Towards Functional Verification of eBPF Programs

Marios Kogias

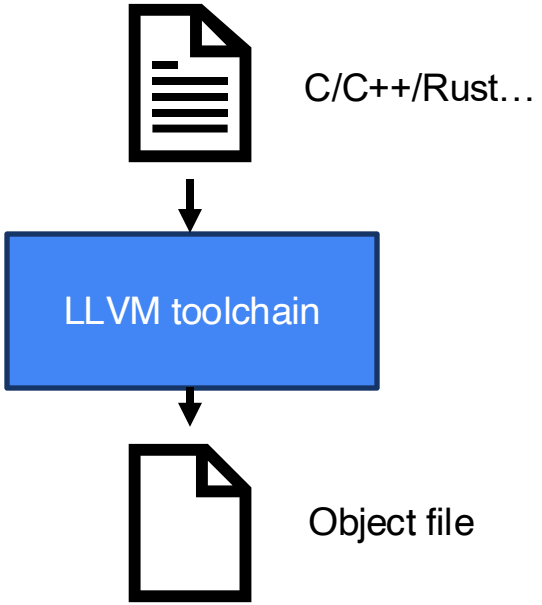
IMPERIAL

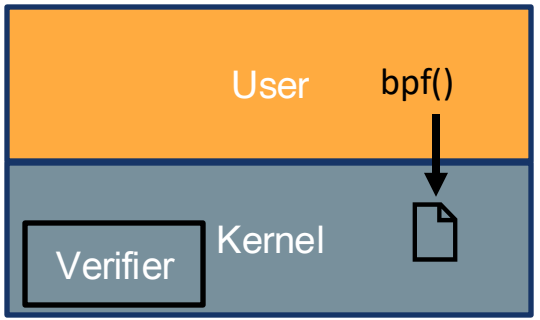
eBPF

- eBPF (extended Berkeley Packet Filter)
- A lightweight virtual machine inside the Linux kernel
- Allows you to run programs in a "sandbox" in certain locations in the kernel
- You can safely and efficiently extend the capabilities of the kernel without having to change the kernel.
- Adopted by the industry and used extensively in projects we use every day
  - Various usecases: monitoring, tracing, scheduling, packet processing and more…
- Probably the "hottest thing" happening in the kernel community at the moment.

eBPF Program Life Cycle

- Write your eBPF logic in your favourite programing language
- Compile through the LLVM tool chain
- Load the program inside the kernel using the bpf() system call
- An **in-kernel verifier** checks the code before execution to ensure: no invalid memory accesses, bounded execution
- Attach the program to a **hook** inside the kernel and then the execution follows an event-driven paradigm

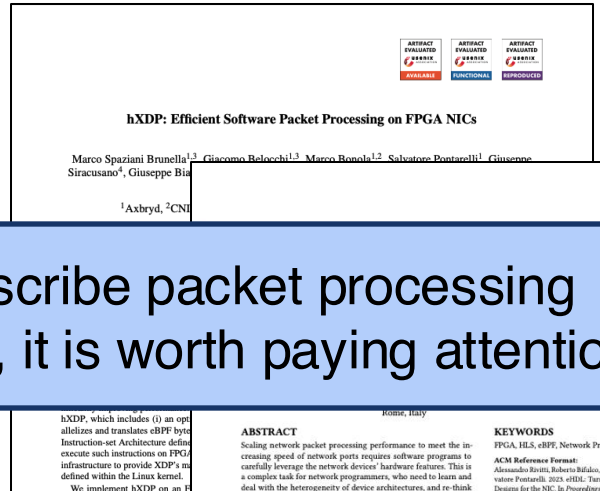




eBPF for Networking

- Different hooks inside the kernel:
  - **XDP**: right after packets are received by the NIC and right before they enter the kernel network stack. Available only for Rx
  - **TC**: traffic shaping layer before the TCP layer
  - **Socket**: after the TCP stack and before the application

- Hardware acceleration:
  - ASICs running eBPF
  - eBPF softcores implemented in FPGAs
  - Translating eBPF to FPGA circuits



eBPF is becoming a popular way to describe packet processing pipelines across different platforms. So, it is worth paying attention to it

eBPF & In-kernel Verifier

- Let's differentiate between eBPF bytecode and its verification process

eBPF bytecode:

- 10+1 registers
- ALU and memory operations
- Access to maps
- Access to helper functions
- Nothing special compared to other bytecodes, e.g. WASM or JVM

In-kernel verifier:

- Ensures safety:
  - No loops, no out-of-bounds memory access, program termination
- Uses abstract interpretation to track state across execution paths
- Does not cover functional correctness
  - eBPF programs can still have logical bugs but are guaranteed not to crash the kernel

😕 Users follow a trial-and-error iterative approach till the verifier is happy…

🙂 eBPF has indirectly enforced the use of formal methods to users without the equivalent background

Key Insight

eBPF programs that satisfy the in-kernel verifier have certain properties that make them amenable to **further static analysis**

Question: How can we use the above insight to improve the development and deployment experience of eBPF programs?

Outline

- DRACO: A tool on exhaustive symbolic execution for eBPF analysis

- Usecases:
  - Verifying functional correctness of individual programs
  - Identifying and securing program interactions
  - Enforcing stricter constrains than the in-kernel verifier for unprivileged eBPF

Draco Core Mechanism

Let's enumerate and explore **all execution paths** for any potential input. This is a tractable problem given the verifier constraints.

What is Draco?

- Symbolic execution engine based on KLEE
- Symbolic models for helpers and data structures, e.g. a packet

- An extensible set of analyses around that infrastructure to reason about the functionality of the eBPF program

Symbolic Execution 101

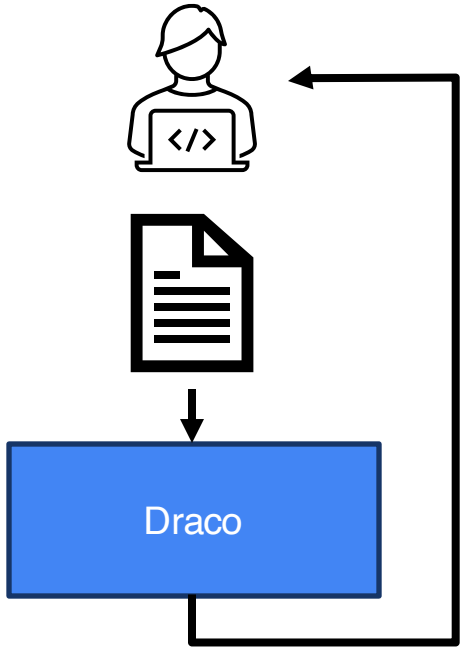
- Programs run on **symbolic** inputs
- **Branches** add constraints to the symbolic values and fork the execution to continue the exploration
- When branching an **SMT** solver determines whether a path is feasible given the constraints

- Usually not exhaustive
  - Used for bug finding
- Can suffer from path explosion
- Not the case here given the verifier constraints

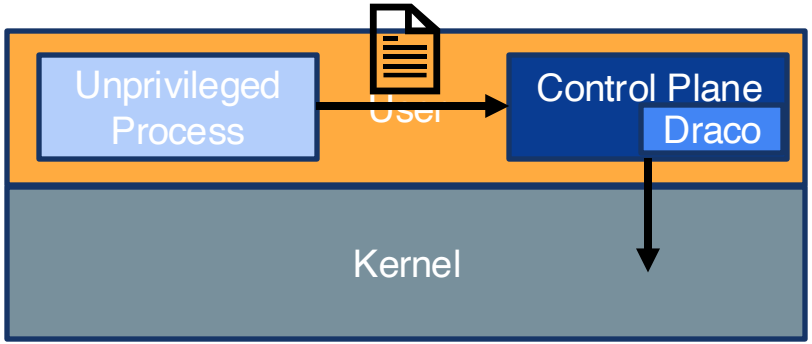
```
int my_function(int a, int b ) {
 if ( a > b ) {
  return 1                          (a > b)
 } else {                           !(a > b)
  if ( a < b ) {
   return -1                        !(a > b) && (a < b)
  } else {
   return 0                         !(a > b) && !(a < b)
  }
 }
}
```

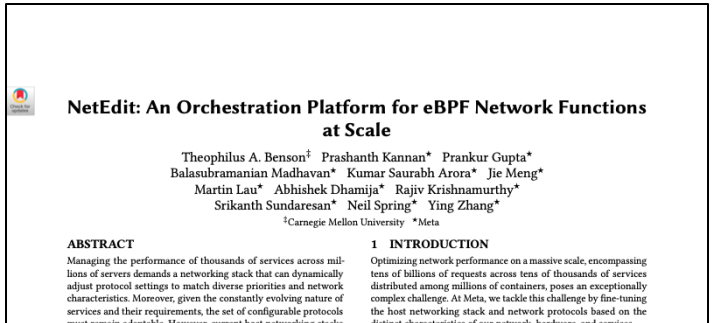
Draco Big Picture

- Independent tool as part of the development/deployment process



- Part of a generic eBPF control plane





NetEdit: An Orchestration Platform for eBPF Network Functions at Scale

Theophilus A. Benson‡ Prashanth Kannan* Prankur Gupta*
Balasubramanian Madhavan* Kumar Saurabh Arora* Jie Meng*
Martin Lau* Abhishek Dhamija* Rajiv Krishnamurthy*
Srikanth Sundaresan* Neil Spring* Ying Zhang*
‡Carnegie Mellon University *Meta

ABSTRACT

Managing the performance of thousands of services across millions of servers demands a networking stack that can dynamically adjust protocol settings to match diverse priorities and network characteristics. Moreover, given the constantly evolving nature of services and their requirements, the set of configurable protocols

1 INTRODUCTION

Optimizing network performance on a massive scale, encompassing tens of billions of requests across tens of thousands of services distributed among millions of containers, poses an exceptionally complex challenge. At Meta, we tackle this challenge by fine-tuning the host networking stack and network protocols based on the

Usecase 1: Functional Correctness

The verifier does not guarantee functional correctness!

Why do I care?

- eBPF is used in critical infrastructure, e.g. firewalls
- eBPF market places start appearing
- AI code generation is becoming more popular

There is a need for verifying the behaviour of eBPF programs!

How do I specify the correct behaviour? I

External specification

- Executable program written in C/C++/Rust that implements the same functionality of the eBPF program either **fully** or **partially**
  - Use the fact that KLEE operates at the LLVM IR level
- Written by the same or different developer than the one implementing the eBPF program

For every execution path check that the (1) return value, (2) changes to the network packet and (3) changes to the BPF maps are equal

Using Driver Program for External Specification

```
int main(int argc, char** argv){
  struct pkt *packet = create_packet(sizeof(struct pkt));
  packet->ether.h_proto = BE_ETH_P_IP;
  packet->ipv4.protocol = IPPROTO_TCP;
  struct xdp_md *ctx = create_ctx(packet, sizeof(struct pkt), 0);
  BPF_MAP_INIT(&tx_port, "tx_devices_map", "", "tx_device");
  BPF_MAP_INIT(&flow_ctx_table, "flowtable", "pkt.flow", "output_port");
  functional_verify(xdp_fw_prog, xdp_fw_spec, ctx, sizeof(struct pkt), 0);
}
```

How do I specify the correct behaviour? II

Integrated specification

- Temporal **assertions** inserted throughout the eBPF program
- Draco implements a library to help developers write integrated specifications
- Written by the same developer implementing the eBPF program
- Assertions checked either according to the program control flow or deferred, i.e. when the program returns

Integrated Specification Examples

```
ethernet = data ;
BPF_ASSERT_CONSTANT(ethernet, sizeof(*ethernet));
```

BPF_ASSERT_CONSTANT asserts a memory location remains constant

```
BPF_ASSERT_IF_ACTION_THEN_NEQ(XDP_DROP, &(ip->protocol), __u8, IPPROTO_TCP);

nh_off +=sizeof(*ip);
if (data + nh_off  > data_end)
    goto EOP;

if(ip->protocol != IPPROTO_TCP){
        goto EOP;
}

BPF_ASSERT_RETURN(XDP_TX);

int key = ip->saddr;
int value = 1;
bpf_map_update_elem(&example_map, &key, &value, 0);

BPF_RETURN(XDP_TX);

EOP:
    BPF_RETURN(XDP_DROP);
```

BPF_ASSERT_IF_ACTION_THEN_EQ asserts that if an XDP action is returned the given memory location must not be equal to the given value

BPF_ASSERT_RETURN asserts that the given XDP action must be returned

Integrated Specification

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

Evaluation

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

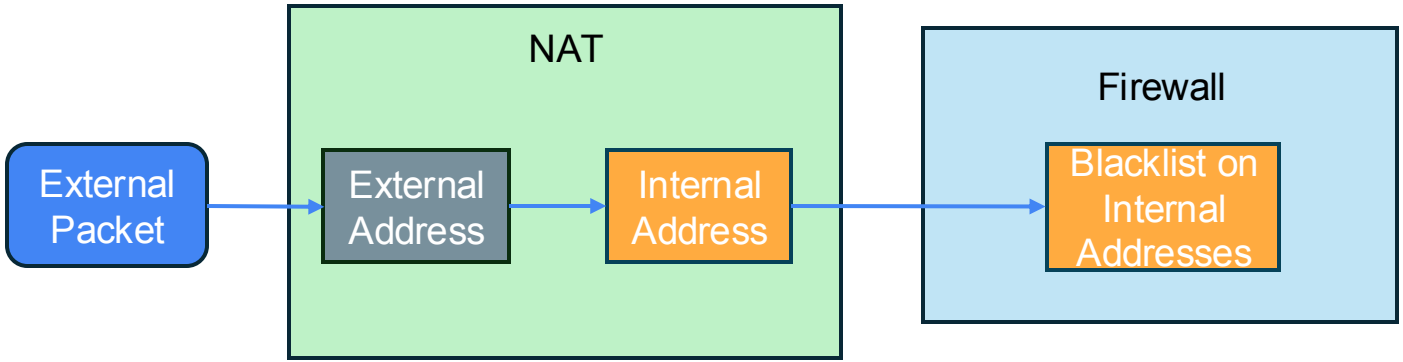
Usecase 2: Verifying Program Interactions

- Verifying the correctness of a single eBPF program might not be enough
- eBPF programs interact with:
  - Other eBPF programs as part of a chain
  - Userspace application through eBPF maps

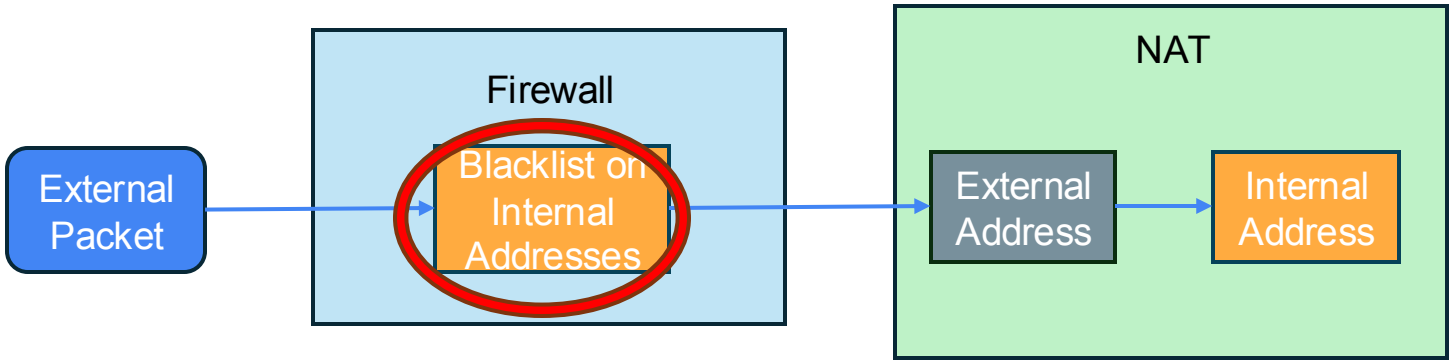
- **Why do I care?**
  - Identify eBPF programs that interfere, hence their ordering matters
  - Identify map dependencies to avoid faulty control plane updates
  - Identify map constraints to avoid faulty paths

Draco can also help reason about the eBPF program interactions

Ordering Example



Ordering Example



Draco will detect that the two programs have a RAW dependency

Dependent Maps Example

```
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);
```

- Access to the redirect map depends on the flow_ctx_table map
- The control plane should update them together and in the correct order

Draco will detect that the two map accesses are dependent

Map Content Constraints

```
flow_leaf = bpf_map_lookup_elem(&flow_ctx_table, &flow_key);
if (flow_leaf)
  BPF_ASSERT(flow_leaf->out_port < 5);
```

- The data plane code can encode map constraints
- Wrong map contents can lead control flow to faulty paths indicated by assertions

Draco can identify map accesses and their constraints that led to failed executions

Mechanisms & Implementation

- Extend KLEE to track the **read and write set** for packet memory addresses and maps for each execution path
- Extend KLEE to track **correlated map accesses**, i.e. the key used to access a map is derived from a previous map access
- Extend KLEE to track **branches dependent on maps**
- Introduce a **program separator** (__separate()) to avoid over-approximating the ordering analysis

Usecase 3: Enforcing stricter constraints

- Root or CAP_BPF required to load an eBPF program
- Very coarse-grained access control (binary)

- Goal: Use Draco as part of a privileged control plane to enforce policies on what eBPF programs can and cannot do

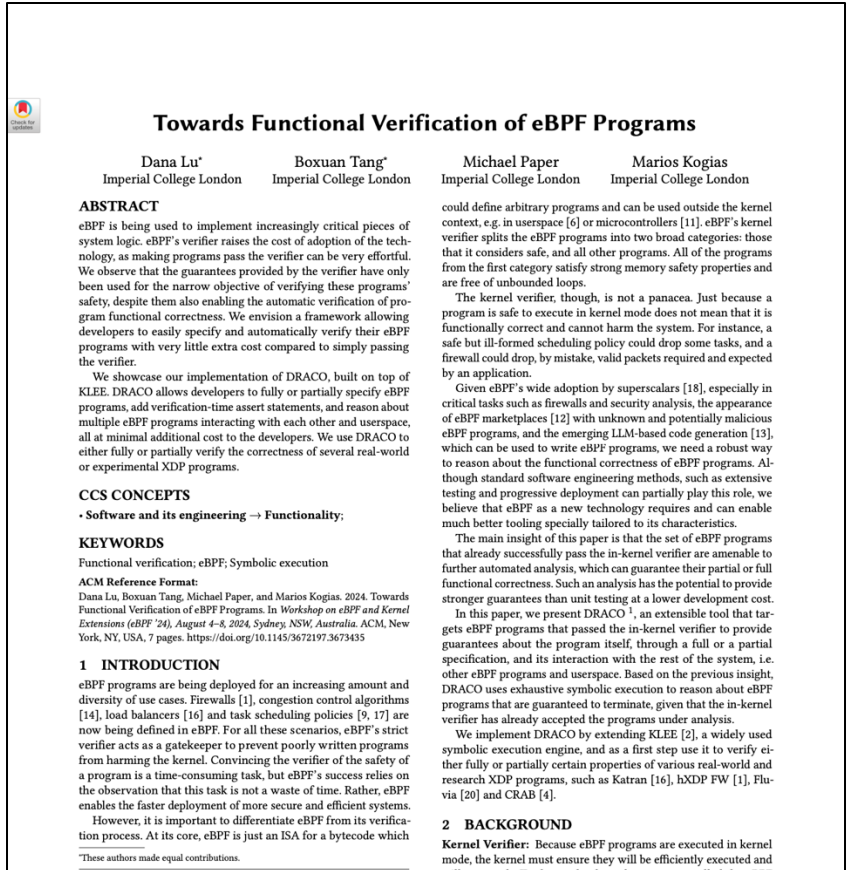
- Example policies:
  - Restrict access to maps/helper functions
  - Prevent/only allow access to certain types of packets
  - Prevent/only allow RW access to certain parts of the packet
  - Prevent/only allow access certain actions, e.g. prevent dropping packets
  - Further constraint eBPF program size

Draco can enable eBPF multi-tenancy and an **eBPF-as-a-Service model**

Current Status

- Initial PoC presented at the eBPF workshop at SIGCOMM 2024 covering the first two usecases
- Currently working on the 3rd usecase and the control plane integration
- Initial opensource version

https://github.com/draco-verifier



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

*These authors made equal contributions.

Takeaways

- eBPF is gaining popularity in writing network functions
- The in-kernel verifier forces developers to write verifiable code
- Draco leverages this insight to reason about eBPF programs using exhaustive symbolic execution and an extensible set of analysis

Thank you

m.kogias@imperial.ac.uk
https://marioskogias.github.io/