Honey Potion: an eBPF Backend for Elixir (Tool Paper)

Kael Soares Augusto
UFMG
Brazil
kaelaugusto@dcc.ufmg.br

Vinícius Pacheco
vinicius-sp@hotmail.com
Cadence
Brazil

Marcos Augusto Vieira
mmvieira@dcc.ufmg.br
UFMG
Brazil

Rodrigo Geraldo Ribeiro
rodrigo.ribeiro@ufop.edu.br
UFOP
Brazil

Fernando M. Quintão Pereira
fernando@dcc.ufmg.br
UFMG
Brazil

Abstract

The Extended Berkeley Packet Filter (eBPF) is a sandboxed virtual machine that runs on operating systems with kernel privileges. Currently, eBPF programs are either translated from a subset of C, called Restricted C, or from bindings available for languages such as Rust or Python. This paper describes Honey Potion, a compiler that compiles Elixir to eBPF binaries. Translation is challenging, for it must not only preserve semantics, but also satisfy the eBPF verifier, which requires proofs of in-bounds memory accesses and termination. The translator relies heavily on this last constraint—ensured termination—to implement different optimizations: constant propagation, type specialization and partial evaluation. Honey Potion is publicly available, and has been used in the development of many eBPF applications, such as packet routers, process monitors and event loggers. To the best of our knowledge, Honey Potion is the first translator of a functional programming language to eBPF.

CCS Concepts: • **Software and its engineering → Compilers**; *Software libraries and repositories.*

Keywords: Code Generation, eBPF, Partial Evaluation

1 Introduction

The *Extended Berkeley Packet Filter* (eBPF) is a virtual machine designed to run with high privileges within the kernel of operating systems [34]. The eBPF programs work in tandem with a *formal verifier*, which ensures that these programs meet constraints such as termination and absence of out-of-bounds memory accesses. The eBPF virtual machine lets users augment the set of functionalities available to the operating system's kernel. EBPF programs are used, for instance, to filter or reroute network packets, log kernel events, perform load balancing, detect DDoS attacks, etc [28]. Being one of the few industrial programming environments where formal verification via static analysis happens by default, eBPF elicited much interest among programming language researchers in recent years [13, 15, 18, 22, 35, 36].

A Challenging Compilation Target. Typical eBPF applications consist of two parts: a frontend, and a backend [28] (an *event handler*). Users interact mostly with the frontend. Interactions with the backend, in turn, are minimal: the handler runs within kernel space. Handlers are not Turing Complete, for the verifier requires provable termination. Frontend applications are implemented in many different programming languages: C, C++, Rust, Python, Go, etc. Backend code, on the other hand, is normally written in *Restricted C*, a subset of the C programming language. Restricted C is not defined by an ISO standard; rather, it is defined by what an *eBPF Verifier* accepts—a component that is also not formally defined [13, 35].[1] The Linux eBPF verifier[2] ensures that every memory access happens in-bounds [35, 36], and that every program terminates. Since LLVM 3.7.0, `clang` can produce eBPF code out of C programs. If this code passes the sieve of the eBPF verifier, then its source is deemed to be Restricted C. To the best of our knowledge, no programming language can be directly compiled into eBPF code that is valid (meaning "verifiable") by construction. Challenges in this case stem not only from the restrictions imposed by the verifier, but also from the scarcity of resources available to run eBPF backends: programs loaded in the kernel, for instance, can use only 512 bytes to implement a function's stack.

The Contributions of this Paper. This paper describes Honey Potion, a compiler that generates eBPF programs out of Elixir code. The design and implementation of Honey Potion is particular in a number of ways. First, following recent methodology, Honey Potion is modeled after what Marr and Ducasse [20] call a *meta-compiler*. In this regard, Honey Potion is built on top of the default Elixir interpreter: it generates code by traversing the abstract syntax tree that this interpreter uses, while propagating the values that are statically known. In contrast to Marr and Ducasse's main use case, Honey Potion is not a just-in-time compiler: it works ahead-of-time. However, the modus operandi of a partial evaluator is still in place, as observed by Latifi [16]: the need to ensure terminating behavior provides Honey Potion with

[1] An effort towards a standardization of the eBPF specification is under way; however, to this date, such standard does not exist yet. See https://lpc.events/event/16/contributions/1355/

[2] In our experience, the source code of the Linux Verifier (at https://github.com/torvalds/linux/blob/master/kernel/bpf/verifier.c) works as the *de facto* specification of Restricted C within the eBPF community.

much static information, including the maximum depth of the recursion stack. As a consequence, functions can be fully inlined by the code generator, as if calls were completely interpreted.[3] This approach finds its inspiration in the design of the Truffle AST interpreter [38], whose implementation is centered around Futamura's First Projection: compiled code is derived from the interpreter via partial evaluation.

Ensured termination opens opportunities for many optimizations. These opportunities come from the fact that Honey Potion operates on program representations (introduced in Section 3.1) whose call graphs are trees, and whose control-flow graphs are acyclic (as explained in Section 3.2). Constants and types are fully propagated and many conditionals are resolved, as discussed in Section 3.3. Concerning the propagation of types, Section 3.4 explains how Honey Potion effectively adds *gradual types* [8, 25, 29] to Elixir. Every eBPF backend handler is a function whose type signature is statically known. Ill-typed programs are rejected at compile time. Honey Potion users still see Elixir as a dynamically-typed language; indeed, the handler manipulates dynamically typed values. However, types of values declared in the handler's signature are propagated by the partial-evaluator; thus, several operations can be implemented using eBPF primitive types, instead of boxed values.

Summary of Results. Honey Potion is available under the GPL 3.0 license. Section 5 evaluates this implementation. EBPF code written in Elixir is typically 2.88x shorter than equivalent C programs (Sec. 5.1). However, binaries produced by Honey Potion are about 2.36x larger than equivalent C codes (Sec. 5.2). Yet, the difference in performance between codes produced via Honey Potion and handwritten C programs is small: these programs can be thought of as exception handlers whose execution demands a small number of machine instructions. Thus, the number of instructions fetched during the execution of Honey Potion programs is less than 1.1x larger than the number of instructions fetched during the execution of their C counterparts (sec. 5.3). Optimizations play a role in the quality of the running code: on average, they reduce by 1.45x the size of the abstract syntax tree of programs (Sec. 5.4). Finally, we show that the compilation time of Honey Potion is practical, despite the fact that it relies on the partial evaluation of Elixir programs: Section 5.5 shows that the time taken to obtain an eBPF program out of a Honey Potion script is about twice the time to obtain an eBPF program out of a handwritten C file. Yet, we emphasize that the former pipeline involves two phases: a first step translates Elixir to C using the Beam virtual machine, and a second step translates C to eBPF using the `clang` compiler.

[3]The connection between partial evaluation and function inlining is well-known in the literature. For instance, in their effort to formalize function inlining, Monnier and Shao [23] introduce two binding-time annotations: *static* and *dynamic*. In their words, "*a static function call is executed at compile time thus is always inlined, while a dynamic call is executed at run time thus is not inlined.*". In practice, Honey Potion compiles a subset of Elixir where every function would be marked with the *static* qualifier.

2 Honey Potion in Examples

A Honey Potion program has four parts, which we refer to as "sections". These sections are: the include zone, the map creation area, the trigger declaration and the main function. The include specifies module dependencies and defines the license that the program uses. The map creation area lets users create maps—data structures to transfer information between eBPF and the user space. The trigger declaration defines which event is responsible for triggering the execution of the program. Finally, the main function contains the actual program that runs when the trigger fires off.

Example 2.1. The program in Figure 1 checks for processes killed with the -9 flag. Whenever such an event happens, it prints logging information through a terminal's standard output. The include section in Line 2 indicates that this program uses Honey Potion as a dependency. The map declaration section creates the `ForceKills` map that has the structure of a hashmap with 64 entries. Updates in this structure will be printed in the standard output, as indicated in Lines 3 to 6. In this regard, notice that the map defined at Line 3 is more than a data structure: in Honey Potion, the only way to pass data between userland and kernelland is via Elixir maps. The trigger of this program is "sys_enter_kill", declared in Line 7. This is a default eBPF system call that runs every time that a process is killed. Finally, Lines 8 to 15 define the main function. This function gathers the identifier of the killed process and the argument used to kill it (Lines 9 and 10). It then verifies if the argument used was 9 (Lines 12 and 13). If true, the program sets the process ID as one of the keys in the map with value 1, which will make it occupy one of the 64 positions available in the map.

```
01 defmodule Forcekill do
02   use Honey, license: "Dual BSD/GPL"
03   defmap(
04     :ForceKills,
05     %{type: BPF_MAP_TYPE_HASH, max_entries: 64, print: true}
06   )
07   @sec "tracepoint/syscalls/sys_enter_kill"
08   def main(ctx) do
09     sig = ctx.sig
10     pid = ctx.pid
11     cond do
12       sig == 9 -> Honey.Bpf_helpers.bpf_map_update_elem(
13           :ForceKills, pid, 1, :BPF_NOEXIST)
14     end
15   end
16 end
```

Figure 1. Program that logs "force-killed" processes.

Memory allocation. EBPF programs contain much static information: constant literals that bound loops or fix the size of buffers. This pattern emerges naturally due to the need to satisfy the eBPF verifier. Honey Potion capitalizes heavily on these constants, to solve expressions at compilation time,

as Section 3.3 explains, or to pre-allocate memory buffers, as Section 4.1 discusses. Honey Potion does provide a heap and a stack, although the latter is very constrained: 512 bytes only. Thus, built-in data structures—strings, tuples and lists—are allocated in the heap, which is called "Honey Stack": a chunk of 2 kB statically allocated. Additionally, Honey Potion provides to the users eBPF maps, a data structure built-in in the eBPF environment, which Example 2.2 discusses.

```
01 defmodule CountSysCalls do
02   use Honey, license: "Dual BSD/GPL"
03
04   printlist = [{"Syscall: enter_read (0) | Qtt:", 0},
05                 {"Syscall: enter_write (1) | Qtt:", 1},
06                 {"SysCall: enter_kill (62) | Qtt:", 62},
07                 {"SysCall: enter_mkdir (83) | Qtt:", 83},
08                 {"SysCall: enter_getrandom (318) | Qtt:", 318}]
09
10   defmap(:Count_Sys_Calls_Invoked,
11     %{type: BPF_MAP_TYPE_ARRAY, max_entries: 335, print: true,
12       print_elem: printlist})
13
14   @sec "tracepoint/raw_syscalls/sys_enter"
15   def main(ctx) do
16     id = ctx.id
17     id_count = Honey.Bpf_helpers.bpf_map_lookup_elem
18       (:Count_Sys_Calls_Invoked, id)
19     Honey.Bpf_helpers.bpf_map_update_elem
20       (:Count_Sys_Calls_Invoked, id, id_count + 1)
21   end
22 end
```

Figure 2. Program that counts occurrences of five system calls. The ID of each syscall (0, 1, 62, etc) is OS specific.

Example 2.2. The program in Figure 2 logs occurrences of five system calls. The map section creates the variable `printlist`, whose entries will be printed in the standard output. This section also creates the map `Count_Sys_Calls_Invoked` with 335 entries. This number accommodates the most common system calls; however, only entries in `printlist` are logged. These data structures are allocated as eBPF maps. The `main` function grabs the ID of the system call (Line 16), checks the map to see how often it has been observed (Line 17), and updates the count in the map (Line 19).

Recursion. Honey Potion supports recursive programs. However, as Section 3.2 will explain, recursive calls must be statically bounded with an extra numeric qualifier: its "fuel". The fuel must be appended to every recursive call. This so-called "Petrol Semantics" [21] makes Honey Potion Turing Incomplete; however, eBPF is also Turing Incomplete. Example 2.3 shows how recursive calls can be performed.

Example 2.3. Figure 3 contains two examples of recursive function calls qualified with the extra "fuel" parameter. The invocation at Line 13 has a maximum recursive depth of two, meaning that at most two instances of its activation record can be stacked up at any time. The invocation at Line 14 is even more restricted: it allows only one activation of `fact`. Due to the section declaration at Line 09, the `main` function is called for every `write` system call, and prints a factorial at that point in the logging file descriptor.

```
01 def fact(x) do
02   if x == 1 do
03     1
04   else
05     x*fact(x-1)
06   end
07 end
08
09 @sec "tracepoint/raw_syscalls/sys_enter"
10 def main(ctx) do
11   const = 3
12   id = ctx.pid
13   a = fuel 3, fact(const)
14   b = fuel 2, fact(id)
15   Honey.Bpf_helpers.bpf_printk(["%d",a+b])
16 end
```

Figure 3. Example of bounded recursive call.

Data Structures. Honey Potion supports the main built in data structures of Elixir, namely: lists, tuples, maps, strings, charlists and binaries. Notice that memory allocation is dynamic: the actual address of data within the honey stack depends on the path taken by the program's control flow. However, the maximum amount of memory necessary to run a program is statically known, because the number of possible execution paths in a Honey Potion program is finite. Honey Potion defines a few built in data structures. Example 2.4 illustrates one of these usages.

Example 2.4. The program in Figure 4 filters and redirects packets. Any UDP packets to the port defined in the Variable `drop_port` will be dropped. Meanwhile, all UDP packets to the port defined by the Variable `redirect_port` will have its destination port changed to `drop_port`. This logic is contained in the `cond do` of line 11. Packets are dropped with the `Honey.XDP.drop()` function and are passed on with the `Honey.XDP.pass()` function.

```
01 defmodule DropUdp do
02   use Honey, license: "Dual BSD/GPL"
03   @sec "xdp_md"
04   def main(_ctx) do
05     drop_port = 3000
06     redirect_port = 3001
07     Honey.Ethhdr.init()
08     protocol = Honey.Ethhdr.ip_protocol()
09     if protocol == Honey.Ethhdr.const_udp() do
10       port = Honey.Ethhdr.destination_port()
11       cond do
12         port == drop_port -> Honey.XDP.drop() # Drop port 3000
13         port == redirect_port -> # Redirect 3001 to 3000
14            Honey.Ethhdr.set_destination_port(drop_port)
15         true -> 0 # Do nothing
16       end
17     end
18     Honey.XDP.pass() # Pass all other cases.
19   end
20 end
```

Figure 4. Program that drops UDP packets from port 3,000, and redirects port 3,001 to 3,000.

3 Design Principles

Honey Potion translates Elixir code into eBPF code. In this effort, Honey Potion reuses two infrastructures: the Elixir interpreter, and the clang C-to-eBPF translator. Figure 5 shows how these tools are used. The passes in Figure 5, e.g., "Normalization", "Optimizations", etc, were implemented by the authors of this paper on top of the Elixir AST interpreter. They constitute what we call Honey Potion. This section explains the key design decisions that underlie this system.

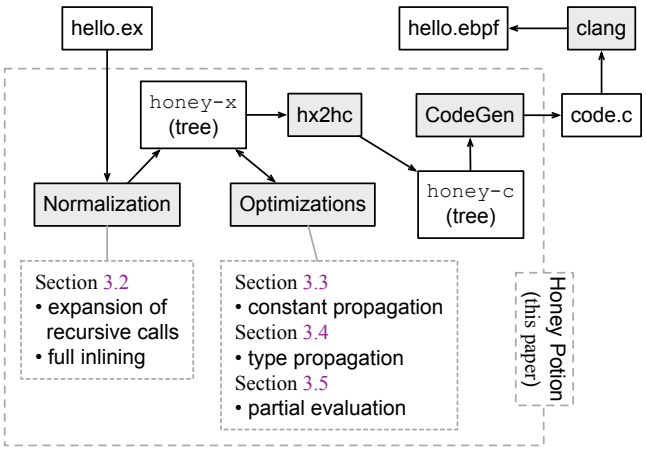

Figure 5. The Honey Potion translation process. White boxes denote code (stored in some specific program representation). Gray boxes denote tools that translate or optimize code.

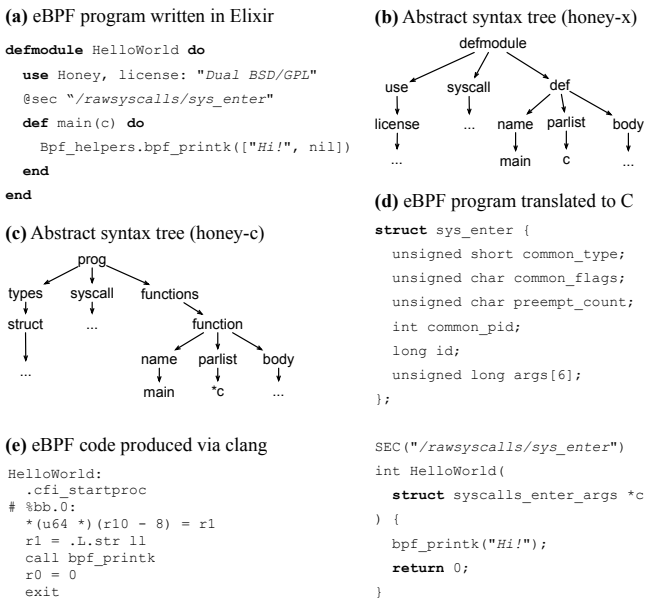

Figure 6. Intermediate representations of Honey Potion.

3.1 Intermediate Program Representation

While producing eBPF code, Honey Potion uses two intermediate representations. In this paper, these two code formats shall be called honey-x and honey-c. Both these representations are tree-like data structures implemented in Elixir. The first of them, honey-x, is close to the Elixir abstract syntax tree. It is onto this representation that Honey Potion applies the optimizations that we describe in Sections 3.3, 3.4 and 3.5. The second intermediate representation, honey-c, represents elements of a C program; hence, it would be closer to an abstract syntax tree produced by a C parser. This representation exists to facilitate code generation. The honey-c AST encapsulates both type and structural information of the generated C program. In addition to easing code translation, honey-c offers a foundation upon which future C-specific optimizations could be implemented. Honey Potion converts honey-c directly to C code. This C program will be translated, by clang, into eBPF instructions. This program is constructed in a way to pass the test of the eBPF verifier that runs on the standard Linux kernel (version 4.9 or higher).

Example 3.1. Figure 6 shows the different formats of a program passing through Honey Potion's pipeline. Honey-x and honey-c are tree-like representations. These trees are fully expanded: they do not contain recursive calls. Nodes in the tree in Figure 6 (b) refer to names declared in the Elixir code in Figure 6 (a). Names in the tree in Figure 6 (c) refer to the program in Figure 6 (d). This C program is created after a traversal of the tree in Figure 6 (c). The translation from Figure 6 (d) to Figure 6 (e) is done by clang.

3.2 Primitive Recursion and The Shape of Programs

Honey-x programs are *strong normalizing* [27]. In other words, they always terminate. Termination is guaranteed because honey-x trees do not contain recursive calls. In fact, honey-x programs do not contain function calls at all: functions present in the original Elixir program are fully inlined into a honey-x tree. This process of normalization that translates Elixir into honey-x follows two steps:

Expansion: Every recursive function in the Elixir program is fully expanded.
Inlining: After expansion, function calls are inlined.

Petrol Semantics. To ensure that the first step above—expansion—is possible, Honey Potion requires that recursive calls are prefixed with an extra parameter: its "*fuel*". This positive integer explicitly determines the maximum number of recursive calls of the function that it qualifies. The same function can be invoked multiple times within the same Elixir program. Its honey-x representation will contain as many expanded calls as the maximum value of fuel passed to it. Example 3.2 illustrates the usage of this extra parameter.

Example 3.2. The program in Figure 3 contains two recursive calls qualified with the "fuel" parameter. The function `fact` will be represented in honey-x as a sequence of two expanded calls. The program in Figure 7 (a) is the Elixir version of the code that would be represented in honey-x[4]. Notice that this program manipulates values of generic type. We

[4]To ease presentation, we represent honey-x code as non-recursive Elixir programs. However, honey-x is a program representation implemented as a tree-like data structure. It is not a programming language.

denote this lack of static type information by subscripting every binary operation with the "gen" qualifier.

```
01 def main(ctx) do
02   const = 3
03   id = ctx.pid
04   a = (
05     x = const
06     if (x ==gen 1) do 1
07     else
08       x1 = x -gen 1
09       x * if (x1 ==gen 1) do 1
10       else
11         x2 = x1 -gen 1
12         x1 *gen if (x2==gen 1) do 1
13         else
14           raise "No more fuel"
15     end end end )
16   b = (
17     y = id
18     if (y ==gen 1) do 1
19     else
20       y1 = y -gen 1
21       y *gen if (y1 ==gen 1) do 1
22       else
23         raise "No more fuel."
24     end end )
25   a +gen b
26 end
```

(a)

```
def main(ctx) do
  const = 3
  id = ctx.pid
  a = (
    x = 3
    if (3 ==gen 1) do 1
    else
      2 = 3 -gen 1
      3 * if (2 ==gen 1) do 1
      else
        x2 = 2 -gen 1
        2 *gen if (1==gen 1) do 1
        else
          raise "No more fuel"
  end end end )
  b = (
    y = id
    if (y ==gen 1) do 1
    else
      y1 = y -gen 1
      y * if (y1 ==gen 1) do 1
      else
        raise "No more fuel."
  end end )
  a +gen b
end
```

(b)

Figure 7. (a) Expanded version of the `fact` function seen in Figure 3. (b) Program after constant propagation.

This idea of bounding the maximum depth of the recursion stack with constant qualifiers is what McBride [21] calls *petrol-driven semantics*. In McBride's words, "*A simple way to give an arbitrary total approximation to partial computation is to provide an engine which consumes one unit of petrol for each recursive call it performs, then specify the initial fuel supply*". Petrol-driven semantics constrains `honey-x` to the set of *Primitive Recursive Functions*, i.e., functions that can be computed by programs whose loops are all bounded [6].

3.3 Constant Propagation

Due to the design decisions mentioned in Section 3.2, the shape of any `honey-x` program admits a topological ordering. This property simplifies the implementation of flow-sensitive data-flow analyses. Any `honey-x` program is in Static Single-Assignment form [10], because every variable identifier is renamed upon assignment. Consequently, data-flow analyses that associate information with variable names (e.g., sparse analyses [33] like constant propagation) can be implemented in a single pass over a `honey-x` program.

Example 3.3. Figure 7 (b) shows the program in Figure 7 (a), after constant propagation. One of the original function calls, in Line 13 of Figure 3, receives a constant argument. This constant can be used to solve every computation related to that computation, in Lines 02-15 in Fig. 7 (b).

3.4 Gradual Typing

Honey Potion meets the two core properties of a gradually typed system, namely, partial static verification and type propagation [19, 29]. In particular, if $v = v_1 \odot v_2$ is any operation within a Honey Potion program, we have that:

Verification If $Type(v_1) = Known$ and $Type(v_2) = Known$, and $Compatible(\odot, Type(v_1), Type(v_2)) = \texttt{false}$, then compilation fails.

Propagation If $Type(v_1) = Known$ and $Type(v_2) = Known$, then $Type(v) = Known$.

There are two sources of static type information in Honey Potion: literals, and the `ctx` parameter. Honey Potion recognizes three types of literals: integers, quoted strings, and characters. Integers are statically represented as a 64-bit signed number, and characters as eight-bit unsigned numbers. String literals are represented as contiguous sequences of characters allocated statically. The second source of static type information is `ctx`, a mandatory argument of every eBPF program. This parameter is represented as a named record; that is, a tuple whose fields are referred by name, instead of numeric indices. The exact set of fields within `ctx` depends on the *trace point* declared in the Elixir program. Example 3.4 illustrates these concepts.

Example 3.4. The function main in Figure 1 and Figure 2 receives one argument: the ctx record, whose structure is determined by the declaration of the trigger. Both share the first four fields of known type: `unsigned short common_type`, `unsigned char common_flags`, `unsigned char common_preempt_count`, `int common_pid`. However, Figure 1 has extra fields of type `int __syscall_nr`, `int pid` and `int sig` and Figure 2 has extra fields `long id` and `unsigned long args[6]`.

At running time, Honey Potion represents values as either generic types, or as one of the built in types that Section 4.2 introduces. Generic types are implemented as in typical dynamically-typed programming languages: as a value plus a type tag. Generic types occupy 16 bytes, which is the type tag plus the size of an eight-byte integer. Honey Potion propagates type information statically known throughout the `honey-x` representation in a single pass—consequence of the acyclic nature of this format. During type propagation, Honey Potion also type checks the program, ensuring that types statically known are correctly used. Type propagation is a simple data-flow optimization, whose implementation is very similar to constant propagation. In addition to replacing generic values with their primitive representations, type propagation might eliminate type-checking rules, similarly to what the "corpse reviver" technique recently proposed by Moy et al. [24] does. Figure 8 shows typical constant and type-propagation rules used in Honey Potion, and Example 3.5 concludes this section.

Example 3.5. The program in Figure 9 (a) continues our running example, showing the effect of type propagation on the representation earlier seen in Figure 7 (b). For presentation purposes only, we illustrate the existence of type information

```
v = cond do
  c1 -> e1
  c2 -> e2
  ...
  cn -> en
end
```

$$\frac{\text{val}(e1) = n \quad \text{val}(e2) = n \quad \ldots \quad \text{val}(en) = n \quad n \neq \text{NAC}}{\text{val}(v) = n}\ [\text{Cond-C}]$$

$$\frac{\text{tp}(e1) = T \quad \text{tp}(e2) = T \quad \ldots \quad \text{tp}(en) = T \quad T \neq \text{GEN}}{\text{tp}(v) = T}\ [\text{Cond-T}]$$

Figure 8. Example of a constant-propagation rule (Cond-C) and a type-propagation rule (Cond-T) used in Honey Potion to perform data-flow analyses on conditional assignments in Elixir.

by subscripting operations with the type inferred for their parameters. As such, the two assignments in Figure 9 (a), at Lines 04 and 16 can have their types fully resolved. Notice that constant propagation is able to fully resolve the first assignment, but not the second; that is, `ctx.id` is known to have an integer type, but its value is not statically known.

```
01 def main(ctx) do
02   const = 3
03   id = ctx.pid
04   a = (
05     x = 3
06     if (3 ==int 1) do 1
07     else
08       2 = 3 -int 1
09       3 * if (2 ==int 1) do 1
10       else
11         x2 = 2 -int 1
12         2*int if (1==int 1) do 1
13         else
14           raise "No more fuel"
15     end end end )
16   b = (
17     y = id
18     if (y ==int 1) do 1
19     else
20       y1 = y -int 1
21       y *int if (y1 ==int 1) do 1
22       else
23         raise "No more fuel."
24     end end )
25   a +int b
26 end
```

(a)

```
def main(ctx) do
  const = 3
  id = ctx.pid
  a = (
    x = 3
    if (3 ==int 1) do 1
    else
      2 = 3 -int 1
      3 * if (2 ==int 1) do 1
      else
        x2 = 2 -int 1
        2*int if (1==int 1) do 1
        else
          raise "No more fuel"
    end end end )
  b = (
    y = id
    if (y ==int 1) do 1
    else
      y1 = y -int 1
      y *int if (y1==int 1) do 1
      else
        raise "No more fuel."
    end end )
  6 +int b
end
```

(b)

Figure 9. (a) Program after type propagation. (b) Program after partial evaluation.

3.5 Partial Evaluation

After constant and type propagation, Honey Potion applies partial evaluation on `honey-x` programs. As already observed by previous work [30], partial evaluation serves a number of purposes in the compilation pipeline: first, it removes assignments that, after constant propagation, define names that are no longer used in the program's code. Second, it removes conditional tests that can be solved at compilation time. Partial evaluation is implemented using the interpreter itself. The `honey-x` program is evaluated, and every computation that is statically solvable is eliminated. Notice that partial evaluation can be seen as a form of dead-code elimination, because the evaluation of conditional tests that use only constants might leave control-flow regions dead—these regions are eliminated in the process of partial evaluation. The next example illustrates the effects of partial evaluation.

Example 3.6. Figure 9 (b) shows the program that results from applying partial evaluation on the code in Figure 9 (a). Partial evaluation yields the final value of variable `a`, which is replaced with the constant six in Line 25 of Figure 9 (b).

4 Implementation Decisions

This section explains some implementation decisions that have been adopted in the project of Honey Potion. These decisions are orthogonal to the principles enumerated in Section 3. However, our decisions are heavily motivated by the shape of programs, which Section 3.2 explained.

4.1 Memory Organization

The eBPF runtime separates only 512 bytes for the allocation of local variables. The region is equivalent to the "stack" used in typical programming languages to store the activation records of functions. Honey Potion programs have just one activation record, because after program expansion, `honey-x` does not contain function calls. Honey Potion stores in this stack space values of known types, and pointers to generic values. The generic values themselves are stored into an eBPF map, which we shall call the "Honey Stack".

Maps and The Honey Stack. In eBPF, data is exchanged between the frontend and the backend parts of a program through associative arrays called "eBPF" maps. Elixir maps are translated into eBPF maps as C structs with four fields: the type of the map, the maximum number of entries, the size of keys and the size of values. This pattern follows the built-in signature of maps in eBPF.

Honey Potion uses one map to circumvent the stack size constraint of 512 bytes. This map, the "honey stack," works as a contiguous block of memory, enabling read and write operations at arbitrary positions. The size of the honey stack is set prior to program initialization and remains immutable thereafter. Currently, Honey Potion allocates 2 kB to the honey stack. The honey stack accommodates values that in Elixir would be stored in the heap. Nevertheless, due to the shape of `honey-x` programs, allocation always happens in a stack-like fashion. These programs do not contain backedges; therefore, every allocation site is traversed at most once by the program flow. Honey Potion does not do deallocation: once a variable is allocated, it remains in place for the lifetime of the program. However, allocation is not static: the address of variables might differ depending on the program flow, for Honey Potion programs still have conditional assignments.

4.2 Representation of Built-in Types

Honey Potion supports seven built-in Elixir types: integers (which can be either four or eight-byte long), strings, tuples, list literals, atoms, generic values and maps (not general Elixir maps; rather, those defined via `defmap`, as in Figure 1).

Figure 10 shows the representation of these types in memory. Honey Potion does not implement a special representation for linked lists. They must be implemented as tuples, e.g., `t = :nil, t = [A|t], t = [B|t]` represents the list `[B, A]`. However, list literals (e.g., lists with constants known at compilation time, such as `[1, 2, 3]`) are implemented as a single tuple. The fact that recursive calls are fully expanded ensures that the maximum size of any list is known statically. Example 4.1 details how tuples are represented.

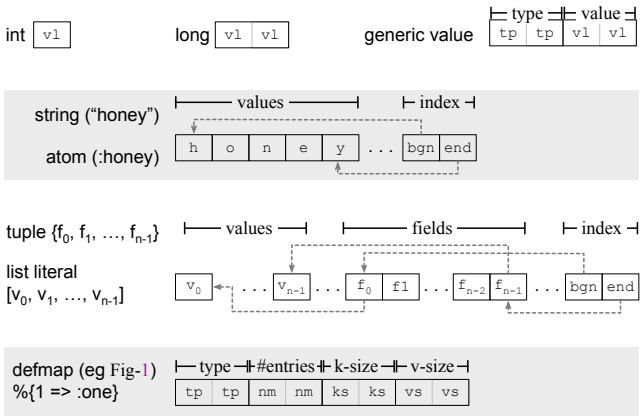

Figure 10. Implementation of built-in Elixir types in Honey Potion. Each box corresponds to four bytes.

Example 4.1. Figure 11 shows how Honey Potion stores a three-element tuple. Assuming an assignment such as `t = {x, y, z}`, then variable `t` is allocated in the standard 512-byte eBPF stack. This variable holds an *index* that points to the honey stack. All indices in the honey stack are stored as 32-bit values. In this example, variable `t` refers to the 100^{th} byte in the honey-stack: that is the location where the tuple is stored. Tuples are stored as pairs denoting the index of their first and last fields. Our example contains three fields. Each field is stored as a 32-bit index. The beginning of `t`, a field called `bgn`, stored in address 100, points to the index of `t`'s first field (e.g., `t.x`), which is stored in address 88 of the honey stack. The end of `t`, a field called `end`, stored in address 104, points to the index of the last field of `t`, which is stored in address 96. Each index points out to a generic value. Each generic value occupies 16 bytes. The pair, the indices and the values are all allocated in the honey stack.

The representation of lists and tuples might seem, at first, inefficient: structs in C occupy exactly the size of their fields. However, Honey Potion programs must satisfy the eBPF verifier: every memory access is guarded by a conditional statement that certifies that the access is within bounds; hence, composite types must carry bound information, like typical fat pointers [11]. Additionally, Example 4.1 shows that the representation of dynamic values requires a non-trivial amount of space. However, type propagation, as explained

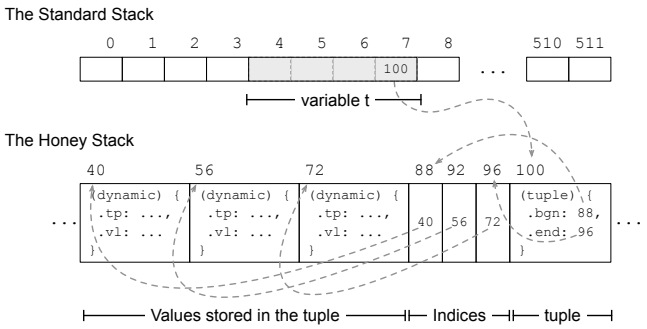

Figure 11. Allocation of tuple `t = {x, y, z}`.

in Section 3.4, can save much of this space. As an example, Figure 12 shows the effect of constant and type propagation on the tuple seen in Figure 11. In practice, type propagation gives to Honey Potion the same effects as the "unboxing" optimization implemented in the `Truffle` Virtual Machine, for instance. To emphasize this last point, we quote Wöß et al. [37]: "*If possible, primitive values are directly stored in the object without boxing or tagging, and can be accessed without any additional indirection*". As seen in Figure 12, the same effect occurs in Honey Potion.

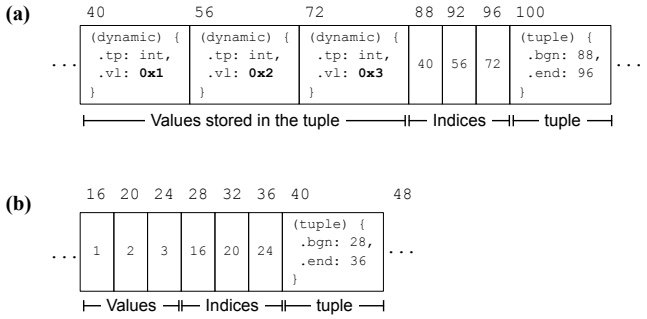

Figure 12. (a) Effect of constant propagation in the tuple seen in Figure 11. (b) Effect of type propagation.

4.3 Exceptions

Honey Potion handles five exceptional runtime events: (i) non-existent key in map; (ii) access to honey stack out of range; (iii) non-existent field in tuple; (iv) access to string out of range; (v) incompatible type operation in dynamically typed value. These events stem from the dynamic nature of Elixir and from the need to satisfy the eBPF verifier. Regarding this last observation, every memory access in Honey Potion is guarded with conditional checks. The verifier uses these guards to certify that the eBPF code produced by Honey Potion is memory safe. Exceptional runtime conditions are treated via a handler at the end of the eBPF code. Thus, an exception causes two events: first, the error code is stored for posterior inspection; second, the program flow is diverted to the end of the eBPF code, where the handler is implemented.

The only action implemented by the handler is to return control to kernel code, passing the right error message, once an exception happens. that future releases of Honey Potion will add support to Elixir exceptions.

Compile-Time Exceptions. The program expansion discussed in Section 3.2 amounts to interpreting the Elixir source code given to Honey Potion. In this process, it is possible that the code generator runs into three Elixir exceptions: (i) declaration of a non-existing trigger; (ii) use of an undefined map; (iii) map indexed with literal keys that are not Elixir atoms. We call such events *compile-time exceptions*. In normal Elixir code, these exceptions would set off at running time; however, in Honey Potion, they trigger at code generation time. These exceptions stop the code generation process and issue an error message to the user.

4.4 Tool Chain

Honey Potion provides users with a standard library. The library contains bindings to `libbpf` backend functions to handle logging, updating and retrieving data from maps. The library also includes utility functions for accessing and modifying packet headers. Calls to the standard Honey library are directly translated to their counterparts in `libbpf`. Section 2 contain several examples of calls to functions in the standard Honey library. These functions exist in name spaces such as `Honey.Bpf_helpers` or `Honey.Ethhdr`.

The Compilation Pipeline. To compile an Elixir program to eBPF via Honey Potion, all that a user must do is to add the "`use Honey`" directive to the module that defines that program. Honey Potion is integrated into Elixir as a mix library—a build tool distributed with Elixir. Honey Potion is invoked onto every elixir file (`.ex` file) that contains the "`use Honey`" pragma. Honey Potion is plugged into the Elixir infrastructure via the `__before_compile__` call back, which is part of the standard implementation of Elixir. This hook gives Honey Potion access to the abstract syntax tree of the target Elixir program. The optimizations discussed in Section 3 run at this point, over the `honey-x` representation. It is possible for a human to observe the effects of each optimization on the textual representation of `honey-x` programs.

The optimized `honey-x` code is translated into `honey-c`. The code generator uses this representation to produce a C file. This file is compiled into eBPF bytecodes using Clang, with the standard flags recommended by `libbpf-bootstrap` (libbpf-bootstrap). Subsequently, Honey Potion invokes `bpf tool` (bpftool) to convert the generated eBPF bytecodes into an eBPF program skeleton. An eBPF skeleton encapsulates function interfaces for loading, attaching, and manipulating the eBPF program, as well as data structures containing metadata about the eBPF program and associated maps.

The Custom Frontend. Honey Potion generates code for eBPF backends; that is, it produces the eBPF code that runs on the BPF virtual machine. A frontend is still needed to access this code. Users can write their own frontends; however, Honey Potion generates a custom frontend for fast deployment. This program opens, loads and attaches the backend code into the Linux trigger that is defined with the `@sec` pragma. The Linux verifier runs upon the backend, once it is loaded by the frontend. In addition to these standard responsibilities, the custom frontend prints logging data (see the `printlist` in Figure 2). Users can determine, through the frontend's command line interface, for how long the backend will remain attached. In terms of code, the frontend is generated as a C file that encapsulates the eBPF bytecodes. This program is compiled into a standalone executable.

5 Experimental Evaluation

This section explores the following research questions:

- **RQ1:** How does Honey Potion programs compare with equivalent C programs in number of lines of code?
- **RQ2:** How does Honey Potion programs compare with equivalent C programs in terms of size of binary code?
- **RQ3:** What is the relative performance of programs compiled via Honey Potion and hand-coded C programs?
- **RQ4:** What is the impact of optimizations onto the size of the `honey-x` AST produced via Honey Potion?
- **RQ5:** How does the compilation time of Honey Potion compare to the compilation time of handwritten C?

Hardware/Software. Experiments evaluated in this section were performed on an AMD Ryzen 7 5700X, featuring 16.68 GB of RAM, clock of 3.4 GHz and eight 64 kB L1 caches. The experimental setup runs on EndeavourOS Linux Cassini Nova R3 with the Kernel 6.1.61-1-lts, featuring the eBPF instruction set version 1, Clang and LLVM version 16.0.6 (Jun'23) and libbpf version 1.2.2 (Jul'23). Honey Potion is implemented onto Elixir version 1.15.7 (Oct'23).

Benchmarks. There is not a standard benchmark suite for eBPF programs. Thus, to provide the reader with some perspective on how Honey Potion programs compare with equivalent C programs, we have recoded common eBPF examples from `BPFabric`[5] and `IOVisor`[6]. These programs appear in Figures 13 and 14. Additionally, Section 5.4 uses all the programs available in the Honey Potion test suite.

5.1 RQ1: Code Size

This section compares the number of lines of code between eBPF programs written in Elixir and equivalent handwritten C programs. None of the C programs used in this experiment were designed by the authors of this paper: they were taken from public repositories and adjusted to run in our setup. We count lines of code via the `tokei` tool[7]. `Tokei` counts any

[5] https://github.com/UofG-netlab/BPFabric/tree/master (May 7th, 2024).
[6] https://github.com/iovisor/bpftrace (May 7th, 2024).
[7] https://github.com/XAMPPRocky/tokei (May 7th, 2024).

line that is not a comment or an empty space as a line of code. Figure 13 shows these results.

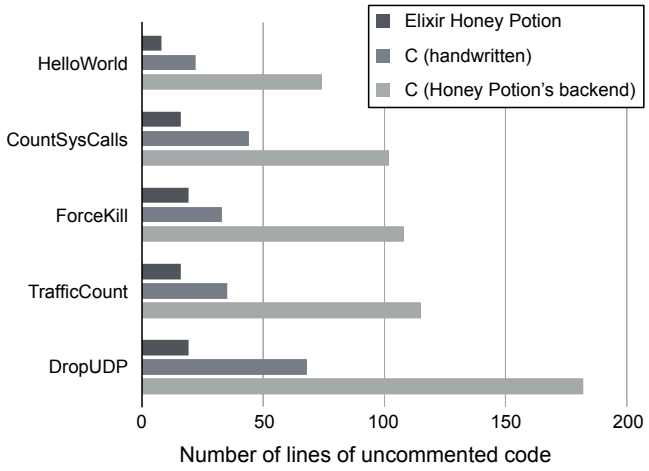

Figure 13. Code size, measured in number of lines of code.

Discussion. Figure 13 shows the size of three groups of programs: the Elixir programs that will be translated into eBPF code; the equivalent C program—which is handwritten by an eBPF developer; and the C code that Honey Potion translates from the Elixir programs. Figure 13 shows that programs in the first group are consistently shorter: in total, these programs add up to 78 lines of code. The handwritten C programs, in turn, add up to 202 lines of code. Finally, the automatically produced C programs contain, together, 581 lines of code. They are substantially larger; however, users of Honey Potion will never interact with these programs.

5.2 RQ2: Binary Size

We measure binary size as the size, in bytes, of the ePBF programs either written manually, or produced via Honey Potion. This binary code includes both the frontend (the code that is actually compiled by clang) and the backend (the code imported as a `.skel.h` file containing the eBPF bytecodes). Figure 14 reports these results.

Discussion. Binaries produced out of handwritten C programs are 2.36x shorter than binaries produced via Honey Potion. The Honey Potion binaries seen in Figure 14 add up to 191,704 bytes. The equivalent C programs add up to 81,352 bytes. Honey Potion programs incur a constant size overhead due to its runtime, such as boilerplate code to read and write maps (Section 4.1), or to handle exceptions (Section 4.3).

5.3 RQ3: Performance

Measuring performance of eBPF programs is not easy: programs run for a very short time, and variance in running time is high. To estimate the performance of Honey Potion programs compared with equivalent C codes, we use the

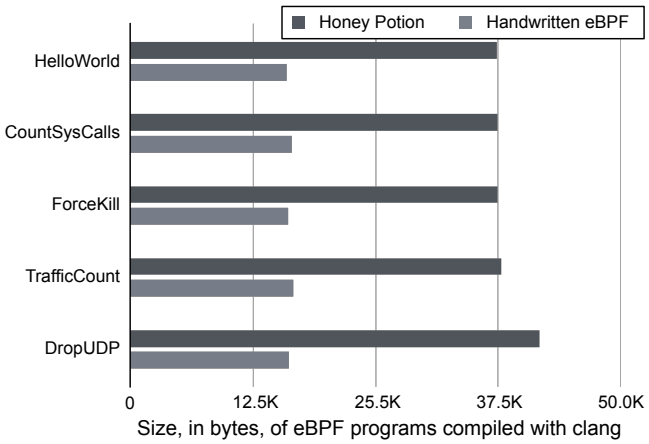

Figure 14. Binary size in bytes, of `.text` region.

`bpftool` utility in profiling mode. This tool reports an estimate of the number of cycles executed whenever the eBPF backend is invoked, and the number of instructions executed in such events. Notice that it is difficult to control how often each backend is invoked. For instance, our implementation of `TrafficCount` runs whenever a new packet arrives. Thus, we report averages: number of instructions divided by number of times each handler was triggered. The methodology for triggering each eBPF handler varies according to the benchmark. For instance, for `CountSysCalls` we let the handler active for one second. For `DropUDP` we send two packets to the Linux server: one to port 3,000, and another to port 3,001, with contents "hi" and "hello", respectively.

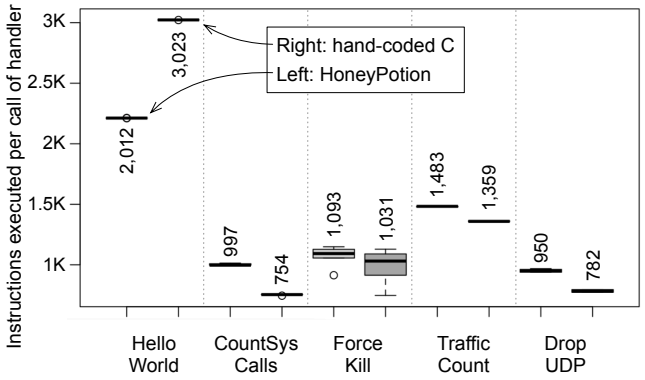

Figure 15. Number of instructions executed per activation of the eBPF backend. Numbers next to boxes are the median of five measurements.

Discussion. Figure 15 shows the average number of eBPF instructions fetched per trigger of the eBPF handler for the five programs already used in Sections 5.1 and 5.2. In four cases, the hand-coded C program runs less instructions per activation of the backend. In one of the cases: `HelloWorld`, Honey Potion yields a smaller instruction count than the C

counterpart. However, we emphasize that these numbers are estimates: the total number of instructions fetched by the eBPF virtual machine divided by the number of times the machine was activated.

5.4 RQ4: Optimizations

This section investigates the impact of constant propagation (Section 3.3) and dead-code elimination (Section 3.5) on the binary code produced by Honey Potion. We perform this study on the 20 programs in Figure 16. This suite includes the five programs evaluated in Sections 5.1 and 5.2, plus fifteen programs distributed with Honey Potion. We compare the effects of optimizations by counting the nodes of the `honey-x` representation after each optimization runs. Type propagation (Section 3.4) cannot be disabled, otherwise many programs in Figure 16 cannot be compiled. Without type propagation, they need more than 512 bytes of stack to run. As an illustration, consider a program that simply declares a map m, plus a number n of variables and, after the declaration point, inserts these variables into m. Lines 12 and 13 of Figure 1 show a case where $n = 1$, for instance. Without type propagation, the maximum n is 3; with it, $n = 8$.

Discussion. Figure 16 shows the selective effect of the optimizations on the different programs. The `honey-x` trees of the unoptimized programs add up to 971 nodes. Dead-code elimination—alone—brings this number down to 927. Constant propagation, also alone, yields 849 nodes. These two optimizations together produce 670 nodes—a reduction of 1.45x. This effect is stronger on recursive programs. For instance, revisiting Example 3.5, we see that the partial evaluation of the factorial function seen in Figure 3 shrinks its `honey-x` tree from 183 to 101 nodes (as seen in Figure 9).

5.5 RQ5: Compilation Time

This section evaluates the compilation time of Honey Potion programs. To this end, we compare the time taken to produce eBPF code out of Honey Potion programs with the time taken to compile equivalent C programs to eBPF. Notice that the compilation time of Honey Potion programs is the sum of two processes: the first converts Elixir code to C; the second converts C to eBPF. The latter step is performed via `clang`, as explained in Section 4.4; the former is implemented on the Elixir interpreter, and involves all the transformations described in Section 3. We measure compilation times in the following way:

- To measure the time taken to convert Honey Potion programs to C files, we use `mix compile -force -profile time` with the compilation step disabled.
- To measure the time taken to compile Honey Potion's C files to binary eBPF code, we use `time make`. In this case, we always clean past compilation steps to prevent `make` from reusing `.o/.ll/.skel` files.
- To measure the time taken to compile the Handwritten C files to binary eBPF code, we use `time make`.

Discussion. In Figure 17, we compare compilation times of Honey Potion and handwritten C. All the times are in milliseconds. Generally, the time to compile Honey Potion programs to eBPF is approximately twice that of handwritten C programs to eBPF. Yet, we emphasize that the Honey Potion pipeline involves two steps, including invoking the Beam virtual machine to run the Elixir interpreter. If we sum up the time taken on each benchmark, we observe that the conversion of Elixir to C takes 352.2 ms, and the translation of C to eBPF takes 948.8 ms. The time to convert handwritten C into eBPF, in turn, takes 677.0 ms. All these numbers are the arithmetic average of five executions.

Notice that Honey Potion's reliance on Partial Evaluation does not significantly compromise its compilation time. As evidence to this fact, we notice that the time to translate C into eBPF, via `clang`, is more than twice the time to translate Elixir into C, via the Elixir interpreter. In this regard, partial evaluation is guaranteed to terminate due to the bounded recursion present in every Honey Potion program. Secondly, the asymptotic complexity of partial evaluation, in the implementation of Honey Potion, is proportional to the size of the binary program produced at the end of the compilation process. To support this observation, we notice that Spearman's rank correlation coefficient between compilation time and the size of the binary that Honey Potion produces, at least for these five benchmarks, is 1.0.

6 Correctness

One of the main goals of Honey Potion is to shield developers from the intricacies of the Linux eBPF verifier. To achieve this goal, the implementation of Honey Potion meets two properties:

Termination: Every eBPF program that is generated via Honey Potion terminates. Code produced via Honey Potion does not contain back edges: every recursive call is fully inlined at code-generation time.

Indexation: Every eBPF program that is generated via Honey Potion only access memory within allocated bounds. The aggregate data structures: tuples, atoms, strings, lists and maps all come with bounds information. Access to these structures is guarded by if-then-else checks, to satisfy the verifier.

Although we believe that these properties are met by our implementation of Honey Potion, we have not proved the correctness of this implementation mechanically, as it has been done, for instance, with the CompCert compiler [17]. Producing such a proof would go beyond what we can do in terms of manpower and expertise. However, we have performed two exercises to enhance our confidence in the correctness of the Honey Potion implementation:

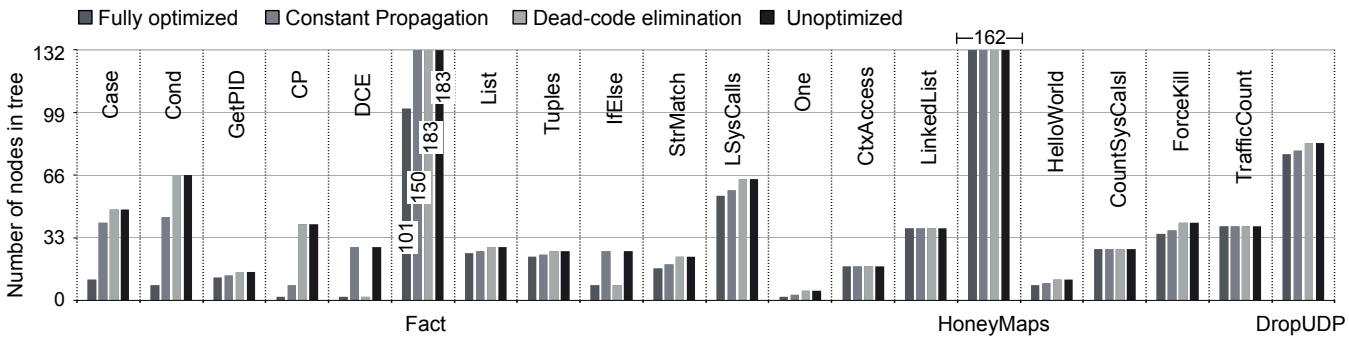

Figure 16. The effect of different optimizations on programs in the Honey Potion test suite. The shorter the bar, the more optimized the code. Every tree of the `HoneyMaps` program contains 162 nodes.

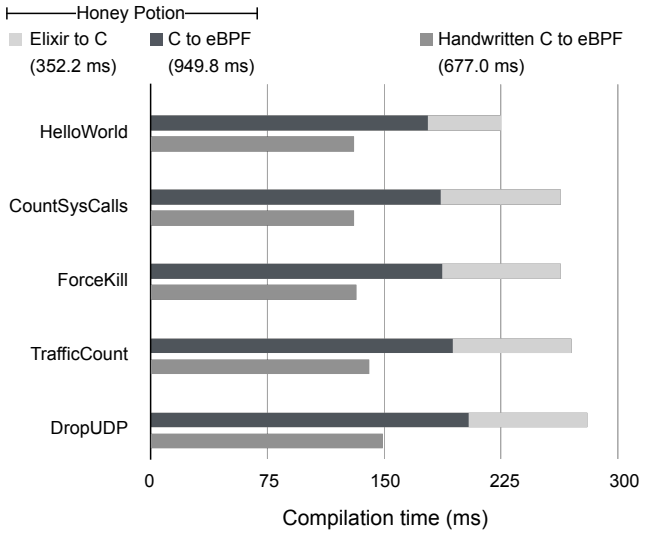

Figure 17. Compilation time, in milliseconds.

1. We have proved two theorems about a prototype (written in Haskell) of this implementation using the Agda Proof Assistant [31].
2. We have exhaustively tested four properties about this prototype using QuickCheck's Property Based Testing [9].

In what follows we describe these theorems and properties.

6.1 Termination of Code-Generation

Code generation in Honey Potion involves partial evaluation with full inline expansion. The termination of this process is assured because inline expansion is always guarded by an integer parameter—the "fuel"—which decreases upon each expansion. To demonstrate this fact, we have built two prototypes of Honey Potion, which we call System R and System L. The former is a version of the simply typed lambda calculus extended with pattern matching and recursive functions and the latter is a version of the simply typed lambda calculus with just pattern matching, where every recursive call has been fully expanded. These two prototypes were implemented in Haskell. We have been able to formalize two properties of them using the Agda proof assistant:

1. All programs in System L terminate.
2. Let $t' = \texttt{transform}(t, n)$ be a function which takes a system R term t and a natural number n and returns an equivalent system L term t' which corresponds to, at most, n recursive calls of system R. The function $\texttt{transform}(t, n)$ terminates and preserves types.

6.2 Semantic Correctness

We have no formal proof that our Implementation of Honey Potion preserves the semantics of Elixir programs. However, we have been able to validate, via exhaustive testing (full coverage), that our transformation from System R to System L preserves semantics. In this case, the validation happens using Property Based Testing, via Haskell's QuickCheck. To achieve full coverage, we have synthesized random System R programs. Synthesis happens by a type-directed procedure which ensures that every generated program is well-typed. Our tests cover the four properties listed below:

P1: All System R programs that pass type verification are well typed.

P2: All System R programs that pass type verification terminate.

P3: For every generated System R term t, if t terminates with value v doing at most f recursive calls, then $\texttt{transform}(t, f)$ yields a System L program that produces v.

P4: For every generated System R term t and some fuel f, if $\texttt{transform}(t, f)$ is a System L program that produces a value v, then t terminates resulting in v.

7 Related Work

The most widely used language for writing eBPF programs is C. EBPF programs written in C are compiled using the LLVM-based Clang compiler. The code is then loaded into the kernel using tools like `bpftool` or `ip`. Nevertheless, other

programming languages can be used as a way to load eBPF programs into the Linux kernel. Python, for instance, is commonly used in this setting. The Python program in Figure 18, for instance, uses the `bpfcc` library to trace system calls to the `open` function. This program is similar to the example that we showed in Figure 2: it prints occurrences of a specific system call. However, in contrast to the program in Figure 2, in the case of Figure 18, no Python code is directly translated to BPF instructions. Instead, a string containing a C program (or more precisely, a C-like program) is translated to BPF bytecodes. A similar approach is used, for instance, in Go, via the `gobpf` library: a C program, represented as a string in Go, is loaded and attached into the Linux kernel. Thus, to the best of our knowledge, Honey Potion is the first compiler for a functional language that fully translates it into eBPF. In this sense, it brings to the functional world—already vibrant with practical use cases [14]—one original practical application.

```
from bcc import BPF
bpf_text = """
#include <uapi/linux/ptrace.h>
int syscall__open(struct pt_regs *ctx, const char *filename) {
    bpf_trace_printk("open: %d %s\\n",
    bpf_get_current_pid_tgid(), filename);
    return 0;
}
"""
b = BPF(text=bpf_text)
b.attach_kprobe(event="sys_open", fn_name="syscall__open")
print("Tracing sys_open... Ctrl+C to exit.")
while True:
    try:
        (task, pid, cpu, flags, ts, msg) = b.trace_fields()
        print("PID %d: %s" % (pid, msg))
    except KeyboardInterrupt:
        break
```

Figure 18. Python program that prints system calls to the `open` function.

Compilation of Terminating Programs. Dependently-typed programming languages and proof assistants demand the assurance of function termination as a fundamental prerequisite for maintaining the logical soundness of their proof verification processes [2, 5, 21]. Compiler designs tailored for these languages employ specific algorithms to ensure that each recursive call is made on "smaller" arguments [1]. However, the inherent undecidability of the halting problem introduces a challenge: these algorithms may mistakenly flag valid programs as non-terminating. To address this issue and facilitate the definition of recursive functions, developers have employed various techniques, such as well-founded recursive relations [26], monads [7] and the concept of "fuel" [21]. As explained in Section 3.2, this "petrol-semantics" transforms recursive functions into finite-depth pattern matching, thereby guaranteeing their termination.

Trace Compilation. The techniques described in Section 3 resemble the idea of *Trace Compilation* [3]. Indeed, the design of the Honey Potion code optimizer centers around the fact that `honey-x` programs can be optimized non-iteratively, much like Bala et al. does no Dynamo, e.g. "*In fact, the Dynamo trace optimizer is non-iterative, and optimizes a trace in only two passes: a forward pass and a backward pass*". To this effect, optimizations such as constant propagation and dead-code elimination are staples in the project of trace compilers [4, 12, 32]. However, the control-flow graph of a `honey-x` program is not strictly a tree of traces. Rather, it is a direct acyclic graph, for joint points exist due to conditional assignments (for an example, see Figure 8). Nevertheless, every data-flow analysis performed by Honey Potion is non-iterative, happening over the topological ordering of the program's control-flow graph.

8 Conclusion

This paper has presented Honey Potion, a tool that compiles Elixir to eBPF. Honey Potion is open software (GPL-3.0). Tutorials, examples and courses about it are publicly available (links removed due to blind review). The implementation of the compiler uses many techniques already well-known within the programming language community; however, it tailors these techniques to the eBPF setting: a resource constrained environment, where termination is assured by a formal verifier. Elixir code is translated in such a way that the verifier can prove termination and ensure the absence of out-of-bounds memory accesses.

In retrospect, we believe the primary benefit of Honey Potion is its ability to shield developers from the intricacies of the verifier. For some discussion on the correctness of this approach, see Section 6. Our experience is that programming eBPF applications is difficult exactly because developers need to write code that passes automatic verification. For instance, a test like $x + 6 \geq s$ does not prove to the eBPF verifier the validity of $x + 6$ as an address within the string pool s, but $x \geq s - 6$ accomplishes this. Solving such issues is demanding, as the primary documentation for the verifier remains its source code. A significant portion of Honey Potion's design effort was dedicated to avoiding such issues.

A Artifact Appendix

A.1 Abstract

This artifact reproduces the research questions listed in Section 5. Four of the research questions (RQ1, RQ2, RQ4 and RQ5) can be reproduced with a single script each, available in a Docker container (`rq1.sh`, `rq2.sh`, `rq4.sh` and `rq5.sh`). RQ3 requires manual intervention. For a more detailed description of this artifact, check https://github.com/lac-dcc/honey-potion/tree/master/artifact.

A.2 Artifact check-list (meta-information)

- **Program:** The Honey Potion compiler that translated Elixir programs into eBPF code.
- **Compilation:** clang-16 (or higher), Elixir and Erlang.
- **Transformations:** C to eBPF (baseline); and Elixir to C to eBPF (Honey Potion).

- **Run-time environment:** Linux (mandatory!) and Docker. If running on a cloud VM, then use specific VM types known to work well with Linux tools.
- **Hardware:** any hardware that supports Linux.
- **Execution:** via Docker, with sudo privileges
- **Metrics:** size of source code (RQ1); size of binaries (RQ2); number of cycles (RQ3); size of the AST (RQ4) and compilation time (RQ5).
- **Output:** as expected in https://github.com/lac-dcc/honey-potion/blob/master/artifact/expected_outputs/output_rq1.txt (for RQ1). The other RQs have similar outputs.
- **Experiments:** four experiments performed via https://github.com/lac-dcc/honey-potion/blob/master/artifact/rq1.sh and similar scripts.
- **How much disk space required (approximately)?:** 1.75GB for the Docker image.
- **How much time is needed to prepare workflow (approximately)?:** 20 minutes to build the Docker image.
- **How much time is needed to complete experiments (approximately)?:** 2 minutes.
- **Publicly available?:** yes, at https://github.com/lac-dcc/honey-potion/tree/master/artifact
- **Code licenses (if publicly available)?:** GPL-3.0.
- **Workflow framework used?:** Docker

A.3 Description

A.3.1 How delivered. Delivered via Docker, available at https://github.com/lac-dcc/honey-potion/tree/master/artifact/docker, or via Zenodo, at https://zenodo.org/records/13729837.

A.3.2 Software dependencies. The eBPF programs must run on the Linux Operating System, with super-user privileges. The first kernel version that supported eBPF was Linux 3.18, released in December 2014. However, the Honey Potion artifact requires at least Linux kernel 4.12, which allows programs with up to 4,096 eBPF instructions (July 2017). Additionally, it requires clang 16.0 or superior (lower versions can still compile eBPF, but the programs will not fit into the limit of 4,096 instructions). Finally, ensure that your `bpftool` library is compiled with a version of clang newer than 10.0. If you run the eBPF programs through Docker, then the version of clang should not be an issue.

A.3.3 Data sets. The Honey Potion benchmarks are available at https://github.com/lac-dcc/honey-potion/tree/master/examples/lib. The baseline hand-written C benchmarks are available at https://github.com/lac-dcc/honey-potion/tree/master/benchmarks.

A.4 Installation

1. Clone the Honey Potion repository:

```
git clone \
  https://github.com/lac-dcc/honey-potion.git
```

2. Build the image. This step takes about twenty minutes:

```
cd honey-potion/artifact/docker/ ;
docker build -t docker-artifact -f Dockerfile .
```

A.5 Experiment workflow

1. Run RQ1 (Section 5.1): takes about 30 seconds

```
docker run --rm docker-artifact bash rq1.sh
```

2. Run RQ2 (Section 5.2): takes about 30 seconds

```
docker run --rm docker-artifact bash rq2.sh
```

3. Run RQ4 (Section 5.4): takes about 30 seconds

```
docker run --rm docker-artifact bash rq4.sh
```

4. Run RQ5 (Section 5.5): takes about 30 seconds

```
docker run --rm docker-artifact bash rq5.sh
```

A.6 Evaluation and expected result

Text files with expected results are available in the artifact's repository:

RQ1 Section 5.1: https://github.com/lac-dcc/honey-potion/blob/master/artifact/expected_outputs/output_rq1.txt
RQ2 Section 5.2: https://github.com/lac-dcc/honey-potion/blob/master/artifact/expected_outputs/output_rq2.txt
RQ4 Section 5.4: https://github.com/lac-dcc/honey-potion/blob/master/artifact/expected_outputs/output_rq4.txt
RQ5 Section 5.5: https://github.com/lac-dcc/honey-potion/blob/master/artifact/expected_outputs/output_rq5.txt

A.7 RQ3 – Profiling and Performance

RQ3 (Section 5.3) compares the performance of hand-written C programs and programs produced via Honey Potion. This is the only research question that we could not automatize with a single script. Thus, profiling these programs requires manual intervention, following the instructions available at https://github.com/lac-dcc/honey-potion/tree/master/artifact/rq3. As we explain in that tutorial, measuring performance involves running docker with sudo privileges:

```
docker run --rm -ti \
    -e BPF_IMAGE=docker-artifact \
   -v /var/run/docker.sock:/var/run/docker.sock \
    ghcr.io/hemslo/docker-bpf:latest \
    bash
```

Open two shell sessions in the container, one for running the eBPF program, and another for profiling it, as follows:

1. To open the two sessions, let us use `tmux`.
   `root@docker:/honey-potion/artifact# tmux`
   To split the window into two sessions with vertical panes, you can press `ctrl+b` and then press `%` within your `tmux` session.
2. (Within shell session 1) Run the eBPF program. We are assuming that you have already compiled it, e.g., using `bash rq2.sh`, for instance. To run the program,

as already mentioned, you can do:
`root@docker:/honey-potion/artifact# sudo ../examples/lib/bin/HelloWorld`

3. (Within shell session 2) Discover the program's ID. You can find this ID with the command below:
`sudo bpftool prog list`
The eBPF process that you want will be (most likely) the last one in the list.
4. (Within shell session 2) Profile the program using the `bpftool` application. If we assume that the process that you want has ID 63, then you can do it as follows:
`sudo bpftool prog profile id 63 duration\ 5 cycles instructions`

For more details, please, refer to https://github.com/lac-dcc/honey-potion/tree/master/artifact/rq3.

References

[1] Andreas Abel and Thorsten Altenkirch. 2002. A predicative analysis of structural recursion. *Journal of Functional Programming* 12, 1 (2002), 1–41. https://doi.org/10.1017/S0956796801004191

[2] Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. 2017. Normalization by evaluation for sized dependent types. *Proc. ACM Program. Lang.* 1, ICFP, Article 33 (aug 2017), 30 pages. https://doi.org/10.1145/3110277

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System. In *PLDI* (Vancouver, British Columbia, Canada). Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/349299.349303

[4] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *ICOOOLPS* (Genova, Italy). Association for Computing Machinery, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

[5] Ana Bove, Alexander Krauss, and Matthieu Sozeau. 2016. Partiality and recursion in interactive theorem provers - an overview. *Math. Struct. Comput. Sci.* 26, 1 (2016), 38–88. https://doi.org/10.1017/S0960129514000115

[6] Walter S. Brainerd and Lawrence H. Landweber. 1974. *Theory of Computation.* John Wiley & Sons, Inc., USA.

[7] Venanzio Capretta. 2005. General recursion via coinductive types. *Log. Methods Comput. Sci.* 1, 2 (2005), 28 pages. https://doi.org/10.2168/LMCS-1(2:1)2005

[8] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proc. ACM Program. Lang.* 3, POPL, Article 16 (jan 2019), 32 pages. https://doi.org/10.1145/3290329

[9] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[11] Gregory J. Duck and Roland H. C. Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 132–142. https://doi.org/10.1145/2892208.2892212

[12] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In *PLDI* (Dublin, Ireland). Association for Computing Machinery, New York, NY, USA, 465–478. https://doi.org/10.1145/1542476.1542528

[13] 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 *PLDI* (Phoenix, AZ, USA). Association for Computing Machinery, New York, NY, USA, 1069–1084. https://doi.org/10.1145/3314221.3314590

[14] Zhenjiang Hu, John Hughes, and Meng Wang. 2015. How functional programming mattered. *National Science Review* 2, 3 (07 2015), 349–370. https://doi.org/10.1093/nsr/nwv042

[15] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. 2023. Kernel Extension Verification is Untenable. In *HOTOS* (Providence, RI, USA). Association for Computing Machinery, New York, NY, USA, 150–157. https://doi.org/10.1145/3593856.3595892

[16] Florian Latifi. 2019. Practical Second Futamura Projection: Partial Evaluation for High-Performance Language Interpreters. In *SPLASH Companion* (Athens, Greece). Association for Computing Machinery, New York, NY, USA, 29–31. https://doi.org/10.1145/3359061.3361077

[17] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[18] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. 2023. MOAT: Towards Safe BPF Kernel Extension. arXiv:2301.13421 [cs.CR]

[19] Kenji Maillard, Meven Lennon-Bertrand, Nicolas Tabareau, and Éric Tanter. 2022. A reasonably gradual type theory. *Proc. ACM Program. Lang.* 6, ICFP, Article 124 (aug 2022), 29 pages. https://doi.org/10.1145/3547655

[20] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters. In *OOPSLA* (Pittsburgh, PA, USA). Association for Computing Machinery, New York, NY, USA, 821–839. https://doi.org/10.1145/2814270.2814275

[21] Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction*, Ralf Hinze and Janis Voigtländer (Eds.). Springer International Publishing, Cham, 257–275.

[22] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. 2023. Understanding the Security of Linux EBPF Subsystem. In *APSys* (Seoul, Republic of Korea). Association for Computing Machinery, New York, NY, USA, 87–92. https://doi.org/10.1145/3609510.3609822

[23] Stefan Monnier and Zhong Shao. 2003. Inlining as staged computation. *J. Funct. Program.* 13, 3 (may 2003), 647–676. https://doi.org/10.1017/S0956796802004616

[24] Cameron Moy, Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. 2021. Corpse reviver: sound and efficient gradual typing via contract verification. *Proc. ACM Program. Lang.* 5, POPL, Article 53 (jan 2021), 28 pages. https://doi.org/10.1145/3434334

[25] Max S. New, Daniel R. Licata, and Amal Ahmed. 2019. Gradual type theory. *Proc. ACM Program. Lang.* 3, POPL, Article 15 (jan 2019), 31 pages. https://doi.org/10.1145/3290328

[26] Bengt Nordström. 1988. Terminating General Recursion. *BIT* 28, 3 (1988), 605–619. https://doi.org/10.1007/BF01941137

[27] Enno Ohlebusch. 1998. Church-Rosser Theorems for Abstract Reduction Modulo an Equivalence Relation. In *RTA*. Springer-Verlag, Berlin, Heidelberg, 17–31.

[28] Liz Rice. 2023. *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security* (1st ed.). O'Reilly Media, USA.

[29] Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on ECOOP 2007: Object-Oriented Programming* (Berlin, Germany) *(ECOOP '07)*. Springer-Verlag, Berlin, Heidelberg, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2

[30] Michael Sperber and Peter Thiemann. 1997. Two for the price of one: composing partial evaluation and compilation. In *PLDI* (Las Vegas, Nevada, USA). Association for Computing Machinery, New York, NY, USA, 215–225. https://doi.org/10.1145/258915.258935

[31] Aaron Stump. 2016. *Verified Functional Programming in Agda.* Association for Computing Machinery and Morgan & Claypool, New York, USA.

[32] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. 2003. Dynamic Native Optimization of Interpreters. In *IVME* (San Diego, California) *(IVME '03)*. Association for Computing Machinery, New York, NY, USA, 50–57. https://doi.org/10.1145/858570.858576

[33] André Tavares, Benoit Boissinot, Fernando Pereira, and Fabrice Rastello. 2014. Parameterized Construction of Program Representations for Sparse Dataflow Analyses. In *Compiler Construction*, Albert Cohen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–39.

[34] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. 2020. Fast Packet Processing with EBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.* 53, 1, Article 16 (feb 2020), 36 pages. https://doi.org/10.1145/3371038

[35] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *CGO*. IEEE Press, Virtual Event, Republic of Korea, 254–265. https://doi.org/10.1109/CGO53902.2022.9741267

[36] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *CAV (Lecture Notes in Computer Science, Vol. 13966)*, Constantin Enea and Akash Lal (Eds.). Springer, Heidelberg, Germany, 226–251. https://doi.org/10.1007/978-3-031-37709-9_12

[37] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An Object Storage Model for the Truffle Language Implementation Framework. In *PPPJ* (Cracow, Poland). Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/2647508.2647517

[38] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. In *PLDI* (Barcelona, Spain). Association for Computing Machinery, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381