Equivalence and its applications to network verification

Dragos Dumitrescu Radu Stoenescu Matei Popovici Lorina Negreanu
Costin Raiciu

University Politehnica of Bucharest

1 INTRODUCTION

Misconfigured or faulty networks ground airplanes, stranding thousands of passengers and render online services inaccessible for hours on end. Network verification promises to detect such rare yet devastating problems by ensuring that networks always follow their operator's stated policy (a correctness specification).

Despite the plethora of recent tools for verifying networks ([2, 5, 8, 6, 3, 1, 4]), the uptake to network verification in production settings is still limited: providing network correctness specifications is cumbersome and usually requires understanding both network policies and transposing them into complex formal languages.

In many cases, network correctness properties can be implicitly derived by equivalence to other networks. Examples include cloud deployments where high-level network abstractions are *compiled* into complex pipelines of packet processing functions and *baselining* operations whereby devops keep previous functionality unchanged in the context of ever-changing network configuration.

Our goal is to verify network correctness in the absence of a classic form of specification. A well-known undecidable problem, we tackle equivalence by using symbolic execution; this technique was initially employed in programming language verification, but has scalability issues due to memory access and while loops. The specifics of network dataplanes - no loops and very little to no state make our approach ripe for scalably deciding equivalent properties.

In Figure 1 we show two code fragments that model a router with three FIB entries; this example is taken and slightly adapted from Symnet - the code is written in the SEFL language[8]. The first program is a simple sequence

of If/Else statements which forward the packet to the correct output port; there is one If per FIB entry. The second program is optimized for symbolic execution: it does not use If instructions at all, first *Forking* the packet (i.e. creating clones of it) and then using the *Constrain* instruction for each clone to restrict the packets that may leave on each port. *Constrain* drops all packets that do not match the constraint and has no effect on packets that do.

The two programs are meant to be equivalent, which means they must process packets in exactly the same way. Informally, this means that injecting *any* packet into equivalent input ports of the two programs (e.g. *in*) will result in both dropping the packet, or both emitting the same packet(s) on equivalent output ports.

Our goal is to automatically and scalably decide if two programs expressed in SEFL are equivalent. Proving equivalence can be done manually by using an iterative specification and refinement process, but this requires significant input from the programmer and has huge costs. Our approach is fully automated, and uses symbolic execution to check equivalence.

For the basic router program, symbolic execution outputs two failed paths (shown in red) and three successful paths (Fig.1). Two successful paths (black and blue) leave the router on interface *if0*, and one on *if1* (green). By taking the union of constraints of the black and blue paths we know exactly which packets can leave *if0*, and by examining the constraints on the green path we know which packets will leave on *if1*. In the optimized program, there are only two successful paths: one on *if0* and one on *if1*.

To decide if the two programs are equivalent, we devise EQ, an algorithm that compares the sets of paths exiting on all ports in both programs and checks if they are equivalent. We implement EQ in the *sdiff* tool and provide it as an extension to Symnet [8].

2 NETWORK PROGRAM EQUIVALENCE

We define behavioral equivalence of network dataplanes using input-output relations. As such, we need to show that for all **input packets** at the same **input ports**, we get the same **output packets** on corresponding **output ports**.

EQ follows a similar approach to that depicted in UC-KLEE [7] and returns whether or not the two programs are

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Posters-Demos '18, August 20–25, 2018, Budapest, Hungary

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5915-3/18/08...\$15.00

<https://doi.org/10.1145/3234200.3234224>

```

If (TTL>0) TTL--;
Else Fail;
If (dst in 10.10.0.1/32)
  Forward("if0");
Else If (dst in 10.10.0.0/16)
  Forward("if0");
Else If (dst in 10.0.0.0/8)
  Forward("if1");
Else Fail;

```

Basic router coded with If/Else: (l)code (r)symbex

Figure 1: Two SEFL programs modeling a router with three entries in its FIB. Are they equivalent?

identical with respect to input-output processing.

The approach taken to show equivalence between two dataplane program snapshots P_1, P_2 involves translating them into SEFL code, identifying the corresponding input and output ports and then exhaustively running symbolic execution starting from each equivalent input port pair. Symbolic execution with Symnet [8] will generate a set of outcomes and the conditions of their occurrence. EQ detects the conditions under which the two SEFL programs behave differently - e.g. issue packets on distinct ports, with different header fields etc. We have built automated tools to translate many network processing primitives - such as OpenFlow and P4 switches, iptables firewalls, routing tables, Click routers - into SEFL. Thus, there is no need to manually create or maintain SEFL models.

Because of special properties of network dataplanes - no *while* loops, very little stateful processing -, we found that equivalence checking is scalable in a number of scenarios and is able to detect subtle implementation bugs. The problem of undecidability of general purpose imperative programs is circumvented due to packet length boundedness. In the following section, some applications of network program equivalence are described.

We built *sdiff* as a reference implementation of EQ. *sdiff* consists of ~2kLOC of Scala code; we use it for the evaluation of the approach.

3 EVALUATION

A major application of equivalence is testing if an SDN controller is compliant to *tenant intention*. An SDN controller "compiles" a high-level tenant intention into a number of machine-specific forwarding entries (OpenFlow rules, P4 table entries, FIB entries). This is a common approach enabling cloud vendors to provide tenants with rich network topologies - traffic control, security, L2 and L3 forwarding primitives.

To evaluate *sdiff*, we used OpenStack, an open-source cloud software and tested with inputs against Neutron, its networking driver. We translated iptables and openvswitch rules to SEFL and check equivalence with respect to the high-level network program configured by the cloud tenant. We successfully discovered 6 implementation bugs (one

```
TTL--;Constrain(TTL>=0);
```

```

Fork (
  Path1 {
    Constrain(dst in 10.10.0.0/16);
    Forward("if0");
  }
  Path2 {
    Constrain(dst in 10.0.0.0/8);
    Forward("if1");
  }
)

```

Optimization with Fork/Constrain: (l)code (r)symbex

Algorithm	Prefixes in FIB					
	100	500	1000	2000	5000	10000
Basic	0.74s	0.6s	0.8s	0.9s	2.3s	6s
Optimized	0.1s	0.2s	0.4s	0.8s	1.6s	3s
sdiff	1.68s	3.6s	6.9s	14.5s	57.9s	240s

Figure 2: Testing two symbolic routers for equivalence.

previously unknown), and 3 configuration bugs.

P4 is a high level language that enables programming switch dataplanes that can also be efficiently implemented in hardware. We use *sdiff* to show that two apparently identical implementations of the same network function differ. The first implementation is modular, easier to design and specify, while the second one is tailored for performance. Despite appearing identical, *sdiff* was able to find two differences between the two.

Another important application for EQ and evaluated using *sdiff* consists of *baselining* operations; in this scenario, an administrator, after having thoroughly tested a network configuration, wants to ensure the same behavior is preserved after configuration changes. We show *sdiff* successfully detects discrepancies between FIBs and pinpoint how they can occur.

We also find equivalence useful for showing that optimized FIB models are correct with respect to their basic counterparts. We evaluate *sdiff*'s scalability for up to moderate sized routing tables in Figure 2.

4 CONCLUSIONS

Equivalence testing enables easy-to-use verification, and we have used it to find many Openstack Neutron and P4 program bugs, among others. While equivalence checking is more expensive than individual symbolic execution of the two programs, it runs in a few minutes for routers with 10K entries in their FIBs and in Openstack deployments with tens of virtual machines. Such scalability widens the applicability of *sdiff*.

REFERENCES

- [1] Ari Fogel et al. "A General Approach to Network Configuration Analysis". In: *NSDI*. 2015.
- [2] Peyman Kazemian, George Varghese, and Nick McKeown. "Header Space Analysis: Static Checking for Networks". In: *Proc. NSDI'12*.
- [3] Ahmed Khurshid et al. "VeriFlow: Verifying Network-wide Invariants in Real Time". In: *Proc. NSDI'13*.

- [4] Hongqiang Harry Liu et al. “CrystalNet: Faithfully Emulating Large Production Networks”. In: *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*.
- [5] Nuno P. Lopes et al. “Checking Beliefs in Dynamic Networks”. In: *Proc. NSDI'15*.
- [6] Haohui Mai et al. “Debugging the data plane with anteater”. In: *Sigcomm*. 2011.
- [7] David A. Ramos and Dawson Engler. “Under-Constrained Symbolic Execution: Correctness Checking for Real Code”. In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 49–64.
- [8] Radu Stoenescu et al. “SymNet: Scalable symbolic execution for modern networks”. In: *SIGCOMM*. 2016.