}(sz, n) = \emptyset$). Also, there is no need to update the state of Δ. In T-StkST, Δ is updated using $\Delta_{\mathbf{stk}}[\cdot,\cdot,\cdot]$. The safety checks are conducted as described in Sec. 5.2.

For type rules for the load operations, we define an auxiliary "load" from a memory region.

Definition 7. *Let Δ be a memory environment, $n \in \mathbb{N}$ be an offset, and $sz \in \mathcal{S}$ be the width. Then, a* load *from a memory region $\rho \in \Delta$:*

$$\Delta(\rho, sz, n) = \begin{cases} \tau & \textit{if } \exists(\tau, sz, n) \in C_\rho \\ \textit{NONE} & \textit{else if } \rho = \mathbf{stk} \wedge \\ & \quad \textit{Overlap}_\rho(sz, n) = \emptyset \\ \mathbf{num} & \textit{else if } \textit{ValidLoad}(\rho, sz, n) \\ \textit{NONE} & \textit{otherwise} \end{cases},$$

where the ValidLoad predicate is defined as:

$$\begin{aligned} \textit{ValidLoad}(\rho, sz, n) = \forall(\tau_0, sz_0, n_0) \in \textit{Overlap}_\rho(sz, n).\\ \tau_0 <: \mathbf{num} \wedge \\ (sz_0 + n_0 < sz + n \Rightarrow \\ \exists(\tau_1, sz_1, n_1) \in \textit{Overlap}_\rho(sz, n). \\ sz_1 + n_1 + 1 = n_0) \end{aligned}$$

The intuition behind the first three cases in $\Delta(\rho, sz, n)$ is as follows:

(1) The loaded type is τ, when there exists a cell in C_ρ, which applies to $\rho \in \{\mathbf{stk}, \mathbf{ctx}\}$. This is the only way pointers are read from stack or context region.
(2) When $\rho = \mathbf{stk}$, and there is nothing stored, then none is loaded.
(3) ValidLoad checks that if there are overlapping cells, then they all must be 1) numeric types and 2) continuous with no gaps between them. If the overlapping set of cells is empty, which is the case for all shared regions, then the load is also valid.

The reason for the complexity of the load operation $\Delta(\cdot,\cdot,\cdot)$ is because we want to be able to support unaligned loads to the stack region but only as long as all of the stack's memory cells involved in the load are numeric.

Each type rule for load instructions needs three general ingredients: a pointer A to some memory region to perform the load from, a natural number sz representing the number of bytes for the data, and a condition guaranteeing safety. Memory safety checks are done similarly to stores. Rules T-PktLD and T-SharedLD always load a number. Rule T-CtxLD can load pointers (if one of the packet pointers is loaded) or numbers (otherwise).

Given the definition for load operations, we now formally define information-flow safety as:

Definition 8. *Let $\mathcal{I} = \iota_0, \ldots, \iota_n$ be a sequence of instructions, with Δ_i, Γ_i be the environments right before an instruction ι_i. We call $\mathcal{I}$ information-flow safe if:*

(1) *There is no ι_i with $0 \le i \le n$ such that ι_i has the form $*A :=_{sz} C$ such that $\Delta_i, \Gamma_i \vdash A : \{\rho'[t'_p] \mid \varphi'\}$, where $\rho' \in \{\mathbf{ctx}, \mathbf{pkt}, \mathbf{shared}_j\}$, and $\Delta_i, \Gamma_i \vdash C : \{\rho[t_p] \mid \varphi\}$*
(2) *If for some ι_i, $\Delta_i(\mathbf{stk}, sz, m) = \{\rho[t_p] \mid \varphi\}$ where $1 \le i \le n$ then there exists ι_k with $0 \le k < i$ such that ι_k has the form $*A :=_{sz} C$ with $\Delta_k, \Gamma_k \vdash C : \{\rho[t_p] \mid \varphi\}$*

The first clause states that there exists no instruction that stores a pointer in the context, shared, or packet regions. The second clause states that if at any point, a pointer can be loaded from the stack region, then there must exist some store operation that stores the same pointer into the stack.

5.4 Application of Rules

In this section, we discuss the criteria for the application of rules defined in figures 5 and ??. There are two important concerns to address. The first concern is, a register containing a certain set of constraints needed later might get overwritten at any time. Transferring the constraints to another register does not work, as we do not know with certainty which registers get rewritten without looking ahead. We resolve this issue by introducing a *ghost* register r_{ghost}, which is not used by any actual eBPF program. r_{ghost} exists to carry constraints in case we suspect a register might get overwritten later in the program. When the constraint is needed, it is transferred back to the

```
 1  r1 : pkt[begin], r2 : pkt[end], r3 : {num⟨s0⟩ | s0 ∈ [14, 18]}
 2  bb0:                                                              
 3    r4 : pkt[begin] := r1;                                          3   T-Assign
 4    r4 : {pkt[begin + s0] | s0 ∈ [14, 18]} += r3;                   4   T-PtrNum, T-Assign
 5    r4 : {pkt[begin + s0 + 4] | s0 ∈ [14, 18]} += 4;                5   T-Num, T-PtrNum, T-Assign
 6    goto bb1;                                                       6   T-Goto1

 8  bb1:
 9    assume(r4 <= r2);                                               9   T-AssumePtr
10    r4 : {pkt[begin + s0 + 4] | begin + s0 + 4 ≤ end ∧ s0 ∈ [14, 18]}
11    r_ghost : {pkt[begin] | begin + s0 + 4 ≤ end ∧ s0 ∈ [14, 18]}   11  T-Prop (r4 → r_ghost)
12    r4 : pkt[begin] := r1;                                          12  T-Assign
13    r4 : {pkt[begin + s0] | s0 ∈ [14, 18]} += r3;                   13  T-PtrNum, T-Assign
14    r4 : {pkt[begin + s0] | begin + s0 + 4 ≤ end ∧ s0 ∈ [14, 18]}   14  T-Prop (r_ghost → r4)
15    r4 : num :=4 *(r4);                                             15  T-PktLD, T-Assign
```

Fig. 7. Example of application of rules and use of register $\mathtt{r}_{ghost}$ with T-Prop.

relevant register. Rule T-Prop is used to propagate such constraints back and forth. However, the second concern is, rule T-Prop can be applied non-deterministically many times. The algorithm only uses the T-Prop rule in certain scenarios to keep inference deterministic. Firstly, after applying Rule T-AssumePtr, use T-Prop to propagate the learned constraint to $\mathtt{r}_{ghost}$. Secondly, before applying T-PktLD or T-PktST, use T-Prop to propagate the constraints from $\mathtt{r}_{ghost}$ to the register used for memory operation. For other rules, we apply them whenever they are applicable. Below, we show an example of the application of rules.

Example 5.1. Consider the code snippet of a load from packet, given in Fig. 7. The assignments `r4 := r1` (line 3 and 12) are typed using rule T-Assign, and operations like += (lines 4, 5, 13) are typed as a combination of rules T-PtrNum and T-Assign. In addition, line 5 requires typing the number 4 as $\mathbf{num}\langle 4\rangle$ using Rule T-Num. At line 9, the T-AssumePtr rule is used, as both `r4` and `r2` are packet pointers, which updates the type of `r4` with a new constraint. Notice that we apply the constraint to only `r4` but not `r2`. At lines 11 and 14, rule T-Prop has been used to propagate the constraint $begin + s_0 + 4 \le end$ back and forth from registers `r4` and $\mathtt{r}_{ghost}$. The operation at line 15 is the combination of T-PktLD and T-Assign.

After line 9, the type of `r4` contains sufficient information to prove that the memory access at line 15 is safe, that is, $begin + s_0 + 4 \le end$ (shown at line 10). However, on line 12, the register `r4` gets reassigned to `r1`, and the information gets lost. We apply the propagation rule after the `assume` operation to propagate the information present in `r4` to $\mathtt{r}_{ghost}$ (result shown at line 11). Note that we always keep $\mathtt{r}_{ghost}$ to point to $\mathbf{pkt}[begin]$ for convenience. When the required information is needed in `r4`, that is, for the load operation at line 15, we selectively propagate the constraints needed back to `r4` before the operation, and the resulting type for `r4` is shown at line 14. Using the propagation rule with $\mathtt{r}_{ghost}$ helps resolve such issues. In general, we only allow the propagation of constraints in a restricted manner to keep the algorithm deterministic.

$$\frac{}{\Delta, \Gamma \models \tau <: \textsc{any}}\ \text{S-Any} \qquad \frac{}{\Delta, \Gamma \models \textsc{none} <: \tau}\ \text{S-None} \qquad \frac{\vdash \varphi \Rightarrow \varphi'}{\Delta, \Gamma \models \{\nu : r \mid \varphi\} <: \{\nu : r \mid \varphi'\}}\ \text{S-Ref}$$

$$\frac{}{\Delta, \Gamma \models \{\nu : r \mid \varphi\} <: r}\ \text{S-NonDet} \qquad \frac{\Delta, \Gamma \models \tau_0 <: \tau_1 \quad \Delta, \Gamma \models \tau_1 <: \tau_0}{\Delta, \Gamma \models \tau_0 = \tau_1}\ \text{S-Eq}$$

Fig. 8. Rules for subtyping.

5.5 Subtyping

Subtyping is necessary for the type inference algorithm's join procedure. The rules for subtyping are given in figure 8. The first rule, S-Any, states that the type any is a supertype for all types, and Rule S-None specifies that the type none is a subtype of all types. These two rules define any and none to be the top and bottom types, respectively. Rule S-Ref states that a refined type is considered a subtype of another refined type precisely when it contains a stronger constraint. Rule S-NonDet states that any refined type is a subtype of the generic type, that is, $\{\mathbf{num}\langle t_n\rangle \mid \varphi\} <: \mathbf{num}$ and $\{\rho[t_p] \mid \varphi\} <: \rho$. Lastly, rule S-Eq specifies that two types are equal precisely when they are subtypes of each other.

We extend the concept of subtyping to environments. Given two type environments Γ_1 and Γ_2:

$$\Gamma_1 <: \Gamma_2 \Leftrightarrow \forall R \in \mathcal{V} \,.\, \Gamma_1(R) <: \Gamma_2(R)$$

where R can be any of the registers, as defined in Grammar 2.

Given two memory environments Δ_1 and Δ_2, we define $\Delta_1 <: \Delta_2$ as:

$$\Delta_1 <: \Delta_2 \Leftrightarrow \forall sz \in \mathbb{N}, n \in \mathbb{N}, \rho \in \mathfrak{R} \,.\, \Delta_1(\rho, sz, n) <: \Delta_2(\rho, sz, n)$$

5.6 Soundness

Our type system is sound but not complete. We guarantee the soundness of our type system by proving a progress and a preservation theorem using call-by-value small-step semantics. Specifically, we construct an abstraction for a program's runtime states called the runtime environment (written $\overline{\Gamma}, \overline{\Delta}$) and define a relation $\cdot \rightsquigarrow \cdot$ that relates two runtime environments if the left-hand side evaluates to the right-hand side. From there, we define a relation, $\cdot \equiv \cdot$, that relates the state of the runtime environments to the state of the type system if the type system overapproximates the runtime environment. We state the progress and preservation theorems.

Theorem 1. *(Progress) Let $\iota \in I$ be an instruction, for environments $\Delta_0, \Gamma_0, \Delta_1$ and Γ_1 if $\Delta_0, \Gamma_0 \vdash \iota \dashv \Delta_1, \Gamma_1$ and let $(\overline{\Delta}_0, \overline{\Gamma}_0)$ be a runtime environment such that $(\Delta_0, \Gamma_0) \equiv (\overline{\Delta}_0, \overline{\Gamma}_0)$ then either ι is a goto instruction or there exists an instruction $\iota' \in I$ and runtime environment $(\overline{\Delta}_1, \overline{\Gamma}_1)$ such that $(\overline{\Delta}_0,\ \overline{\Gamma}_0,\ \iota;\iota') \rightsquigarrow (\overline{\Delta}_1,\ \overline{\Gamma}_1,\ \iota')$.*

Theorem 2. *(Preservation) Let $\iota \in I$ be an instruction, $(\overline{\Delta}_0, \overline{\Gamma}_0), (\overline{\Delta}_1, \overline{\Gamma}_1)$ be runtime environments, for environments $\Delta_0, \Gamma_0, \Delta_1, \Gamma_1$; if $\Delta_0, \Gamma_0 \vdash \iota \dashv \Delta_1, \Gamma_1$ with $\Gamma_0 \equiv \overline{\Gamma}_0, \Delta_0 \equiv \overline{\Delta}_0$ and $(\overline{\Delta}_0,\ \overline{\Gamma}_0,\ \iota;\iota') \rightsquigarrow (\overline{\Delta}_1,\ \overline{\Gamma}_1,\ \iota')$ then $\Gamma_1 \equiv \overline{\Gamma}_1$ and $\Delta_1 \equiv \overline{\Delta}_1$.*

6 Type Inference And Type Checking

We have already defined the type rules in the previous section. The other two important components of our type system are type inference and type checking algorithms, which are discussed in this section.

Algorithm 3: JOIN: Procedure for joining types

```
Input: Two eBPF types $\tau_0$ and $\tau_1$
Output: An eBPF type representing the joined type (written $\tau_0 \uplus \tau_1$)
1  if $\tau_0 = \tau_1$ then
2      return $\tau_0$
3  else if $\tau_0 = \{v : r_0 \mid \varphi_0\} \land \tau_1 = \{v : r_1 \mid \varphi_1\} \land r_0 = r_1 \neq \textbf{pkt}$ then
4      $(s_0 + \cdots + s_i + s_j + \cdots + s_n + c_0,\ \ s_0 \in [a_0, b_0] \land \cdots \land s_n \in [a_n, b_n]) \longleftarrow \textsc{normalize}(\tau_0)$
5      $(s_0 + \cdots + s_i + s_\ell + \cdots + s_m + c_1,\ \ s_0 \in [a_0, b_0] \land \cdots \land s_m \in [a_m, b_m]) \longleftarrow \textsc{normalize}(\tau_1)$
6      $X \leftarrow \big\{[a, b] \mid \varphi_0 \Rightarrow s_x \in [a, b], x \in \{j, \ldots, n\}\big\} \cup \big\{[c_0, c_0]\big\}$
7      $Y \leftarrow \big\{[a, b] \mid \varphi_1 \Rightarrow s_y \in [a, b], y \in \{\ell, \ldots, m\}\big\} \cup \big\{[c_1, c_1]\big\}$
8      $[n, m] \leftarrow [\min(\sum_{[a,b]\in X} a\ ,\ \sum_{[a,b]\in Y} a),\ \max(\sum_{[a,b]\in X} b\ ,\ \sum_{[a,b]\in Y} b)]$
9      $\varphi_{new} \leftarrow s_0 \in [a_0, b_0] \land \cdots \land s_i \in [a_i, b_i] \land s_k \in [n, m]$
10     $t_{new} \leftarrow s_0 + \cdots + s_i + s_k$
11     if $r_0 = \textbf{num}$ then
12         return $\{\textbf{num}\langle t_{new}\rangle \mid \varphi_{new}\}$
13     else if $r_0 = \rho$ then
14         return $\{\rho[t_{new}] \mid \varphi_{new}\}$
15 else if
     $\tau_0 = \{\textbf{pkt}[p + t_{p_0}] \mid \varphi_{mb_0} \land \varphi_{be_0} \land \varphi_0\} \land \tau_1 = \{\textbf{pkt}[p + t_{p_1}] \mid \varphi_{mb_1} \land \varphi_{be_1} \land \varphi_1\} \land p \in \{meta, begin, end\}$
     then
16     $(p + s_0 + \cdots + s_i + s_j + \cdots + s_n + c_0,\ \ s_0 \in [a_0, b_0] \land \cdots \land s_n \in [a_n, b_n] \longleftarrow \textsc{normalize}(\tau_0)$
17     $(p + s_0 + \cdots + s_i + s_\ell + \cdots + s_m + c_1,\ \ s_0 \in [a_0, b_0] \land \cdots \land s_m \in [a_m, b_m] \longleftarrow \textsc{normalize}(\tau_1)$
18     $X \leftarrow \big\{[a, b] \mid \varphi_0 \Rightarrow s_x \in [a, b], x \in \{j, \ldots, n\}\big\} \cup \big\{[c_0, c_0]\big\}$
19     $Y \leftarrow \big\{[a, b] \mid \varphi_1 \Rightarrow s_y \in [a, b], y \in \{\ell, \ldots, m\}\big\} \cup \big\{[c_1, c_1]\big\}$
20     $[n, m] \leftarrow [\min(\sum_{[a,b]\in X} a\ ,\ \sum_{[a,b]\in Y} a),\ \max(\sum_{[a,b]\in X} b\ ,\ \sum_{[a,b]\in Y} b)]$
21     $\varphi'_{mb} \leftarrow \textsc{weaker}_{mb}(\varphi_{mb_0}, \varphi_{mb_1})$
22     $\varphi'_{be} \leftarrow \textsc{weaker}_{be}(\varphi_{be_0}, \varphi_{be_1})$
23     $\varphi_{new} \leftarrow s_0 \in [a_0, b_0] \land \cdots \land s_i \in [a_i, b_i] \land s_k \in [n, m]$
24     $t_{new} \leftarrow s_0 + \cdots + s_i + s_k$
25     return $\{\textbf{pkt}[p + t_{new}] \mid \varphi'_{mb} \land \varphi'_{be} \land \varphi_{new}\}$
26 else
27     return $\mathit{ANY}$
```

6.1 Joins

Thus far, we have only focused on type rules for individual instructions, but have not touched upon the analysis in the presence of control flow. In this section, we discuss how type information from two branches is reconciled or *joined*. A *join* operation $\uplus$ over two types τ_0 and τ_1 is defined in Alg. 3. It has multiple cases.

In the first case (line 1), $\tau_0 = \tau_1$, and the algorithm returns τ_0 without loss of generality. In the second case (line 3), types $\tau_0 = \{v : r \mid \varphi_0\}$ and $\tau_1 = \{v : r \mid \varphi_1\}$ are to be joined, and $r \neq$ **pkt**. The algorithm calls function NORMALIZE, which transforms the types into a tuple (t, φ), where φ contains the definitions of slacks while skipping any other constraints. Recall that the grammar for types only allows addition and subtraction of terms, and subtraction can be written as addition without loss of generality. For the two types to be joined, there is possibly a sum of slacks $s_0 + \cdots + s_i$ that is shared by the term in both types, while $s_j + \cdots + s_n + c_0$ and $s_\ell + \cdots + s_m + c_1$ terms contain slacks that may differ, including constants c_0 and c_1. The algorithm extracts the interval values from $s_j + \cdots + s_n + c_0$ and $s_\ell + \cdots + s_m + c_1$ by replacing slack variables with the intervals they represent and computes the join of these intervals (lines 6-8). Any slacks that are shared between the types are not considered, in the hope of keeping any important information that might be required later in the analysis. The algorithm then computes the joined term t_{new} and joined constraint φ_{new} (lines 9-10). Depending on whether the types were numeric or pointers, it computes the joined type (lines 11-14).

In the third case (line 15), the algorithm considers the join of packet pointer types. Most operations are similar to the ones in the preceding case; however, there is an extra term p representing one of the symbols *meta*, *begin*, or *end*. To compute the join, both types must contain the same symbol p. There are additional constraints φ_{mb} and φ_{be} which represent $meta+t'_p \leq begin$ and $begin+t''_p \leq end$, respectively. Since the join must keep the information common at both branches, we compute the weaker constraints. For φ_{mb} and φ_{be} constraints, we invoke the auxiliary functions $\textsc{weaker}_{mb}$ and $\textsc{weaker}_{be}$ defined as,

$$\textsc{weaker}_{mb}(\varphi_1,\varphi_2) = \begin{cases} \varphi_1 & \text{if } \varphi_0 \Rightarrow \varphi_1 \\ \varphi_0 & \text{if } \varphi_1 \Rightarrow \varphi_0 \\ meta \leq begin & \text{otherwise} \end{cases}, \quad \textsc{weaker}_{be}(\varphi_1,\varphi_2) = \begin{cases} \varphi_1 & \text{if } \varphi_0 \Rightarrow \varphi_1 \\ \varphi_0 & \text{if } \varphi_1 \Rightarrow \varphi_0 \\ begin \leq end & \text{otherwise} \end{cases}$$

These functions return the weaker of the two formulas, and if neither formula implies the other, then a generic constraint such as $meta \leq begin$ or $begin \leq end$ is returned. We extend the concept of joins to type environments and memory environments in a point-wise fashion. Given two type environments Γ_1 and Γ_2, we define $\Gamma_1 \uplus \Gamma_2$ as:

$$\Gamma_1 \uplus \Gamma_2 = \{(R, \tau_1 \uplus \tau_2) \mid (R,\tau_1) \in \Gamma_1 \wedge (R,\tau_2) \in \Gamma_2\} \tag{3}$$

Given two memory environments Δ_1 and Δ_2, we define $\Delta_1 \uplus \Delta_2$ as:

$$\begin{aligned}\Delta_1 \uplus \Delta_2 = \{(\rho, size_\rho, C_\rho) \mid (\tau, sz, n) \in C_\rho \Leftrightarrow \\ \exists(\rho, size_\rho, C_{\rho_1}) \in \Delta_1, (\tau_1, sz, n) \in C_{\rho_1}, (\rho, size_\rho, C_{\rho_2}) \in \Delta_2, (\tau_2, sz, n) \in C_{\rho_2}. \tau = \tau_1 \uplus \tau_2\}\end{aligned} \tag{4}$$

Eq. (3) states that the join over Γs is the join over common variables in both. Naturally, we extend the definition of join to a list of environments, $\uplus([\Gamma_0,\ldots,\Gamma_n])$ and $\uplus([\Delta_0,\ldots,\Delta_n])$, by multiple applications of $\uplus$.

6.2 Type Inference

Our formal analysis of eBPF programs is primarily driven by type inference. Before presenting the details of the algorithm, we assume the existence of a Control-Flow Graph $\mathcal{G} = (\mathbb{V}, \mathbb{E})$, which represents the structure of the program being analyzed. The $\mathbb{V}$ set represents the vertices (basic blocks), and the $\mathbb{E}$ set represents edges (or jumps/branches). The graph $\mathcal{G}$ may contain cycles (loops) represented using backward edges in the graph. For a deeper understanding of $\mathcal{G}$ and its properties, we refer readers to [8]. Here, we provide only an intuitive definition of the relevant concepts used in the algorithm:

- GetBasicBlocks($\mathcal{G}$)/GetInstructions($\mathcal{G}$) – Returns all basic blocks/instructions in $\mathcal{G}$.
- WeakTopologicalOrder($\mathcal{G}$) – Constructs a weak topological ordering of the basic blocks.
- GetPredecessors(bb) – Get all predecessors for a basic block bb.
- isCycleHead(bb) – Checks if a basic block bb is the *head* of a cycle (loop). When dealing with cycles, one of the basic blocks inside the cycle is labeled as the representative block of the cycle, or *head*.
- isEntry(bb) – Checks if the basic block bb is the first (entry) basic block in $\mathcal{G}$.
- UpdateCycleComponents($bb, post, \Sigma$) – Given a *head* basic block bb, conditions $post$, and proof certificates Σ, calls AnalyzeBasicBlock (to be given in Alg. 5) on all components

(basic blocks) inside a cycle), updates the *post* for all components and returns the updated *post* and Σ.

- GetLineNumber(*inst*) – Give the line number for a given instruction *inst* in $\mathcal{G}$.
- GetRegisters(*inst*)/GetCells(*inst*) – Extract registers/memory cells used in the instruction *inst*. Cells here consist of "region type/width/offset/region size" pairs $(\rho, sz, n, size_\rho)$.
- Widen(Γ, Γ_{new})/Widen(Δ, Δ_{new}) – Handling loops can lead to excessive calls to the join procedure without guaranteeing convergence. The widening procedure allows faster convergence at the cost of precision, ensuring the termination of the analysis.

The type inference algorithm, given in Alg. 4, loops through the basic blocks of an eBPF program and infers the types for all instructions. The input to the algorithm is a tuple $(\mathcal{G}, \mathcal{D})$, where $\mathcal{G}$ is a control-flow graph for the program, and $\mathcal{D}$ is a context descriptor (as defined in Sec. 4.2). The output is a set of type annotations Σ, which stores, for each instruction in the program, all relevant variables (registers and memory cells) and their types. For registers, the type annotations constitute "register/type" pairs (R, τ), and for memory cells, the annotations constitute "region type/width/offset/region size/type" tuples $(\rho, sz, n, size_\rho, \tau)$. Line 2 initializes the environments using the descriptor $\mathcal{D}$. The computation of joins requires the type information after each basic block; hence, at line 3, the algorithm constructs *post* to keep track of type environments and memory environments, and populates it throughout.

The algorithm loops through the basic blocks in a weak topological order (line 4). If the basic block *bb* is a cycle head (line 5), then initial environments are created by joining the environments of predecessors of *bb* (line 6). However, the algorithm must ensure that Γ and Δ represent all iterations of the cycle; hence, a fixed point must be reached. The loop (line 7) runs until a fixed point is reached. At each iteration (lines 8-11), the algorithm constructs new environments (Γ_{new} and Δ_{new}) by analyzing the components of the cycle one iteration at a time, starting with Γ and Δ, and algorithm checks that Γ_{new} and Δ_{new} reach a fixed point with Γ and Δ (line 12). To conclude that the fixed point is reached, the algorithm uses the subtyping relation, which induces a join semi-lattice on the environments, which has a finite height. If the fixed point is not reached, the algorithm widens Γ and Δ with Γ_{new} and Δ_{new} to include newly-learned behaviors of the cycle (line 15) to Γ and Δ.

If *bb* is not a cycle head (line 16), initial Γ and Δ are constructed, either as Γ_{init} and Δ_{init} if *bb* is the entry block (line 18) or as a join of predecessors of *bb* otherwise (line 20). The algorithm analyzes basic block *bb*, given initial Γ and Δ to get updated Γ, Δ, and certificates Σ (line 21). The type annotations Σ are returned once the algorithm finishes (line 23).

In the next two paragraphs, we describe how basic blocks are analyzed to update the environments Γ, Δ, and type annotations Σ in Alg. 5. The algorithm loops through each instruction *inst* in the basic block (line 1) and applies rules enumerated in Sec. 5 (line 2). If the rule application heuristic succeeds then we generate the new type/memory environments Γ' and Δ' as well as the set of rules r that were applied. If r is empty then none of the rules can be applied, and an error is reported to the user (line 3), along with partially constructed type annotations. The annotations constructed so far are important for the user to provide relevant information to be able to debug the program.

Whenever the algorithm successfully computes the updated Γ and Δ, it also records the type annotations for all relevant registers and memory cells used in the instruction *inst* (lines 6 and 7). Only the specific registers/memory cells accessed by the instruction will be shown to the user. On line 8, the algorithm adds the relevant types to the proof certificate Σ. On line 9, it records the set of rules that were applied to $\mathscr{R}$, which is a global-scope mapping of instructions to sets of rules. Finally, the algorithm returns the updated type/memory environments along with the proof certification Σ.

Algorithm 4: Type Inference

```
  Input: an EBPF program's control flow graph $\mathcal{G}$, context descriptor $\mathcal{D}$
  Output: Type Annotations $\Sigma \subseteq \mathcal{I} \times (\mathcal{V} \times \mathcal{T}) \times (\mathfrak{R} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathcal{T})$
1 $\Sigma \leftarrow \emptyset$
2 $\Gamma_{init}, \Delta_{init} \leftarrow init(\mathcal{D})$
3 $post \leftarrow \{bb : (\Gamma_{empty}, \Delta_{empty}) \mid bb \in \textsc{GetBasicBlocks}(\mathcal{G})\}$
4 for $bb \in \textsc{WeakTopologicalOrder}(\mathcal{G})$ do
5     if $\textsc{isCycleHead}(bb)$ then
6         $\Gamma, \Delta \leftarrow \uplus\ ([post(p) \mid p \in \textsc{GetPredecessors}(bb)])$
7         while $\top$ do
8             $\Gamma', \Delta', \Sigma \leftarrow \textsc{AnalyzeBasicBlock}(bb, \Gamma, \Delta, \Sigma)$
9             $post[bb] \leftarrow (\Gamma', \Delta')$
10            $post, \Sigma \leftarrow \textsc{UpdateCycleComponents}(bb, post, \Sigma)$
11            $\Gamma_{new}, \Delta_{new} \leftarrow \uplus\ ([post(p) \mid p \in \textsc{GetPredecessors}(bb)])$
12            if $(\Gamma_{new} <: \Gamma) \wedge (\Delta_{new} <: \Delta)$ then
13                break
14            else
15                $\Gamma, \Delta \leftarrow \textsc{Widen}(\Gamma, \Gamma_{new}), \textsc{Widen}(\Delta, \Delta_{new})$
16    else
17        if $\textsc{isEntry}(bb)$ then
18            $\Gamma, \Delta \leftarrow \Gamma_{init}, \Delta_{init}$
19        else
20            $\Gamma, \Delta \leftarrow \uplus\ ([post(p) \mid p \in \textsc{GetPredecessors}(bb)])$
21        $\Gamma, \Delta, \Sigma \leftarrow \textsc{AnalyzeBasicBlock}(bb, \Gamma, \Delta, \Sigma)$
22        $post[bb] \leftarrow (\Gamma, \Delta)$
23 return $\Sigma$
```

To sum up, the inference algorithm constructs an implicit derivation tree by applying the typing rules at each instruction. Once the types have been inferred, a *type-checking* algorithm verifies that the generated types are correct. Specifically, it verifies that the derivation tree is valid by checking that the inferred type can be constructed using operand types by application of given typing rules. If this can be checked for all instructions in the program, the checker accepts the program; otherwise rejects it. In the context of eBPF programs, the checker can reside in a secure environment and verify programs before execution.

Algorithm 5: Type annotations generation for one basic block (AnalyzeBasicBlock)

```
  Input: A basic block $bb$, type environment $\Gamma$,
         memory environment $\Delta$, and type
         annotations $\Sigma$
  Output: A tuple $(\Gamma', \Delta', \Sigma)$
1 for $inst \in bb$ do
2     $\Gamma', \Delta', r \leftarrow \textsc{ApplyRules}(inst, \Gamma, \Delta)$
3     if $r = \emptyset$ then
4         Report type error to user
5         return $(\Gamma', \Delta', \Sigma)$
6     $R \leftarrow \textsc{GetRelevantRegisters}(inst)$
7     $C \leftarrow \textsc{GetRelevantMemoryCells}(inst)$
8     $\Sigma[inst] \leftarrow \big( (R, \Gamma'(R)),\ (C, \Delta'(C)) \big)$
9     $\mathscr{R}[inst] \leftarrow r$
10 return $(\Gamma', \Delta', \Sigma)$
```

7 Implementation and Evaluation

We have implemented the type inference algorithm for our type system in the tool called VeRefine. VeRefine takes as input an eBPF program in bytecode representation and runs an inference algorithm for constructing types for each program instruction. VeRefine prints the program in the form of eBPF bytecode, with the type annotations with each instruction. If the program is considered safe, this gives the user a chance to understand why the program is safe, by reading the type annotations. When the program is considered unsafe, an error is reported to the user along with the location of the error. The user may then debug the program, using type annotations, to find the root cause.

VeRefine uses Abstract Interpretation [15], a static analysis technique, to construct *types* [14] environments to keep track of types for variables and memory regions. The core of our type inference analysis relies on a reduced product of three abstract domains [16], namely *region domain*, *packet offset domain*, and *interval domain*. The *region domain* precisely keeps track of the region to which each pointer in the analysis belongs. Since stack and context regions store pointers, to track the region for each pointer stored in stack and context cells, we also need to know the offset at which those pointers are stored. Hence, the *region domain* also requires keeping track of offsets for stack and context for precise region-tracking. For simplicity, we keep track of offsets for all regions whose offsets are known at compile time, that is, all regions except packet. The *packet offset domain* keeps track of offsets of the packet pointers and necessary packet-related constraints, by using refinement types. Finally, the *interval domain* keeps track of numeric values using refinement types and interval analysis. It is further reduced to *signed interval domain* and *unsigned interval domain*, keeping track of signed and unsigned values for program variables. Details about the interval analysis can be found in [22].

VeRefine has been implemented as part of Prevail, reusing its parser, fixpoint algorithm, and its interval domain. The implementation of VeRefine is over 9000 lines of C++ code, excluding reused code from Prevail. As stated before, we only support a type inference algorithm in VeRefine, while only a sketch of the type checking algorithm is given. A type checking algorithm can be implemented in place in VeRefine with minimal effort in the future. We also note that Prevail injects safety checks on each relevant instruction, including operand types, as *assertions* in the bytecode. VeRefine then verifies these assertions while analyzing the bytecode. If any assertions fail, an error is reported. Hence, limited support for checking has been implemented; however, a standalone checker does not exist.

7.1 Experimental Results

We have evaluated VeRefine against Prevail verifier, available at https://github.com/vbpf/prevail, commit `16e06cf`. VeRefine and Prevail both take the eBPF bytecode for the program. Prevail checks whether the program is safe or not. Optionally, it also provides a list of `pre-invariants` and `post-invariants` for each basic block in the program. VeRefine, on the other hand, checks the safety and provides type annotations for each instruction in the program.

We consider a list of 420 publicly available synthetic and industrial benchmarks in bytecode representation[1]. The benchmarks considered in the evaluation have been collected from projects including Linux, Cilium, OVS, Falco, Suricata, Prototype-Kernel, Beyla, and other publicly available repositories.

The evaluation results are given in Table 1 with Prevail treated as the baseline tool. Hence, if the output by VeRefine matches the output by Prevail, we consider it correct. Prevail solves 401 benchmarks out of 420 benchmarks, while VeRefine solves all 420. The 19 benchmarks that Pre-

[1]A publicly available subset of benchmarks comes from https://github.com/vbpf/ebpf-samples, commit `325cce1`.

Table 1. Comparison of VeRefine and Prevail. Green numbers show the number of benchmarks considered Safe, and red numbers show the number of benchmarks considered Unsafe.

	# benchmarks solved	**# benchmarks verified**	**Unique benchs verified**	**Average time**	**Maximum time**
VeRefine	420	406 (337/69)	15	0.06s	1.12s
Prevail	401	401 (332/69)	10	0.51s	14.59s

vail does not solve, but VeRefine does are benchmarks that require reading variables stored in a data section (i.e., platform variables described in Sec. 3), which is a common feature of eBPF programs nowadays. Prevail does not support handling such reads at the time of evaluation, but VeRefine does. Among the 401 benchmarks that Prevail solves (i.e., gives a correct result, whether safe or unsafe, for), VeRefine does so for 391 benchmarks. Fig. 9 gives the runtime comparison of the verifiers on these benchmarks. The 10 benchmarks that VeRefine fails to verify are false positive results, while programs are safe. Out of the unique 19 benchmarks VeRefine supports, it verifies 15 of them, all of which are verified to be safe.

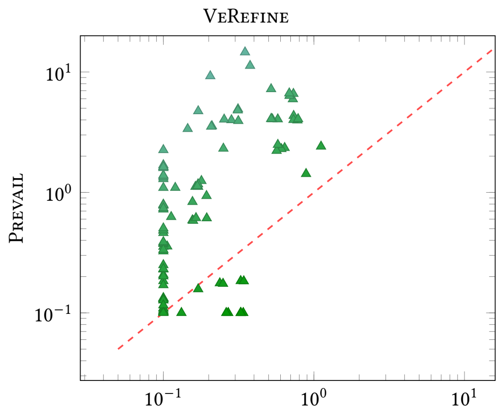

Fig. 9. Comparison of time (in sec) of VeRefine and Prevail: Each benchmark has a point representing a pair of running times. Runtimes lower than 0.1s are rounded up to 0.1s.

The remaining 4 benchmarks out of 19 are considered unsafe by VeRefine, and we report them as "unknown" as we do not have a baseline to compare them. The table also reports the number of benchmarks that are verified to be Safe/Unsafe for both tools, and the average and maximum times for both tools on the set of common 401 benchmarks that both solved. Clearly, VeRefine is an order of magnitude faster than Prevail on such benchmarks.

We must state that Prevail is a mature industry-grade tool that can conduct a more detailed analysis for eBPF programs. On the other hand, VeRefine has a restricted and simpler analysis defined by our type system. But it improves the performance and exhibits a more user-directed behavior while matching the analysis of Prevail in most cases. Our evaluation demonstrates that VeRefine inference achieves significantly better runtime performance compared to Prevail. However, the benefits of using VeRefine not only lie in the better runtime performance but also in the effectiveness of the type annotations, which we discuss in the next section.

7.2 Effectiveness of Type Annotations

We show the effectiveness of using type annotations provided by VeRefine over previous techniques that either provide no additional information when programs are rejected (for example, Linux Kernel Verifier) or provide limited information at certain program points (for example, Prevail). We showed an example in Sec. 2 explaining how the type annotations improve the debuggability of eBPF programs. This subsection provides a more detailed discussion of Prevail's output to that program called *pre-invariant* and *post-invariant*, one of which is given in Fig. 10 (for basic block bb2). In turn, Fig. 11 gives the debug information, manually extracted from such invariants but

```
Pre-Invariant: [
    r0.type=number,
    r1.svalue=s[468...471].svalue, r1.svalue=s[506...507].svalue,
    r1.type=number, r1.uvalue=s[468...471].uvalue,
    r1.uvalue=s[506...507].uvalue,
    r10.stack_offset=512, r10.svalue=[512, 2147418112], r10.type=stack,
    r6.map_fd=1, r6.svalue=[1, 2147418112], r6.svalue=r6.uvalue,
    r6.type=map_fd, r6.uvalue=[1, 2147418112],
    s[468...471].svalue=s[506...507].svalue, s[468...471].type=number,
    s[468...471].uvalue=s[506...507].uvalue, s[506...507].type=number
]
Stack: Numbers -> {[468...495], [506...507]}
```

Fig. 10. PREVAIL's pre-invariant for basic block bb2 for example in Fig. 1.

```
r1 : stk[506], r2 : num⟨2⟩, r3 : num,
r6 : map_1, r10 : stk[512]
bb0:
  r0 = probe_read(r1, r2, r3);
  goto bb1;
r0 : num, r6 : map_1, r10 : stk[512],
stack[506-507] : num

r0 : num, r6 : map_1, r10 : stk[512],
stack[506-507] : num
bb1:
  r1 = *(u16 *)(r10 - 6);
  *(u32 *)(r10 - 44) = r1;
  goto bb2;
r0 : num, r1 : num, r6 : map_1,
r10 : stk[512], stack[506-507] : num,
stack[468-471] : num
```

```
r0 : num, r1 : num, r6 : map_1,
r10 : stk[512], stack[506-507] : num,
stack[468-471] : num
bb2:
  r1 = *(u32 *)(r10 - 44);
  *(u32 *)(r10 - 4) = r1;
  r2 = r10;
  r2 += -4;
  r1 = r6;
  r0 = map_lookup_elem(r1, r2);
  goto EXIT;
r0 : num, r6 : map_1, r10 : stk[512],
stack[506-507] : num,
stack[468-471] : num,
stack[508-511] : num
```

Fig. 11. Example bytecode with (simplified) PREVAIL invariant from Fig. 1.

provided only at the beginning or end of a basic block. We use the same notation as the types in our type system to provide concise information.

This also shows that the issue is not with the verbosity of the debug information (PREVAIL invariants are hefty), as we could convert PREVAIL invariants to types and the debuggability is still not improved[2]. However, since PREVAIL does not use slack variables, we omit them from the types and provide generic types.

7.2.1 Debuggability. The error reported for `map_lookup_elem` is attributed to two possible scenarios: (1) the type of map in `r1` is wrong or map has not been constructed correctly, (2) the value at the location pointed by `r2` is not correct/sufficient. To debug the first scenario, the user might need to trace back till the pre-invariant of `bb2` to find the type of `r6` and hence `r1`, and then further

[2]Recently, there has been an effort to make PREVAIL invariants simpler: https://github.com/vbpf/ebpf-verifier/pull/798.

debug whether the map is constructed correctly. To debug the second scenario, the user might need to first find where `r2` points to, which again requires tracing back to the pre-invariant. Once the pointer is computed, the user has to figure out what is stored at location `stack[508-511]`, which seems easier to find out from the post-invariant; however, generally, there might be many instructions between the error location and the end of the block. Debugging in case of Prevail requires multiple scans of the basic block. This effort gets amplified as finding the root cause of the error might involve multiple blocks. Type annotations provided by VeRefine require fewer look-ups, as shown in Sec. 2.

7.2.2 Expressiveness of Type Annotations. We argue that the type annotations are expressive enough to help understand the errors in the program for the user. When type annotations are provided, the user does not need to understand the underlying type system to figure out any details. However, the absence of the type annotations might demand a deeper understanding. Assume that the user tries to understand line 4 in Fig. 11. The user reads about `probe_read` to know that it stores the contents read to the location `stack[506-507]`. However, to understand what is being stored at that location requires knowledge of the underlying type system. Prevail stores a number with no known value when the analysis cannot infer an exact value. However, other analyses may handle this differently. As a result, the user must understand the specifics of the analysis being performed, which can be challenging. The type annotations are expressive enough to tell the user the required information without requiring any underlying details.

8 Related Work

The most closely related eBPF verifier, Prevail [24], is based on the Zone abstract domain [36], which results in an expensive analysis as it tracks all pairwise variable relations. Instead, VeRefine uses refinement types to only track relations that are needed to verify safety, which in most cases are operations over packet pointers, e.g., pointer arithmetic and `assume` operations. Since eBPF variables are mutable, the type inference algorithm provided by VeRefine is flow-sensitive. Furthermore, mutability might invalidate relations among program variables, which is addressed in VeRefine with immutable slack variables. Comparing with Prevail, our choice of a more specialized abstraction allows faster verification while providing sufficient ability to verify for memory safety with a few more false positives. A work-in-progress verifier for eBPF programs, Exoverifier [4], is based on both abstract interpretation and symbolic execution and supports the generation and checking of proofs for memory safety. Symbolic execution is generally known to suffer from the path explosion problem. With both settings, the tool is much slower in proof generation and proof checking, and reports timeouts and errors. Linux verifier also explores all possible program executions, and is known to report false positive results. VeRefine avoids exploring all paths by reconciling information at branch join points. The analysis based on the flow-sensitive refinement type system is sufficient to avoid the pitfalls of the path explosion problem and avoids the complex analysis by only keeping the required information. This results in simpler analysis and better performance than state-of-the-art verifier Prevail. Another tool called `bpfverify` [9] considers bit- and memory-precise verification of functional properties in eBPF programs, specified as pre-conditions and post-conditions. The eBPF programs are modeled as Constrained Horn Clauses (CHCs), which enables the use of external solvers such as [31]. `bpfverify` has only limited support for kernel-helper functions, and does not support loops.

The work in [26] introduces flow-sensitive typing to scripting languages, which constructs static typing of variables using runtime tags for JavaScript programs, although with limited support, like typing for only non-recursive data types, intra-procedural analysis, and considering only local effects. Many following works explore flow-typing in JavaScript, including flow [11] and

TypeScript [50], which attempt to improve the limitations in [26], such as providing a type inference procedure and more non-local reasoning. VeRefine differs from these approaches by targeting a low-level language; thus VeRefine supports low-level constructs such as pointers. The work in [40] provides an algorithm for sound and complete flow-typing in the presence of unions, intersections, and negations, and the work in [39] introduces a calculus for constraint-based flow-typing. The work in [30] provides a framework for semantically sound flow-sensitive type systems, in the context of secure information-flow in While programs. Inspired by previous work in flow-sensitivity, the work in [41] presents a flow-sensitive type system for a statically-typed While language, giving the language the feel of a dynamically-typed language. However, each of these works consider languages with while-loops unlike VeRefine. The flow-sensitive analysis has also been applied to the functional programming paradigm in [35].

Refinement types were first introduced for ML [21]; however, this type system does not support imperative constructs. Liquid types are similar to refinement types, which were introduced in [43], and the paradigm was extended to low-level languages in [44]. SolType [48] uses a similar framework for refinement types as VeRefine, which utilizes a refinement type system to verify the safety of arithmetic operations in Solidity smart contracts. SolType focuses on arithmetic properties of contracts, considers any data containers immutable, and provides a flow-insensitive analysis. VeRefine supports reasoning about the contents of memory regions to check for information-flow safety, which requires flow-sensitive reasoning. Refinement types have also been used for verifying security properties of cryptographic protocols and access control mechanisms [7] and for TypeScript [18, 51]. The work in [20] combines flow-sensitive analysis with refinement typing, but restricts pointer aliasing to ease their analysis of strong updates. The work in [38] implements a data flow refinement type inference algorithm for functional programs. Later approaches that combine flow-sensitivity and refinement types include ConSORT [49] and [33]. ConSORT applies these concepts in imperative languages where mutability and aliasing are present. In [33], types are refined using *security labels*, which are then used in the information-flow analysis. Unlike VeRefine, all these type systems analyze high-level languages. VeRefine uses a construction we call *slack* variables, which is a standard technique in numeric optimization [12] used to transform an inequality constraint into an equality constraint. However, in our type system, these are used differently and serve two purposes: 1) to encode interval/range types, and 2) to keep track of packet pointer offsets. The latter usage is closest to the traditional usage of slack variables. Other type systems [10] use a concept similar to ours, albeit for a very different purpose, namely, variables in this type system encode discrete constraints over type constructors rather than continuous numeric constraints. Compared to all of these works, VeRefine does not develop a novel theory in the field of refinement type systems; instead, it applies well-known techniques to a novel domain, namely, eBPF verification.

Several type systems target low-level languages such as TALx86 [17], and more advanced type systems have been developed for low-level languages such as LTLL [44], DTAL [52], and SPush [46]. The type system developed in [23] targets WebAssembly and uses the concept of indexed types to replace dynamic runtime checks with static checks, thus improving the performance. Unlike the aforementioned type systems, our type system provides types for pointers annotated with a location in memory. The work in [32] uses flow-sensitive points-to analysis as a reachability problem for value flow graphs. Value flow analysis reasons about what values the variables hold at any point. Some other works, [28, 29, 53], use *def-use chains* in the flow-sensitivity setting to restrict the unnecessary propagation of pointer information. For programs containing pointers in general (e.g., C programs), such analysis can be expensive. VeRefine uses type inference rules for such information propagation and uses simpler *def-use* rules since eBPF uses well-defined memory regions with restricted use of pointer operations, which contributes to efficient but also sufficient implementation for checking memory safety.

Many works target memory safety for programs in languages other than eBPF. Checked C [19, 45] is a framework for checking memory safety (specifically spatial safety, i.e., memory accesses are within allowed bounds) of C programs. It enables backward compatibility and incremental conversion with legacy C. It does not use refinement types. The Cyclone tool [25] verifies C programs for both spatial safety and temporal safety (i.e., pointers are not accessed after freeing) using region-based memory management. It requires the user to explicitly write region annotations (which region each pointer belongs to); however, this effort is reduced with the help of default annotations and local type inference. VeRefine infers sufficient annotations to check memory safety automatically. Unlike Checked C and Cyclone, but similar to VeRefine, Deputy [13] uses dependent types to incorporate bounds information to verify spatial safety. However, their type system is flow-insensitive and requires adding runtime checks in the presence of flow-sensitivity. However, a recent work [47] constructs a fully static dependent type system for verifying spatial memory safety for low-level C programs.

9 Conclusion and Future Work

eBPF is an emerging technology that relies on the verification of programs to prove safety, before programs can run inside a secure Linux or Windows kernel environment. Previous verifiers, including the Linux eBPF Verifier and Prevail verifier (for Windows), have provided opportunities to successfully verify eBPF programs, however, such verifiers cannot provide certificates about failing or passing eBPF programs through the verifier, and help programmers understand the root cause of verification failure. We eliminate such a lack of user-guidedness by providing type annotations for eBPF programs being verified. We provide a robust tool based on a type inference algorithm, called VeRefine, that generates such annotations, giving programmers a chance to understand the reasoning behind the cause of error, which supports better debugging for the user. We successfully evaluated VeRefine on 420 benchmarks, showing that it outperforms Prevail in most of the benchmarks. We also provided a subjective study based on the benefits of using type annotations for the debugging process, as compared to verification logs by Prevail.

For future work, we plan to support *user annotations*. User annotations are similar to type annotations and allow users to specify properties about variables in the program at a given program location. When an error is reported and the user identifies it as a false positive result, the user may specify an annotation to guide the verifier to accept the program. To our knowledge, currently, eBPF programmers go through a painful trial-and-error process to get a program verified, by changing the high-level source code. User annotations further bridge the gap between the reporting of an error and helping users fix that error. VeRefine currently only supports memory safety properties, similar to Prevail verifier. In the medium term, we plan to target eBPF for Windows that uses Prevail as its verifier. Hence, the support for only memory safety properties is sufficient. In the long term, we plan to support other properties, like termination and deadlock freedom for eBPF programs. Currently, our support for verification of safety properties is on par with Prevail but lies behind the Linux Verifier. As our work is motivated by the PCC infrastructure, a natural extension of the work is to support a proof-checking algorithm for the types/proofs that resides inside the secure environment of the Kernel.

Artifact Available. A virtual machine is available to reproduce the reported results [27].

Acknowledgements. This work was partially supported by the National Science Foundation grants 1816936 and 2106949.

References

[1] [n. d.]. https://docs.kernel.org/bpf/verifier.html.

[2] [n. d.]. https://www.kernel.org/doc/html/v5.17/bpf/instruction-set.html.

[3] [n. d.]. https://www.ietf.org/archive/id/draft-ietf-bpf-isa-04.html#section-5.4.

[4] [n. d.]. https://github.com/uw-unsat/exoverifier/.

[5] John Phineas Banning. 1978. *A method for determining the side effects of procedure calls.* Ph. D. Dissertation. Stanford, CA, USA. AAI7905815.

[6] John P. Banning. 1979. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. Association for Computing Machinery, New York, NY, USA, 29–41. doi:10.1145/567752.567756

[7] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 8 (Feb. 2011), 45 pages. doi:10.1145/1890028.1890031

[8] François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings (Lecture Notes in Computer Science, Vol. 735)*, Dines Bjørner, Manfred Broy, and Igor V. Pottosin (Eds.). Springer, 128–141. doi:10.1007/BFB0039704

[9] Martin Bromberger, Simon Schwarz, and Christoph Weidenbach. 2024. Automatic Bit- and Memory-Precise Verification of eBPF Code. In *Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing, Vol. 100)*, Nikolaj Bjørner, Marijn Heule, and Andrei Voronkov (Eds.). EasyChair, 198–221. doi:10.29007/sj4l

[10] Robert Cartwright and Mike Fagan. 1991. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 278–292. doi:10.1145/113445.113469

[11] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. 2017. Fast and precise type checking for JavaScript. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 48 (Oct. 2017), 30 pages. doi:10.1145/3133872

[12] Vašek Chvátal. 1983. *Linear Programming.* W.H. Freeman. 14–15 pages. Print.

[13] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. 2007. Dependent Types for Low-Level Programming. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4421)*, Rocco De Nicola (Ed.). Springer, 520–535. doi:10.1007/978-3-540-71316-6_35

[14] Patrick Cousot. 1997. Types as Abstract Interpretations. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 316–331. doi:10.1145/263699.263744

[15] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. doi:10.1145/512950.512973

[16] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen (Eds.). ACM Press, 269–282. doi:10.1145/567752.567778

[17] K Crary, Neal Glew, Dan Grossman, Richard Samuels, F Smith, D Walker, S Weirich, and S Zdancewic. 1999. TALx86: A realistic typed assembly language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*. 25–35.

[18] Ivo Gabe de Wolff and Jurriaan Hage. 2017. Refining types using type guards in TypeScript. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Paris, France) *(PEPM 2017)*. Association for Computing Machinery, New York, NY, USA, 111–122. doi:10.1145/3018882.3018887

[19] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018.* IEEE Computer Society, 53–60. doi:10.1109/SECDEV.2018.00015

[20] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 1–12. doi:10.1145/512529.512531

[21] Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 268–277. doi:10.1145/113445.113468

[22] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2014. Interval Analysis and Machine Arithmetic: Why Signedness Ignorance Is Bliss. *ACM Trans. Program. Lang. Syst.* 37, 1 (2014), 1:1–1:35. doi:10.1145/2651360

[23] Adam T. Geller, Justin Frank, and William J. Bowman. 2024. Indexed Types for a Statically Safe WebAssembly. *Proc. ACM Program. Lang.* 8, POPL (2024), 2395–2424. doi:10.1145/3632922

[24] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1069–1084. doi:10.1145/3314221.3314590

[25] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 282–293. doi:10.1145/512529.512563

[26] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2011. Typing Local Control and State Using Flow Analysis. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6602)*, Gilles Barthe (Ed.). Springer, 256–275. doi:10.1007/978-3-642-19718-5_14

[27] Ameer Hamza, Lucas Zavalia, Arie Gurfinkel, Jorge A. Navas, and Grigory Fedyukovich. 2025. Artifact for the OOPSLA'25 paper: A Flow-Sensitive Refinement Type System for Verifying eBPF Programs . doi:10.5281/zenodo.15760800

[28] Ben Hardekopf. 2009. *Pointer Analysis: Building a Foundation for Effective Program Analysis.* PhD thesis. University of Texas at Austin, Austin, TX, USA.

[29] Ben Hardekopf and Calvin Lin. 2009. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 226–238. doi:10.1145/1480881.1480911

[30] Sebastian Hunt and David Sands. 2006. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '06)*. Association for Computing Machinery, New York, NY, USA, 79–90. doi:10.1145/1111037.1111045

[31] Hari Govind V. K., Grigory Fedyukovich, and Arie Gurfinkel. 2020. Word Level Property Directed Reachability. In *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020.* IEEE, 107:1–107:9. doi:10.1145/3400302.3415708

[32] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2011. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 343–353. doi:10.1145/2025113.2025160

[33] Peixuan Li and Danfeng Zhang. 2017. Towards a Flow- and Path-Sensitive Information Flow Analysis. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017.* IEEE Computer Society, 53–67. doi:10.1109/CSF.2017.17

[34] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993.* USENIX Association, 259–270. https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet

[35] Jan Midtgaard. 2012. Control-flow analysis of functional programs. *ACM Comput. Surv.* 44, 3, Article 10 (June 2012), 33 pages. doi:10.1145/2187671.2187672

[36] Antoine Miné. 2001. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2053)*, Olivier Danvy and Andrzej Filinski (Eds.). Springer, 155–172. doi:10.1007/3-540-44978-7_10

[37] George C. Necula. 1997. Proof-Carrying Code. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 106–119. doi:10.1145/263699.263712

[38] Zvonimir Pavlinovic, Yusen Su, and Thomas Wies. 2021. Data flow refinement type inference. *Proc. ACM Program. Lang.* 5, POPL, Article 19 (Jan. 2021), 31 pages. doi:10.1145/3434300

[39] David J. Pearce. 2013. A calculus for constraint-based flow typing. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs, FTfJP 2013, Montpellier, France, July 1, 2013*, Werner Dietl (Ed.). ACM, 7:1–7:7. doi:10.1145/2489804.2489810

[40] David J. Pearce. 2013. Sound and Complete Flow Typing with Unions, Intersections and Negations. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7737)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, 335–354. doi:10.1007/978-3-642-35873-9_21

[41] David J Pearce and James Noble. 2011. Structural and Flow-Sensitive Types for Whiley. (2011).

[42] Liz Rice. 2023. *Learning eBPF.* O'Reilly Media.

[43] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. doi:10.1145/1375581.1375602

[44] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2010. Low-level liquid types. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 131–144. doi:10.1145/1706299.1706316

[45] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. 2019. Achieving Safety Incrementally with Checked C. In *Principles of Security and Trust - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11426)*, Flemming Nielson and David Sands (Eds.). Springer, 76–98. doi:10.1007/978-3-030-17138-4_4

[46] Ando Saabas and Tarmo Uustalu. 2006. Compositional Type Systems for Stack-Based Low-Level Languages. In *Theory of Computing 2006, Proceedings of the Twelfth Computing: The Australasian Theory Symposium (CATS2006). Hobart, Tasmania, Australia, 16-19 January 2006, Proceedings (CRPIT, Vol. 51)*, Joachim Gudmundsson and C. Barry Jay (Eds.). Australian Computer Society, 27–39. http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV51Saabas.html

[47] Julien Simonnet, Matthieu Lemerre, and Mihaela Sighireanu. 2024. A Dependent Nominal Physical Type System for Static Analysis of Memory in Low Level Code. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 272 (Oct. 2024), 30 pages. doi:10.1145/3689712

[48] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. 2022. SolType: refinement types for arithmetic overflow in solidity. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–29. doi:10.1145/3498665

[49] John Toman, Ren Siqi, Kohei Suenaga, Atsushi Igarashi, and Naoki Kobayashi. 2020. ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 684–714. doi:10.1007/978-3-030-44914-8_25

[50] Dan Vanderkam. 2024. *Effective TypeScript: 83 Specific Ways to Improve Your TypeScript* (2 ed.). O'Reilly Media.

[51] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 310–325. doi:10.1145/2908080.2908110

[52] Hongwei Xi and Robert Harper. 2001. A dependently typed assembly language. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming* (Florence, Italy) *(ICFP '01)*. Association for Computing Machinery, New York, NY, USA, 169–180. doi:10.1145/507635.507657

[53] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 218–229. doi:10.1145/1772954.1772985

Received 2025-03-25; accepted 2025-08-12