Towards Functional Verification of eBPF Programs

Dana Lu* Imperial College London
Boxuan Tang* Imperial College London
Michael Paper Imperial College London
Marios Kogias Imperial College London

ABSTRACT

eBPF is being used to implement increasingly critical pieces of system logic. eBPF's verifier raises the cost of adoption of the technology, as making programs pass the verifier can be very effortful. We observe that the guarantees provided by the verifier have only been used for the narrow objective of verifying these programs' safety, despite them also enabling the automatic verification of program functional correctness. We envision a framework allowing developers to easily specify and automatically verify their eBPF programs with very little extra cost compared to simply passing the verifier.

We showcase our implementation of DRACO, built on top of KLEE. DRACO allows developers to fully or partially specify eBPF programs, add verification-time assert statements, and reason about multiple eBPF programs interacting with each other and userspace, all at minimal additional cost to the developers. We use DRACO to either fully or partially verify the correctness of several real-world or experimental XDP programs.

CCS CONCEPTS

• **Software and its engineering** → **Functionality**;

KEYWORDS

Functional verification; eBPF; Symbolic execution

ACM Reference Format:
Dana Lu, Boxuan Tang, Michael Paper, and Marios Kogias. 2024. Towards Functional Verification of eBPF Programs. In *Workshop on eBPF and Kernel Extensions (eBPF '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3672197.3673435

1 INTRODUCTION

eBPF programs are being deployed for an increasing amount and diversity of use cases. Firewalls [1], congestion control algorithms [14], load balancers [16] and task scheduling policies [9, 17] are now being defined in eBPF. For all these scenarios, eBPF's strict verifier acts as a gatekeeper to prevent poorly written programs from harming the kernel. Convincing the verifier of the safety of a program is a time-consuming task, but eBPF's success relies on the observation that this task is not a waste of time. Rather, eBPF enables the faster deployment of more secure and efficient systems.

However, it is important to differentiate eBPF from its verification process. At its core, eBPF is just an ISA for a bytecode which could define arbitrary programs and can be used outside the kernel context, e.g. in userspace [6] or microcontrollers [11]. eBPF's kernel verifier splits the eBPF programs into two broad categories: those that it considers safe, and all other programs. All of the programs from the first category satisfy strong memory safety properties and are free of unbounded loops.

The kernel verifier, though, is not a panacea. Just because a program is safe to execute in kernel mode does not mean that it is functionally correct and cannot harm the system. For instance, a safe but ill-formed scheduling policy could drop some tasks, and a firewall could drop, by mistake, valid packets required and expected by an application.

Given eBPF's wide adoption by superscalars [18], especially in critical tasks such as firewalls and security analysis, the appearance of eBPF marketplaces [12] with unknown and potentially malicious eBPF programs, and the emerging LLM-based code generation [13], which can be used to write eBPF programs, we need a robust way to reason about the functional correctness of eBPF programs. Although standard software engineering methods, such as extensive testing and progressive deployment can partially play this role, we believe that eBPF as a new technology requires and can enable much better tooling specially tailored to its characteristics.

The main insight of this paper is that the set of eBPF programs that already successfully pass the in-kernel verifier are amenable to further automated analysis, which can guarantee their partial or full functional correctness. Such an analysis has the potential to provide stronger guarantees than unit testing at a lower development cost.

In this paper, we present DRACO [1], an extensible tool that targets eBPF programs that passed the in-kernel verifier to provide guarantees about the program itself, through a full or a partial specification, and its interaction with the rest of the system, i.e. other eBPF programs and userspace. Based on the previous insight, DRACO uses exhaustive symbolic execution to reason about eBPF programs that are guaranteed to terminate, given that the in-kernel verifier has already accepted the programs under analysis.

We implement DRACO by extending KLEE [2], a widely used symbolic execution engine, and as a first step use it to verify either fully or partially certain properties of various real-world and research XDP programs, such as Katran [16], hXDP FW [1], Fluvia [20] and CRAB [4].

2 BACKGROUND

Kernel Verifier: Because eBPF programs are executed in kernel mode, the kernel must ensure they will be efficiently executed and will not crash. To that end, a kernel component called the eBPF verifier is in charge of exploring all possible execution paths of eBPF programs loaded in the kernel. The verifier conservatively checks a set of general safety properties of eBPF programs, such as the absence of NULL pointer dereferencing and the absence of

*These authors made equal contributions.

[1]Draco was an Athenian legislator famous for his harsh laws.

eBPF '24, August 4–8, 2024, Sydney, NSW, Australia

ACM ISBN 979-8-4007-0712-4/24/08.
https://doi.org/10.1145/3672197.3673435

unbounded loops. To achieve this, the verifier depends on abstract interpretation[3] which tries to reason about variable lifetimes and memory accesses of the eBPF programs on top of having to traverse all potential execution paths.

Functional Correctness: However, even if a program passes the verifier, it does not mean that the program is correct. The program may have functional bugs, which the verifier is not built to detect. For example, incorrect XDP programs can lead to the loss of network reliability, and since XDP programs can be used for packet filtering and access control, incorrect programs can compromise network security. Even if an eBPF program is not buggy, it might not be configured correctly through its control plane userspace program or it might not fully cover the target behaviour. The latter is highly likely in the case of eBPF marketplaces [12] from which people can deploy eBPF programs they have not written themselves. Therefore, verifying the functional correctness of an XDP program and reasoning about its expected interactions with userspace and other eBPF programs is crucial.

Symbolic Execution: Apart from abstract interpretation used in the eBPF verifier, another method for program static analysis is symbolic execution. Symbolic execution automatically and systematically tries to explore all feasible paths in a program. The key idea behind it is the notion of symbolic variables - variables that can represent any value, or a set of values defined by constraints generated any time a symbolic variable affects a program branch. As the number of branches in a program increases, the number of paths increases exponentially - a phenomenon known as path explosion. Thus, large programs can require a lot of time and resources to exhaustively symbolically execute, if at all possible. Consequently, symbolic execution can be used as a tool for both extensive testing to find bugs - even if it does not explore all possible paths in a program - and as a tool for verification when exhaustive. The latter is the case we explore in DRACO, since passing the kernel verifier already ensures the lack of path explosion.

3 DRACO'S DESIGN

We design DRACO to be an extensible tool that can be used to verify different properties about either individual programs or program interactions for programs that run in the kernel. DRACO's core is a symbolic execution engine that can exhaustively explore all paths in a given eBPF program. On top of this engine, DRACO provides a set of components, each running a different analysis. Currently, DRACO targets XDP programs only but could be extended to support other types as well. In the rest of this section, we introduce the design of the current components and explain how developers can use and extend DRACO.

3.1 Verifying Individual Programs

To verify the functional correctness of an individual program, one has to first define what is the correct and expected behaviour of an eBPF program, namely to create a spec. Such specs can take different shapes and forms, be written in different languages, by different sets of developers, and provide different levels of coverage.

DRACO supports two types of specifications: external and integrated. External specifications are executable programs written in C/C++ or Rust that implement the same functionality with the target eBPF program either fully or partially. Such specifications can either be written by the same eBPF developers or developers that do not implement the eBPF functionality themselves. Integrated specifications take the form of assert statements throughout the eBPF program and, unlike the previous approach which is black-box, they can reason about the internals of an eBPF program. Specifications also include a *context*, a set of constraints of the incoming network packet and the state of the maps the eBPF program might access. Initial map constraints can specify key-value pairs in a map with specific values or their absence. Keys that are not specified will have both "absent" and "present" execution paths verified when looked up with the "present" path returning a symbolic value. To ensure the specifications can be written at minimal cost to the developers, they use a similar syntax to eBPF programs to reduce the barrier of entry to writing specifications.

Assertion	Description
ASSERT(bool)	*bool* holds (normal assert function)
ASSERT_RETURN(action)	Main function must return *action*
ASSERT_CONSTANT(addr, len)	The contents at `addr` for `len` bytes are the same when execution reaches the assertion as when the main function returns
ASSERT_END_EQ(addr, type, x)	The value at `addr` is `x` when the main function returns
ASSERT_IF_ACTION_EQ(action, addr, type, x)	Same as above but only when the return value of the main function is `action`
ASSERT_END_EQ_ADDR(addr_end, addr_now, len)	The contents at `addr_now` when execution reaches the assertion are the same as the contents at `addr_end` when the main function returns for `len` bytes

Table 1: Temporal assertions supported by DRACO

A full specification is expected to fully describe the expected behaviour of an eBPF program across all inputs, which in DRACO's case, which focuses on XDP programs, is the incoming packet. For example, in the case of network functions the full spec can be an executable version of the equivalent RFC [27]. A partial spec only focuses on parts of the expected functionality. For example, a partial spec for a firewall is to eliminate all ICMP traffic. In this case, the firewall can eliminate more types of traffic beyond ICMP. Finally, the integrated specs can be more fine-grained and targeted. For example, they can assert that certain paths in the code are incompatible, e.g. there can be no TCP processing if there is no TCP header in a packet, or that certain program variables have their expected values at different points of the execution.

External Specs: An external specification (partial or full) comprises the *program logic* and the *context*. A program is said to match a specification if given the *context*, all execution paths that are feasible in the eBPF program and the specification are equivalent, i.e. the final packet state, map state, and return value are the same. The difference between a full and a partial specification is that any path in the full spec should be equivalent to any path in the eBPF program and vice versa, while in the partial spec, the relationship is one way. Partial specs also allow for a more lenient definition of equivalence, to allow focus only on certain parts of the program end state, e.g. only the return code.

Integrated Specs: eBPF developers develop the integrated spec as part of the program at the same time as the program logic. Such

```
int xdp_program(struct xdp_md *ctx) {
  struct eth_hdr *eth = get_eth(ctx);
  ASSERT_CONSTANT(&eth->source, sizeof(eth->source));
  ASSERT_IF_ACTION_EQ(XDP_DROP, eth->eth_proto,
    __be16, BE_ETH_P_IPV6);
  if (eth->eth_proto == BE_ETH_P_IPV6) return XDP_DROP;
  return XDP_PASS;
}
```

Listing 1: Asserting that a program keeps the Ethernet source unchanged for all packets and all dropped packets are of IPv6 packets

specs take the form of `assert` statements. Beyond the usual assertions, though, that can be triggered on the spot, DRACO allows reasoning about the end state of an eBPF program based on its current state at specific execution points. Table 1 enumerates a selected list of supported assertions; the inequality assertions and assertions about map contents are omitted for brevity. Listing 1 shows an example using temporal assert statements to assert an unchanged Ethernet source and that all dropped packets are IPv6.

Multi-core programs: DRACO supports some analysis of multi-core programs. BPF maps accessed by multi-core programs can either be per core or shared between cores. Multi-core programs with per-core BPF map access are logically equivalent to single-core programs, so they can be verified using both external and integrated specifications. Multi-core programs with shared BPF maps can be verified against only integrated specifications. Each lookup to shared BPF maps always forks execution as another core could have updated the BPF map since the previous access. In one path, the key is assumed to be absent and in the other path, a new unconstrained symbolic value is returned.

3.2 Verifying Interactions Across Programs

DRACO goes beyond a single eBPF program and tries to reason about interactions between different eBPF programs, specifically targeting multiple eBPF programs attached to the XDP hook. Although such chaining encourages the development of reusable and modular programs that can be composed to build complex systems, an incorrect update to the state by one program can trigger an avalanche of erroneous execution paths if other programs rely on that state. Additionally, certain programs may need to execute in a specific order if one program relies on the function of another. For example, the order of applying a firewall or an ACL relative to a NAT is crucial. We focus on interactions that can be made through maps and packet data, leaving dependencies on the internal kernel state for future work.

A key insight in determining if the ordering of programs matters is that sequenced programs affect each other through shared state. Read after Write (RAW), Write after Write (WAW), and Write after Read (WAR) dependencies are the ones that matter for DRACO. With this in mind, we define the *ReadSet(P)* of a program P as the set of externally visible state locations that the program reads from, for each execution path. Similarly, the *WriteSet(P)*, of a program P is defined to be the set of externally visible state locations the program writes to, for each execution path. Using this definition, the read set of an XDP program is the set of bytes in the packet from which the program reads, *PacketRead(P)*, in addition to the set of keys looked up from shared maps, *MapRead(P)*:

$$ReadSet(P) = PacketRead(P) \cup MapRead(P)$$

The definition for the write set is similar, using the set of bytes in the packet the program writes to, *PacketWrite(P)*, and the set of key-value pairs from shared maps the program updates, *MapWrite(P)*:

$$WriteSet(P) = PacketWrite(P) \cup MapWrite(P)$$

With the *ReadSet(P)* and *WriteSet(P)* of a program P defined, we now introduce a method to check interactions between different programs. Consider two programs, **P1** and **P2**. Given their respective *ReadSet*s and *WriteSet*s, the *Overlap* of the programs is defined as:

$$\begin{aligned} Overlap(P1, P2) =& (ReadSet(P1) \cap WriteSet(P2)) \\ & \cup (WriteSet(P1) \cap ReadSet(P2)) \\ & \cup (WriteSet(P1) \cap WriteSet(P2)) \end{aligned}$$

If two programs have no overlap between their *ReadSet* and *WriteSet*, i.e. $Overlap(P1, P2) = \emptyset$, it is safe to run the programs in any order. This commutative property of the programs arises because the interactions of one program with shared state do not affect the interactions the other program has. Thus, the externally observable state will be identical no matter the order the programs run. However, if the *Overlap* of two programs contains at least one element, the order of execution becomes important since one program's updates can be observed by the other program.

3.3 Verifying Interactions with Userspace

The other type of interaction eBPF programs have is with userspace programs through maps. Userspace programs usually play the role of the control plane for the eBPF-based data plane. Hence, DRACO aims to aid with control plane configurations and updates by analysing the interactions with userspace and the eBPF program. Note that DRACO does not try to verify the behaviour of the userspace program, but only provides hints as to how a control program should be written.

Correlation between Multiple Maps: Analysing the interactions between different BPF maps in an eBPF program provides valuable insights into how the maps influence each other, allowing developers of userspace programs to update maps in a more sensible way. Figure 2 shows an example from an eBPF program [1] of a map correlation where the value returned from a lookup on a map can be used as the key to a redirect on another map. Such map correlation indicates that userspace programs need to update maps in a particular order with meaningful values.

A value is said to be *derived* from a map helper function call if it uses the return value in some way. By tracking all such *derived* values, if an argument to a map helper function is *derived*, these maps are identified as being correlated. DRACO detects and reports these correlations so that control plane developers can either enforce or take into consideration this correlation when updating those maps.

Maps and Control Flow: The control flow of eBPF programs is further complexified as they often interact with BPF maps, similar to the importance of table contents in match-action pipelines [15, 22]. So, it is essential for these maps to be properly configured to avoid unexpected execution paths. This DRACO component works in conjunction with the assert-based specification, as it depends on

```
...
flow_leaf = bpf_map_lookup_elem(&flow_ctx_table, &flow_key);
    if (flow_leaf)
        return bpf_redirect_map(&tx_port, flow_leaf->out_port, 0);
...
```

Listing 2: Example of correlation between maps

asserts to identify illegal paths and the conditions on the maps under which they are executed.

Using the same definition for a value being *derived* from a map helper function call, a map affects the control flow if the condition in a conditional branch is *derived* from a map helper function call. By tracking all such branches, these branches as well as the original map helper function called can be reported upon assertion failures. Additionally, the outcome of the branch condition of each *derived* branch is reported as well - together these form conditions on the map under which the erroneous path is taken. Developers can then use these insights to correctly configure the maps to prevent the execution of these incorrect paths [21] or find out what map contents would break the intended behaviour of the program.

4 IMPLEMENTATION

DRACO depends on exhaustive symbolic execution. To do so, it builds on top of KLEE [2], which is a symbolic execution engine operating at the LLVM IR level, hence DRACO currently assumes the availability of the source code. DRACO uses the symbolic models for the `libbpf` functions also used in PIX [10], as defined in the `ebpf-se` [5] opensource project.

4.1 Verifying Individual Programs

External Specification: DRACO uses a driver program that combines the executable specification with the eBPF program and uses KLEE to ensure their functional equivalence. Specifically, it uses a symbolic packet and asserts that `spec(packet) == prog(packet)` and that the packet is modified in the same way in both cases. KLEE will ensure that the assertion holds for all possible paths. For stateful programs, i.e. ones that manipulate maps, DRACO ensures that the eBPF program and the specification operate on the same initial map state and asserts that the final state of the maps is identical.

The case of partial specification follows a similar logic. Given the one-way equivalence, i.e. every valid path in the specification should behave the same in the eBPF program but not vice versa, DRACO only checks equivalence for the valid specification paths. To do so, it first runs the specification to extract the set of symbolic constraints for each valid path and only for those paths it runs the actual program and asserts the final state. For the case of partial specification, the definition of equivalence can vary. DRACO also allows for partial equivalence, e.g. the partial spec can ignore the final contents of the maps and only focus on the return value of the program and/or the packet contents.

Listing 3 partially shows the driver program DRACO uses to verify the correctness of an eBPF program based on a full specification. Initially, it creates a symbolic packet and its copy, one for the spec and one for the actual program, both with the same initial symbolic constraints. Next, it configures the initial map values and runs the program. Afterwards, it resets the maps to their initial values and runs the specification. It then determines if the current execution

```
void functional_verify(xdp_func prog, xdp_func spec, ...) {

    void *pkt1 = create_symbolic_packet(...);
    set_initial_packet_constraints(pkt1);
    void *pkt2 = copy_packet(pkt1);

    set_initial_map_constraints();

    // Run the eBPF program
    struct xdp_state prog_state = get_end_state(prog, ctx);

    reset_symbolic_maps();
    set_initial_map_constraints();

    // Run the executable specification
    struct xdp_state spec_state = get_end_state(spec, ctx_copy);

    // Check if the same assumptions about map state were taken
    if(different_map_assumptions()) return;

    // Assert equivalence
    assert(xdp_state_equal(&prog_state, &spec_state));
}
```

Listing 3: Driver program for verifying external full specifications

path should be verified based on whether the same assumptions were taken about the initial contents of the map, i.e. if a key was assumed to be present during a BPF map lookup in the program it must also be assumed to be present when executing the specification. Finally, it asserts that the final states, i.e. return value, packet contents, and map contents, are identical.

Integrated Specification: `asserts` added by the eBPF developers in the source code can be split into two groups: the ones that trigger immediately when the symbolic execution engine goes through them and the ones that trigger at the end of the execution. For the latter, DRACO leverages a set of queues which store the relevant data at the point when the assertions are made, which are checked in the relevant manners when the main XDP function returns.

4.2 Verifying Program Interactions

Unlike when verifying individual programs, the verification of interactions between programs and the system requires additional information to be generated when the program is symbolically executed. DRACO extends KLEE to enable dynamic determination of exactly when a program accesses maps or packet data, and the results of branching decisions leading to different points in the program for its interaction analysis.

Interactions Across Programs: DRACO extends the KLEE interpreter to obtain the memory object storing the packet data, and instruments all `load` and `store` instructions to determine if they access the same memory object that stores the packet data, thus creating the *PacketRead* and *PacketWrite* sets. To calculate the *MapRead* and *MapWrite* sets of a program, a similar method is used to track all memory objects that store maps. All reads and writes to these objects are captured, creating the *MapRead* and *MapWrite* sets. Note that even if BPF helper functions are not used to access these maps, for example by using a pointer dereference, accesses to maps are still captured, as there must be a `load` or `store` instruction to read from or write to memory where maps are stored, respectively.

```
vip_info = bpf_map_lookup_elem(&vip_map, &vip);
...
vip_info->vip_num = 7;
```

Listing 4: Example of update without using map helper functions

Program	LOC	Type	Context	Spec	Paths	Time
hXDP FW	686	Full	2	27	64	6.93s
hXDP FW	686	Full	12	18	4	3.45s
Fluvia	156	Partial	0	4	23	23.35s
Katran	4244	Partial	0	17	10	71.24s
CRAB	365	Assert	16	20	5	1.90s

Table 2: Evaluation of different methods to functionally verify individual programs by counting lines of code in each specification's *context* and *program logic*/assertions and measuring the time taken for verification

To ensure that the analysis covers meaningful paths that exist in both programs, DRACO uses a third driver program that runs the two main XDP functions, with a separator to indicate their boundary. During the symbolic execution of the first program, DRACO collects the aforementioned read and write sets. After reaching the separator, KLEE switches to stop adding elements to the *ReadSet* and *WriteSet*, but instead to check for presence in those sets. This method determines the *Overlap* of two programs for any viable path instead of an over-approximation across all possible paths.

Interaction with Userspace: Contents of BPF maps are accessed through calls to map helper functions, thus DRACO also extends KLEE to intercept all such calls. Each call is tracked in a data structure which maps call instructions to a set of values that use the return value, unique for each execution path. Instructions are added to the set for a call if at least one of their operands is present in the set.

Using this common data structure, correlations between maps and maps that affect the control flow can be identified. Arguments to map helper functions that are present in one of the sets indicate a dependency between maps, and the original call to the map helper function that affects a branch can be identified. As information is maintained per execution path, when an assertion failure occurs, only the dependent branches for the error path can be reported.

Since the userspace program may not be available, maps can contain fully symbolic contents to ensure all possible behaviours are captured. If it is known how the userspace will populate these maps, verification helper functions can be used to constrain map contents, to avoid symbolic execution of unfeasible paths. Currently, the constraints are limited to specifying a map contains or doesn't contain a key, which can be symbolic.

5 EVALUATION

We use DRACO to specify and analyze the set of XDP programs included in the `ebpf-se` [5] tool. Through the evaluation, we aim to answer the following questions:

- How easy is it to write full or partial specs for XDP programs?
- How long will it take to evaluate the equivalence of the specs and perform the interaction analysis?

	Execution Time (s)		Avg Memory (MiB)	
Program	Original	Extended	Original	Extended
hXDP FW	0.15	0.16	30.45	30.88
Fluvia	0.33	0.43	30.43	30.76
CRAB	0.17	0.19	32.26	32.72
Katran	77.84	100.42	36.30	37.01

Table 3: Benchmark of performance for extended KLEE vs original

Specification Equivalence: Table 2 summarises the results of our evaluation. hXDP FW is a firewall program that is verified fully. The second full specification verification places more constraints on the initial state resulting a simpler specification with fewer execution paths and a faster verification time. Fluvia is an IPFIX exporter and its partial specification verifies that regardless of context, it always returns `XDP_DROP` with no modifications to the packet state. Meta's Katran load balancer is partially verified that all fragmented IPv4 packets are dropped and all ICMP Echo packets are transmitted with possible modifications. Finally, we use an integrated specification for CRAB, which behaves similarly to an L4 load balancer but only handles SYN packets and adds a custom redirection header in those packets. DRACO conducts various sanity checks to ensure a redirection options header is correctly prepended to the packet for TCP SYN packets, the MAC address is correctly updated and various values remain constant after certain points during the code execution. It is important to keep in mind that the performance of partial specification and assert statements methods depend heavily on the complexity of the functionalities being verified, which is represented as *Spec* - referring to the lines of code of partial specification or the number of assert statements.

Comparing the lines of code against the lines of code used to write the specification or assertions, it is evident that after writing an eBPF program it takes significantly less work to write some form of specification to verify certain program functionality. In all cases, we are also able to complete the functional verification in a reasonably fast time, especially if map state equivalence is not required, concluding that DRACO could also run as part of the CI/CD pipeline.

Interaction Analysis: To evaluate the analysis of interactions between a program and the system, we measure the time taken and additional memory DRACO takes to generate all of the data required for all of the analysis between a program and a userspace program, not considering computing the *Overlap*. Since the generation of the data happens during symbolic execution, we compare the performance of the original unmodified version of KLEE with the extended KLEE on these programs. The results are shown in Table 3.

Compared with the base performance of the original, unmodified version of KLEE, the extended version of KLEE runs on average 9.5% slower. This is due to the extra checks performed to generate the data. An additional 0.40 MiB of memory is used on average to store all information required for generating the data for analysis.

6 RELATED WORK

eBPF Verifier: Since the verifier determines if programs are safe to run or not, bugs in the verifier can be exploited by attackers, compromising the security of the system. Agni [25] proves the soundness of the verifier's value range analysis by developing soundness specifications, and found bugs where the analysis was not sound. BVF [23] finds correctness bugs in the verifier through the use of a test oracle.

P4 Programs: p4v [15] is a tool for statically verifying the correctness of P4 programs through the use of user-written annotations. It introduces the concept of control plane interfaces which describe the set of possible behaviours of the control plane. The interfaces are written in GCL and must be written by hand.

Vera [22] is a tool that can automatically verify snapshots of P4 programs using symbolic execution. Through the use of symbolic entries in the match-action tables, Vera is also able to automatically verify multiple snapshots at once. Users can also write program-specific properties in NetCTL, an extension of Computation Tree Logic.

ASSERT-P4 [8] verifies P4 behaviour by making temporal assertions about the future behaviour of a program, then verifying these assertions using symbolic execution in KLEE. It utilises a Python script to convert P4 files into C while adding assert statements and variables (to store data to be used by the assert statements), before executing the translated file in KLEE to check the assertions. DRACO's design and implementation of using assert statements to verify eBPF programs is inspired by ASSERT-P4. However, the implementation differs as eBPF functions are already written in C, so the dependency of using an additional scripting language is avoided and the implementation is inlined.

Network Functions: VigNAT [27], later extended to Vigor [26], is used for verification of network functions written in C using KLEE. Programs are split into stateful and stateless parts, which are verified individually and then stitched together. While the stateless code can be verified easily using symbolic execution, the stateful code must be verified by hand by verification experts using separation logic. Vigor also introduces the "pay-as-you-go" ideology allowing developers to write partial specifications for network functions. The way these partial specifications are supported influenced our design choices in supporting the verification of partial specifications of eBPF programs.

Other approaches to eBPF verification: Serval [19] is a framework for creating automated verifiers for systems software, including the BPF ISA. It relies on symbolic evaluation to verify programs against a specification written in Rosette [24], an extension of the solver-aided programming language Racket [7]. However, the prerequisite knowledge of this language presents a barrier to developers writing specifications for their programs.

7 FUTURE WORK AND OPEN QUESTIONS

We identify several directions for future work. Some of them are more engineering-heavy than others. For example, symbolic modelling for eBPF maps currently only covers certain BPF map types. Adding more map types, will allow DRACO to be used with a wider range of XDP programs. Similarly, DRACO currently only supports XDP programs. Extending to more eBPF program types beyond XDP will require supporting more helper functions and implementing symbolic models for kernel state that can become more challenging, since it can affect the interaction between different eBPF programs.

In terms of its model of use, currently DRACO assumes the availability of the source code for the programs it analyzes, instead of requiring just the eBPF bytecode. This limitation stems from the fact that KLEE works at the LLVM IR level. We are currently working on implementing a lifter from eBPF byte code to LLVM IR that will allow DRACO to be applicable even in cases without source code access, which might be the model in future eBPF marketplaces.

The main open question, though, is whether the main insight based on which we build DRACO will continue to hold. Given the active development of the in-kernel verifier and the alternative proposals by the research community, we need to identify whether exhaustive symbolic execution will always apply to all eBPF programs, hence allowing their functional verification based on that approach.

8 CONCLUSION

This paper introduces DRACO, a tool for verifying the functional correctness of eBPF programs and aiding the developer in understanding how unknown eBPF programs behave. This tool builds upon the symbolic execution engine KLEE [2], leveraging the insight that passing the in-kernel verifier enables exhaustive symbolic execution. We use DRACO to analyze a series of XDP programs showing we can do so with minimal development effort and execution overhead.

ACKNOWLEDGEMENTS

We would like to thank Rishabh Iyer and Nikola Bojanic for providing us with an understanding of eBPF-SE and how it works. We thank the anonymous reviewers for their helpful feedback. This work does not raise any ethical concerns.

REFERENCES

[1] Axbyrd. 2020. FW source code respository. (2020). https://github.com/axbryd/hXDP-Artifacts
[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, USA, 209–224.
[3] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.
[4] EPFL Data Center Systems Lab. 2021. CRAB source code respository. (2021). https://github.com/epfl-dcsl/crab
[5] EPFL Dependable Systems Laboratory. 2024. eBPF-SE source code respository. (2024). https://github.com/dslab-epfl/ebpf-se
[6] Eunomia. 2023. bpftime Userspace eBPF Runtime. https://github.com/eunomia-bpf/bpftime. (2023).
[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A programmable programming language. *Commun. ACM* 61, 3 (feb 2018), 62–71. https://doi.org/10.1145/3127323
[8] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *Proceedings of the Symposium on SDN Research (SOSR '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3185467.3185499
[9] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*

(SOSP) (SOSP '21). Association for Computing Machinery, New York, NY, USA, 588–604. https://doi.org/10.1145/3477132.3483542

[10] Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance Interfaces for Network Functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 567–584. https://www.usenix.org/conference/nsdi22/presentation/iyer

[11] Zandberg K, Baccelli E, Yuan S, Besson F, and Talpin JP. 2022. Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers. In *23rd ACM/IFIP International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada*. ACM, New York, NY, USA, 161–173. https://doi.org/10.1145/3528535.3565242

[12] L3AF. 2024. L3AF Marketplace. https://l3af.io/. (2024).

[13] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[14] Linux Foundation. 2024. net/ipv4/bpf_tcp_ca.c in the Linux Kernel Source Tree. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/bpf_tcp_ca.c?h=v6.9. (2024).

[15] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. p4v: practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 490–503. https://doi.org/10.1145/3230543.3230582

[16] Meta. 2024. Katran source code respository. (2024). https://github.com/facebookincubator/katran

[17] Meta and Google. 2024. sched_ext: BPF extensible scheduler class. https://github.com/sched-ext/scx. (2024).

[18] Bill Mulligan and Daniel Borkman. 2023. The Silent Platform Revolution: How eBPF Is Fundamentally Transforming Cloud-Native Platforms. https://www.infoq.com/articles/ebpf-cloud-native-platforms/. (2023).

[19] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 225–242. https://doi.org/10.1145/3341301.3359641

[20] NTT Communications. 2024. Fluvia source code respository. (2024). https://github.com/nttcom/fluvia/

[21] p4language. 2020. p4-constraints source code respository. (2020). https://github.com/p4lang/p4-constraints

[22] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 518–532. https://doi.org/10.1145/3230543.3230548

[23] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. 2024. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 689–703. https://doi.org/10.1145/3627703.3629562

[24] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

[25] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 226–251.

[26] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://doi.org/10.1145/3341301.3359647

[27] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 141–154. https://doi.org/10.1145/3098822.3098833