Dataplane equivalence and its applications

Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu and Costin Raiciu
firstname.lastname@cs.pub.ro
University Politehnica of Bucharest

Abstract

We present the design and implementation of `netdiff`, an algorithm that uses symbolic execution to check the equivalence of two network dataplanes modeled in SEFL [42]. We use `netdiff` to find new bugs in Openstack Neutron, to test the differences between related P4 programs and to check the equivalence of FIB updates in a production network. Our evaluation highlights that equivalence is an easy way to find bugs, scales well to relatively large programs and uncovers subtle issues otherwise difficult to find.

1 Introduction

Misconfigured or faulty networks ground airplanes, stranding thousands of passengers and render online services inaccessible for hours on end, leading to disgruntled users and massive losses in revenue. Network verification promises to fix such rare yet devastating problems by ensuring that networks always follow their operator's stated policy. Verification proposals can uncover faulty dataplane configurations [19, 29, 42, 30, 20], can simulate the effect of control plane changes (such as configuration changes) before they are applied [8, 11, 4] or inject these changes into an emulated clone of the live network to examine their effects [26]. The key behind the success of network verification in traditional networks is the simplicity of the policy, a mix of reachability and isolation constraints that administrators can readily specify once and verify recurrently.

As networks become more dynamic and programmable, both ensuring network correctness and specifying policy are significantly harder. Virtual networks are instantiated dynamically in cloud networks on tenant demand by massive software stacks, potentially developed by multiple players (e.g. Openstack); here, the key challenge is to ensure that tenant demands are implemented correctly and that tenant traffic is properly isolated from other tenants.

Languages such as P4 [5] or Flowblaze [34] allow the implementation of customized packet processing logic that can be deployed and run at wire speeds on real switch hardware (e.g. Barefoot's Tofino). Specifying the behavior of programmable dataplanes entails specifying the expected output packet(s) for every possible input packet; such a specification relies on formal methods and expert time [38, 35, 46], being out of reach of network administrators and programmers.

We observe that, in many cases, dataplane correctness properties can be specified implicitly by equivalence to other dataplanes. A P4 programmer might need to restructure or trim his program to meet the target switch constraints [16, 39] while preserving the functionality. In cloud computing, the abstract network configuration provided by tenants (e.g. two VMs connected via a L2 network) is translated by the cloud management software into an actual configuration for switches and routers that must offer *equivalent* functionality to the two VMs. Finally, a network administrator that knows his network behaves correctly[1] simply wants the network to behave in the same way in the future.

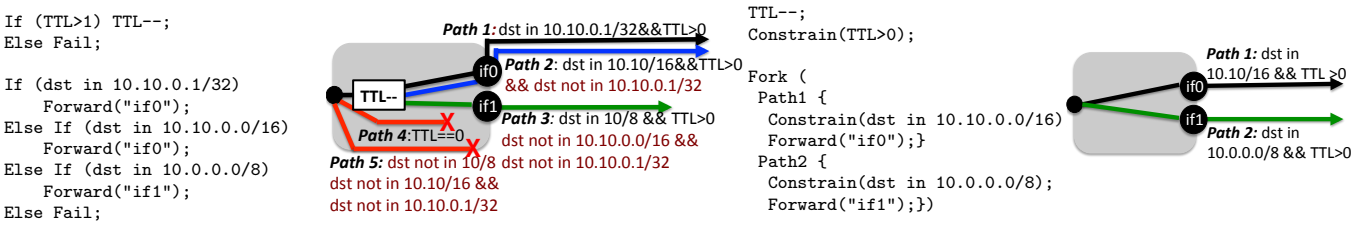
Checking equivalence could therefore be very useful for easy-to-use verification of modern dataplanes. Unfortunately, checking equivalence between arbitrary programs is a well-known undecidable problem. Variants of it, however, are decidable for domain-specific programming languages; in the networking field, NetKAT [3], NOD[29] and HSA [19] support various forms of equivalence checking. These languages, however, are not expressive or not efficient enough to check programmable dataplanes such as P4.

In this paper we show that checking equivalence is possible for programmable dataplanes and that it scales well enough to uncover many interesting bugs. `netdiff`, our proposed algorithm, is implemented on top of the Symnet symbolic execution engine and can test the equivalence of two network dataplanes expressed in the SEFL language [42]. We formally prove `netdiff` correctly decides if two dataplanes are equivalent when they do not contain infinite loops; we rely on prior work to detect infinite loops [43].

We have used `netdiff` to find bugs in Neutron, OpenStack's cloud management software networking driver, by checking the equivalence of tenant configurations and the low-level implementation of those configurations. We have found ten implementation bugs in Neutron, three of which were unknown, and four configuration bugs. We have also used `netdiff` to check that P4 program optimizations preserve correctness, to test different dataplane models of the same network functionality are equivalent, to detect routing changes in a university network and to check that a FatTree instance behaves like a single, big switch. `netdiff` runs all these tasks in seconds/minutes. Finally, to enable scalability to a large Neutron deployment, we rely on a compositional verification approach where we test equivalence for independent components in isolation.

2 Goals

Network dataplane equivalence has many potential applications, a subset of which we explore in detail in §6. To guide our exposition, we use as running example the code snippets



(a) Basic router coded with If/Else: code(left) symbex(right)

(b) Optimized router: code(l) symbex(r)

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

in Figure 1 that model a router with three FIB entries; the code is adapted from [42]. Despite its simplicity, the example highlights well the difficulty of equivalence checking.

The first program is simple to understand as it relies on a sequence of `If/Else` clauses that forward the packet to the correct output port; there is one `If` per FIB entry. The second program is optimized to enable faster dataplane verification: it does not use `If` instructions at all, first `Fork`*ing* 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; 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. Even though the two programs seem trivial, checking their equivalence is not possible today.

Our goal is to automatically and scalably decide if two dataplane programs are equivalent.

Before we discuss possible solutions, we first give a formal definition of equivalence. Let *Prog* denote the set of programs - defined as mappings (functions) between *Ports* (function names) and instructions. Let *Packet* denote the set of all admissible input packets. Injecting a packet p into a program $prog$ at port $port_0$ will result in a set of output packet and port pairs $O(prog, p, port_0)$ defined as follows.

Definition 2.1 *Let* $O : Prog \times Packet \times Ports \rightarrow 2^{Packet \times Ports}$*:*

$$O(prog, p, port_0) = \{(\sigma_1, port_1), (\sigma_2, port_2), ..., (\sigma_n, port_n)\}$$

be the set of packet and output port pairs resulting from the execution of prog given packet p on input port $port_0$*.*

We define **network equivalence** as follows:

Definition 2.2 *Let* $p \in Packet$ *an input packet,* $P_1, P_2 \in Prog$*,* $Ports_1$ *and* $Ports_2$ *the program ports of* P_1 *and* P_2 *respectively. Let* $\mathscr{I}$ *and* $\mathscr{R}$ *be partial injective functions between* $Ports_1$ *and* $Ports_2$ *called input and output port correspondence respectively.*
We call programs P_1 *and* P_2 ***equivalent*** *with respect to input packets* $Q \subseteq Packet$*, input ports* $port_1 \in Ports_1$ *and* $port_2 \in Ports_2$ *s.t.* $\mathscr{I}(port_1) = port_2$ *and output relation* $\omega \subseteq Packet \times Packet$ ***iff*** $\forall p \in Q$*,* $\exists \chi$ *bijection between* $O(P_1, p, port_1)$ *and* $O(P_2, p, port_2)$ *s.t.*

$$\chi(\sigma_{1i}, pc_{1i}) = (\sigma_{2j}, pc_{2j}) \iff \mathscr{R}(pc_{1i}) = pc_{2j} \wedge (\sigma_{1i}, \sigma_{2j}) \in \omega$$

Definition 2.3 *We call* P_1 *and* P_2 ***equivalent*** *with respect to* $Q \subseteq Packet$ *and* $\omega \subseteq Packet \times Packet$ *iff* $\forall (port_1, port_2) \in \mathscr{I}$ P_1 *and* P_2 *are equivalent w.r.t. Q,* port_1*,* port_2 *and* ω*.*

Intuitively, the above definition goes to say that given the same input packet, the number of output packets from both programs coincide and there must be a one-to-one correspondence between packets emitted by both programs. Also, packets in correspondence must satisfy the output packet condition ω, which typically requires that the values of selected header fields in the two packets are equal.

It is the verifier who provides $\mathscr{I}$, $\mathscr{R}$ and ω. For example, in Figure 1, $\mathscr{R}$ maps $if0$ and $if1$ in a) to ports with the same name in b) and $\mathscr{I}$ maps the input port in a) to that in b). ω usually identifies a subset of header fields which must be equal. In our running example, two packets are equivalent if the ttl and dst fields are equal. In our evaluation, we use sensible defaults for these functions - L2, L3 and L4 fields.

3 Approaches to checking equivalence

Exhaustive testing for all inputs is one way to test equivalence, but it is not feasible to use in practice for networks: network headers size are 64B or more, meaning that one needs to test 2^{512} possible packets.

Existing work on dataplane verification allows us to scalably explore how packets are processed by a dataplane [19, 3, 29, 20, 42, 7, 30]. Intuitively, all these works try to find equivalence classes of packets that are handled in the same way, and explore their processing in one go; as long as the number of such classes is small, these tools can fully characterize dataplane processing without needing to explore each individual packet. We will take the same approach to answer whether two dataplanes are equivalent.

To enable scalability, all dataplane analysis tools restrict the language in which the dataplane can be described, place additional constraints on possible encapsulations and use optimized data structures to track packet equivalence classes.

All these choices limit the extent to which we can check equivalence; we will come back to these limitations after we discuss the equivalence properties we seek to capture.

All dataplane verification tools are able to predict the allowed values of packet header fields as they exit a given network port. The simplest way to check equivalence is to compare which packets can exit any given port by examining the feasible values for each header field—we call this **output equivalence**.

In the example in Figure 1, consider the two `if1` ports: compared to the basic model, the optimized model wrongly allows more packets to pass (packets in 10.10.0.0/16), so the two paths are not equivalent, and thus the models are not equivalent for ports `if1`. If, however, we consider the two `if0` ports, we find that TTL $\in (0, 255]$ and dst $\in 10.10.0.0/16$, thus the two paths are equivalent.

The careful reader will have noticed that the two routers differ in how they treat packets when `TTL` is 0. The basic router will drop the packet straightaway. However, the optimized router will decrement the `TTL` regardless of its value, and when it is zero it will wrap around to 255 as `TTL` is unsigned — thus, the constraint `TTL`> 0 always holds. The two models are not equivalent, but checking just output equivalence is not enough to capture this problem.

The next step is to also check the constraints applied on the original (input) values of the header fields, before any modifications are made; when combined with output equivalence checking, we are now checking for **input** and **output equivalence**. With input/output equivalence, we will find that the basic model only allows packets to pass when `TTL`> 1 while the optimized model allows packets when `TTL`$\neq 1$; as the two ranges are not the same, the two models are not equivalent.

Checking for input/output equivalence is necessary to find bugs, but on its own it is still not sufficient. To see why this is the case, consider two trivial models where one leaves the TTL field unchanged, while the other executes the instruction $TTL = 255 - TTL$. Both the input values (0-255) and the possible output values (0-255) of the two models are the same, yet they are obviously not equivalent. What we also need is **functional equivalence**: regardless of the initial value of the TTL, the two values of the TTL after executing the two programs should always be equal. In our example, functional equivalence is not true because the condition $TTL = 255 - TTL$ never holds.

Note that all three checks are simultaneously needed to ensure equivalence: removing a single check leads to wrong results. We have already shown that input/output equivalence is not sufficient and functional equivalence is needed. Let's show that any combinations of two checks is insufficient.

Do functional and output equivalence imply two models are equivalent? Consider one program that simply sets `TTL = 0`, and another that runs `Constrain(TTL>100);TTL=0`. The output is always 0, and for any allowed packet we have both functional and output equivalence. Yet, the first model allows all packets through, while the second only allows those with `TTL>100`; the programs are not equivalent.

Algorithm	**Equivalence**			**Expressiveness**
	Input	**Output**	**Func.**	
HSA[19], Veriflow[20]	✓	✓	✗	forward,filter
NetKAT [3]	✓	✓	✓	switch,filter
NOD [29]	✓	✗	✗*	forward,filter,tunnel
Dobrescu[7],Symnet[42] UC-KLEE[36]	✓	✗	✓	programmable dataplane (e.g. P4)

Table 1: Checking equivalence with existing tools.

Input and functional equivalence are also insufficient. Compare a NoOp program with one that `fork`s the packet. These two are equivalent from an input and functional point of view, however they are not equivalent on output: the first emits a single packet while the secqond emits two.

3.1 Existing solutions fall short

Existing dataplane verification tools (see Table 1), cannot check equivalence for programmable dataplanes. Header Space Analysis [19] and Veriflow [19] optimize for OpenFlow-like processing by tracking equivalence classes of packets through the network. Both are fast and their outputs can be fed to SMT solvers to check for output and input equivalence. Unfortunately, they do not track (symbolic) assignments and cannot scalably check functional equivalence.

NetKAT [3] offers a strong theoretical foundation to OpenFlow verification by reducing it to a Kleene algebra with tests. They show that equivalence is decidable in this algebra, and offer an efficient equivalence checking algorithm [9]. Compared to HSA and Veriflow, NetKAT supports assignment but lacks support for arithmetic operations. As such, it cannot express programmable dataplanes.

Network-optimized datalog [29] uses datalog to express network processing and policies. NOD is more expressive than prior tools because it also supports arbitrary tunnels, and checking equivalence is just another datalog query that can be fed to Z3 [6]. On the downside, it is very difficult to use datalog queries to reason about packet multiplicity on various ports. Furthermore, NOD's difference-of-cubes is very inefficient for arithmetic operations, both space-wise [2] and computation-wise [3]. Thus, NOD does not support neither output nor functional equivalence.

Symbolic execution for network dataplanes has been proposed by Dobrescu et al.[7] and Symnet [42]; it tracks the symbolic values of header fields and supports assignment, encapsulation and arithmetic operations. Symbolic execution is expressive enough to analyze programmable dataplanes as shown by recent work [40, 10, 32]. While symbolic execution has traditionally been plagued by poor scalability, applying it to dataplanes has been shown to scale quite well.

Checking dataplane equivalence via symbolic execution is not supported by [7, 42, 40, 32, 10], but prior work from program symbolic execution can be adopted. UC-KLEE is the

leading proposal [36]: it can check for all types of equivalence for standard programs, but is not expressive enough to deal with packet duplication, a common primitive in network processing. We present `netdiff`, our algorithm that fixes this shortcoming.

4 Dataplane equivalence with `netdiff`

`netdiff` uses symbolic execution to show equivalence of two dataplanes according to definition 2.2. To enable scalability and expressiveness, we consider network dataplanes written in the SEFL language that only provides a set of basic instructions such as `if then else`, a filter (`constrain`) instruction, variable assignments and jumps to predetermined locations in the program called *ports*. The specificity of dataplane processing consists in the existence of an additional cloning instruction (`fork` in SEFL) which produces multiple copies of the same packet and pushes them on different paths through the network.

The symbolic execution state of a dataplane program is represented by a set of variables (packet header values and associated per-flow state) and a program counter which indicates the next instruction to be executed. A path through such a program is a list of program counters. Symbolic execution begins at some initial *port*, takes a *packet* as input, where some or all header fields can have symbolic values, and produces a set of packets issued on output *ports*. Symbolic execution is a method to exhaustively infer predicates on the input variables of a program in order for the execution to take some path [22]. The outcome of symbolic execution is a set of pairs comprised of a *path condition* and the corresponding *path*. The *path condition* is a logical proposition required by the inputs to a program such that execution will follow a certain path through the program.

For concreteness, consider Figure 1 where we inject at router input a packet with symbolic `TTL` and IP destination address (`dst`) fields, meaning they can take any value allowed in their range. The figure shows the symbolic execution of the two programs in our running example, the resulting paths and path conditions.

When a branch condition depends on a symbolic variable, the symbolic execution engine uses a constraint solver to check if the condition is satisfiable: if it is, the constraint is recorded in the path and the execution continues on the "then" branch (path). At the same time, the engine checks whether the negated constraint holds, and if it does it also continues execution on the "else" branch, recording the negated constraint that must hold. Both paths are now explored until they finish, independently. For instance, in the basic router program, the first `If` branch results in a path where the `TTL` is at least 2, then decrements the `TTL` and forwards the packet to the appropriate interface(s). The else path where the TTL is 0 or 1 is also explored, but it stops immediately because the packet is dropped.

Algorithm 1 `netdiff` equivalence algorithm

```
1:  function EQUIVALENCE(M1, M2, i1, i2, p0)        ▷ Are M1 and M2 equivalent for input symbolic packet described by predicate p0 injected on ports i1 and i2?
2:      Q1 ← DataplaneSymbex(M1, i1, p0)
3:      for all (q1, π1) ∈ Q1 do
4:                                                  ▷ for each path π1 and path condition q1
5:          Q2 ← DataplaneSymbex(M2, i2, q1)
6:          for all (q2, π2) ∈ Q2 do
7:              if ¬EQP(π1, π2, q2) then
8:                  return false
9:              end if
10:         end for
11:     end for
12:     return true
13: end function
```

`netdiff`, our proposed algorithm, is shown in Algorithm 1 and uses symbolic execution to check for equivalence between two SEFL programs. `netdiff` takes as input SEFL programs M_1 and M_2 and injects a set of packets given by predicate p_0 into the user-specified input ports i_1 of M_1 and i_2 of M_2. The procedure $DataplaneSymbex(M, i, p)$, described in detail in subsection 4.1, performs symbolic execution for program M starting at input port i with a symbolic input given by predicate p on the set of all possible input *Packet*s. Each π resulting from symbolic execution represents a pathset, which is an individual path or a set of paths that have the same path condition q (the latter captures cloned packets).

`netdiff` follows a similar approach to UC-KLEE [36]: it performs symbolic execution of M_1 (line 2) and then, for each resulting pathset (q_1, π_1), performs symbolic execution of M_2 starting with initial symbolic packet described by the path condition q_1 (line 5). The algorithm then compares each resulting pathset (π_2) to (π_1) for the packets in q_2 (line 7).

`netdiff` ensures input equivalence by design because the union of the sets of packets described by all predicates $q_2 \in Q_2$ must be equal to the set described by q_1, and all sets described by q_2 predicates are disjoint (see Lemma 1). The EQP predicate's job is to check for output and functional equivalence for each pair of outputs. There are two main differences between `netdiff` and UC-KLEE, both stemming from packet cloning, that we describe in detail below:

1. The `fork` instruction can result in multiple paths for the same set of input packets; for `netdiff` to work correctly, the standard symbolic execution is followed by processing that groups output paths that have overlapping path conditions into pathsets (see §4.1).
2. Finding the right path equivalence predicate *EQP* to reflect Definition 2.1 is also tricky. For sequential imperative languages, this predicate is a simple equality check for the output values of the two paths being compared. To cope with packet cloning, we need to compare pathsets instead of individual paths (see §4.2).

Algorithm 2 Dataplane symbolic execution

```
1: function DATAPLANESYMBEX(M, i, p_0) ▷ Run symbolic execution by injecting the symbolic packet described by predicate p_0 in port i.
2:     Q ← Symbex(M, i, p_0)
3:     L ← ∅                                  ▷ L holds pathsets with disjoint conditions
4:     for all (q, π) ∈ Q do                  ▷ Sieving algorithm to collapse paths
5:         for all (l, s) ∈ L do              ▷ with overlapping path conditions.
6:             if SAT(q ∧ l) then
7:                 L ← L \ {(l, s)}
8:                 L ← L ∪ {(q ∧ l, s ∪ π), (l ∧ ¬q, s)}
9:                 (q, π) ← (q ∧ ¬l, π)
10:            end if
11:        end for
12:        if q ≠ ∅ then
13:            L ← L ∪ {(q, π)}
14:        end if
15:    end for
16:    return L
17: end function
```

4.1 Dataplane symbolic execution

Algorithm 2 shows our dataplane symbolic execution algorithm. On dataplane code, standard symbolic execution *Symbex* returns tuples of (path condition, path) in the set Q, but the path conditions are not guaranteed to be disjoint, as they would be in a standard program. This can be seen in our running example (Figure 1.b): symbolic execution yields Path 1 and 2 with overlapping path conditions ($dst \in 10.10/16$).

Dataplane symbolic execution performs sieving to eliminate path condition overlaps, returning *pathsets* with disjoint path conditions. To achieve this, the algorithm first runs standard symbolic execution, and then proceeds to group all paths that have overlapping path conditions (lines 6-9).

The result is collected into L, which starts as the empty set and always contains pathsets with disjoint path conditions. Whenever there is a path condition overlap (which we test with Z3), we remove the existing pathset from L and insert two new pathsets: one containing the overlapping condition $q \wedge l$ and the union of the paths, and one corresponding to original pathset with updated path condition $l \wedge \neg q$. The iteration then continues with the remaining path predicate $(q \wedge \neg l)$; note that the newly added entries are not revisited.

In the example from Figure 1.b, L has two pathsets: $(10.10/16, \{Path1, Path2\})$ and $(10/8 \setminus 10.10/16, \{Path2\})$.

The complexity of this algorithm is $O(|Q|^2)$; as symbolic execution can yield many paths, this cost can quickly become prohibitive. Changes to the symbolic execution procedure to reduce this cost to be proportional to the number of `fork` instructions are part of our future work. Instead, in our implementation we use a heuristic where we only run the algorithm for broadcast packets and disable it for unicast packets; the user can override this behavior via a command-line flag.

4.2 Equivalence between pathsets

Given pathsets O_1 (from M_1) and O_2 (from M_2) and path condition pc, we need to decide if the two pathsets obey functional and output equivalence.

Algorithm 3 EQP predicate between two pathsets

```
1: function EQP(O_1, O_2, pc) ▷ True if bijection between O_1 and O_2 found
2:                            ▷ O_1 and O_2 are pathsets and pc the path condition
3:     if |O_1| ≠ |O_2| then
4:         return false       ▷ If cardinality different, there is no bijection
5:     end if
6:     E ← ComputeEdges(O_1, O_2, pc)
7:     G = (V = (O_1 ∪ O_2), E)
8:     return MaxBipartiteMatching(G)
9: end function
```

Algorithm 4 Edge computation

```
1: function COMPUTEEDGES(O_1, O_2, pc) ▷ Return the adjacency matrix
2:                                     ▷ O_1 and O_2 are pathsets, pc the current path condition
3:     for all (σ_i, p_i) ∈ O_1 do
4:         for all (σ_j, p_j) ∈ O_2 do
5:             E[i][j] = p_i ℛ p_j                          ▷ output port equivalence
6:             if E[i][j] then
7:                 E[i][j] = ¬SAT(pc ∧ ¬(σ_i ω σ_j))        ▷ Func. equiv.
8:             end if
9:         end for
10:    end for
11:    return E
12: end function
```

First, consider the simple case where the two pathsets have exactly one path each. Let σ_1 and σ_2 denote the values of the header fields for the two paths at the output ports, expressed as constants or functions of the symbolic header values at input. To decide equivalence according to Definition 2.1 we must check whether: (1) the two paths exit on equivalent output ports and (2) the output packet values satisfy the ω relation (i.e. the header fields of interest are the same for all input packets described by pc).

`netdiff` checks equivalence between two paths as follows: $p_i \mathscr{R} p_j \wedge \neg SAT(pc \wedge \neg(\sigma_1 \omega \sigma_2))$. The port check is obvious; the second check asks the solver for an input packet that satisfies pc and results in output packets that are not equivalent. If the check is not satisfiable, functional equivalence holds for all packets allowed by path condition pc. In this case, there is a single path in each pathset and input and functional equivalence guarantee output equivalence.

Now, consider the case where the two pathsets have different number of paths. This implies the two programs are not equivalent because there is some input packet which results in a different number of output packets being emitted.

Finally, consider the remaining case where the two pathsets have the same cardinality $N > 1$. To check equivalence we must find a bijective mapping between the paths in O_1 and O_2, i.e. each path in O_1 must have a path in O_2 that is equivalent, and all paths in O_2 must have an equivalent in O_1. If such a mapping exists, then the two pathsets are equivalent, otherwise they are not.

We now show that finding this bijection can be reduced to the classical problem of maximum bipartite matching (MBM). MBM takes as input a bipartite graph $G = (V = (X \cup Y), E)$ with $X \cap Y = \emptyset$, where X is the set of workers

and Y is the set of tasks such that $|X| = |Y|$. If worker i qualifies for job j, there is an edge from i to j, i.e. $E[i][j] = 1$. In MBM, no worker can take more than a job; the algorithm decides whether all workers can get jobs.

The EQP equivalence algorithm is implemented in Algorithms 3 and 4. O_1 and O_2 are the pathsets to be compared; their paths will form the vertices of our bipartite graph. We have an edge between two paths if the paths are matching: $E[i][j] = 1$ iff the output ports are corresponding w.r.t. $\mathscr{R}$ and the output packets are equivalent w.r.t. ω. EQP calls MBM to find whether all paths in O_1 have an equivalent in O_2. If the answer is positive. then is a bijection between these paths which guarantees both functional equivalence and output equivalence. As input equivalence is ensured by the way in which we run symbolic execution, `netdiff` correctly decides whether the two programs are equivalent.

The bottleneck in EQP is the necessity of computing the *output equivalence* between all pairs of paths in O_1 and O_2 in order to derive the graph (Algorithm 4, line 6). Computing satisfiability of a complex formula is a hard problem and dominates the entire equivalence decision procedure. That is why our algorithm first checks for output port equivalence; when this check fails, the solver call in Line 6 is not made.

4.3 Correctness and complexity

It is worth noting that we can decide the equivalence of two dataplane programs only if they terminate on all inputs. `netdiff` uses an existing loop detection algorithm to detect infinite loops [43], which leads to the following possibilities:

- Both programs are infinite-loop free. We can decide on their equivalence with a worst case complexity of $C(DataplaneSymbex_1) + n \cdot \big(C(DataplaneSymbex_2) + m \cdot p^2 \cdot C(SMT)\big)$, where n and m are the number of M_1's and M_2's pathsets, p is the maximum number of paths in any pathset and $C(SMT)$ is the complexity of the SMT solver.
- When only one of the programs contains an infinite loop, equivalence is decidable - the programs are not equivalent.
- When both programs have infinite loops, we cannot decide

Theorem 4.1 EQUIVALENCE$(M_1, M_2, i_1, i_2, p_0)$ *is true iff* M_1, M_2 *are equivalent w.r.t.* $Q = \{x \in Packet | p_0(x) = true\}$ *and input ports* i_1, i_2.

Proof: Because of the properties of symbolic execution listed in Appendix A, all path conditions corresponding to pathsets resulting after symbolically executing M_2 are equivalence classes of the initial input space w.r.t. the outcomes of M_1 and M_2. The full proof can be found in Appendix A. □

5 Implementation

Our implementation of `netdiff` takes as input two SEFL programs, together with two correspondence maps, one between their input ports and the other with respect to their output ports. By default, it injects basic Ethernet/IP/{TCP, UDP, ICMP} packets into the provided input ports and checks for equivalence between a number of header fields including IP and Ethernet source and destination, TTL, IP protocol, L4 ports, etc. Note that both the symbolic input packet and the fields to be checked can be customized by the user if needed.

When equivalence fails, `netdiff` outputs a series of tuples *(Input condition, Offending path in program 1, Offending path in program 2, reason)*, where the reason may be either different number of output packets, unmatched ports, unmatched output fields. A path in program x is a series of symbolic output packets together with a list of visited ports and either the output port or the failure reason.

We ran `netdiff` on the example shown in Figure 1. It takes around 200ms to check equivalence and output the results to file. The tool reports 10 symbolic packets that were treated differently by our two programs. The first five packets catch the TTL decrement bug in the optimized model (the TTL underflows when it is zero). The other five packets highlight the second problem: there is an overlap between packets exiting on port `if0` in the basic model and `if1` in the optimized model - i.e. a packet in 10.10/16. `netdiff`'s output makes it easy to find the bug and then correct it.

Core implementation. The core of `netdiff` is implemented in just 200 lines of Scala code. We also implemented a series of additions and changes to Symnet resulting in cca. 2kLOC that ensure `netdiff` behaves correctly. In detail, we implemented sieving (Algorithm 2) that takes the outputs of Symnet—a list of tuples (Input condition, single path through network)— and outputs pathsets of the form (path condition, paths) with non-overlapping path conditions. Furthermore, we changed the internal representation of Symnet's state to allow a clean separation between path conditions and current packet values, in order to make `netdiff`'s internals more scalable and easier to test. To aid debugging and model validation, we also implement an input generator which produces an example packet from a path condition.

5.1 OpenStack Neutron Integration

`netdiff` requires two SEFL programs to check equivalence. To generate SEFL programs for our evaluation, we have both created new translators (see below) and re-used existing ones: Vera to translate from P4 programs to SEFL [40] and existing translators from router FIBs to SEFL [1].

Checking OpenStack Neutron is our most significant use of equivalence so far; it required 15KLOC of Java, Scala and ANTLR4 parser grammars to automatically integrate with Neutron and translate to SEFL. Out of this, only 1KLOC is OpenStack-specific, while the biggest part consists of reusable translators for iptables, OVS, ipsets, which are widely deployed in numerous scenarios - e.g. Kuberenetes. Furthermore, this work is one-time only and can be further used for any other verification purpose. We contrast this effort to

other verification techniques which also require writing and maintaining correctness specifications. We describe our implementation here.

OpenStack is an open-source cloud management platform. Similarly to other cloud platforms (e.g. EC2), OpenStack abstracts away the complexity of the provider's infrastructure, allowing the tenant to create and manage virtual machines with ease. Tenants may connect their VMs in rich network topologies comprised of VLANs connected via routers and NATs, and also enforce security policies at VM level.

Neutron is the networking service of OpenStack and is implemented as a distributed middleware application which takes the *tenant network configuration* and implements it in the actual network. Neutron's operation is complex as it depends on multiple software components, so bugs may occur anytime during the translation process leading to non-compliance to the tenant configuration. Neutron's complexity and its distributed deployment makes manual troubleshooting cumbersome.

To verify whether Neutron correctly implements a given tenant configuration, we automatically generate two SEFL models and use `netdiff` to check their equivalence. The first model is derived from the tenant configuration. The second model is created from a snapshot of the actual Openstack deployment resulting after the VMs are instantiated and it includes OVS OpenFlow rules, iptables rules, etc.

Modeling tenant configurations. Our translator uses a tenant-level snapshot of the configuration (a dump of the Neutron database in practice) and then generates SEFL code that implements each user defined resource (e.g. router, switch, NAT). The simplest abstraction is a virtual network which forwards packets according to a static CAM table mapping Ethernet addresses to virtual ports. Virtual routers perform L3 routing and provide virtual machines with access to and from the Internet (via NAT). Neutron also defines *security groups* which are rules that filter traffic at VM level. Each abstract object is translated separately to SEFL, and they are connected using links according to the tenant configuration. For more details, we refer the reader to[41].

Dataplane modeling. A Neutron deployment is usually implemented as interconnected Linux servers running a number of network processing tools. We implement parsers and SEFL translators for many of the Linux Kernel packet processing primitives - `iptables` rules, ipsets, OpenVSwitch (OVS) software switches [33], OpenFlow tables, VXLAN tunnels, routing tables, Linux Bridges, ARP tables. We then interconnect the distinct components based on the physical or virtual links acquired from the topology.

Both *iptables* and *OVS bridges* use similar concepts such as tables and rules which match against packet header fields or per-packet metadata and apply one or more actions. To translate such matches we generate simple `If/Else` constructs; provided that all *matches* in a rule are satisfied, an action will be fired which will either alter the state of the packet (e.g. push a tunnel header) or alter the processing pipeline (e.g. drop or forward to further processing).

Modeling stateful processing in SEFL is straightforward as long as the state depends only on the given flow (i.e. it is not global state) [42]. We use a similar technique to model the connection tracking engine (or *conntrack*) implemented within the Linux Kernel. Conceptually, *conntrack* defines a connection as a 5-tuple and tracks it independently. To model *conntrack* we use two sets of metadata variables called *forward* and *backward expectations*. The former represent packets flowing in the same direction as the initial SYN packet, while the latter represent reply packets belonging to the same connection. When state is created for a connection (a conntrack commit action), we store it as metadata; the metadata is then checked when execution arrives at the conntrack module and the appropriate action is taken.

Dataplane modeling caveats. One of the issues that we encountered during our experiments was missing information from the dataplane snapshot which lacked ARP tables and switch CAM tables. A possible solution to bypass this issue is to simply modify the acquisition script to gather ARP tables for all Linux network namespaces and CAM tables for all OVS and Linux Bridges in the topology. However, these entries would only depict a transient state of the network dataplane with incomplete or stale information.

Our solution implies converging the ARP and CAM tables into a steady, concrete state. Following the observation that cloud provider middleware implements anti-spoofing techniques, we use the constraint solver's capabilities to derive all possible ARP packets which may reach a certain point in the network. With this information, we infer *(IP, MAC)* pairs for all network namespaces in the system. We use a similar approach to infer all CAM tables on all switches in the deployment. We implement our approach in a tool called `ARPSim` and apply it to our department's Openstack deployment containing 87 servers. For this deployment, `ARPSim` discovers 885 ARP entries in 4 minutes and infers 28889 CAM entries in L2 switches in 7 minutes.

Mapping ports. As discussed in §2, `netdiff` provides sensible defaults for mapping ports and deciding what packets are equivalent. For Neutron, we map virtual ports to corresponding OpenVSwitch tap interfaces (functions $\mathscr{R}, \mathscr{I}$).

6 Evaluation

We run our evaluation of `netdiff` on a server with a Xeon E5-2650 processor @ 1.7GHZ and 16GB of RAM. Our main goal is to understand whether `netdiff` can catch interesting bugs or behaviors in practice, for realistic network dataplanes. We examine a range of applications including Openstack Neutron, P4 program equivalence, and the correctness of FIB updates. Finally, we test `netdiff`'s scalability and contrast its performance to NoD [29]. The scenarios described below make use of header arithmetics and packet du-

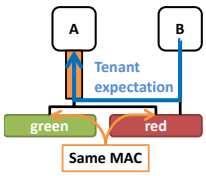


Figure 2: Identical trunked MACs

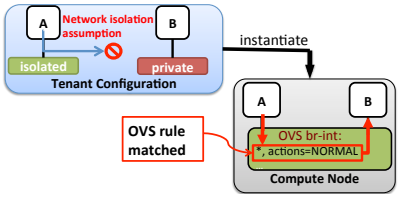


Figure 3: Network isolation bug

plication primitives and motivate the use of `netdiff` instead of systems like HSA, NetKAT or NoD.

6.1 Neutron bugs

To validate `netdiff`'s capabilities in finding production bugs, we begin our experiments by first reproducing known bugs and discovering a number of unknown bugs in Openstack Neutron in a small scale deployment in our lab. Our experiments here focus mostly on functionality.

While Neutron is only one of many cloud networking drivers, the same method applies to most network virtualization solutions deployed by commercial cloud providers.

Identical MAC Addresses for trunking ports. The first bug appears in the configuration shown in Fig.2: the tenant defines a topology with two VMs connected via 2 networks (red and green). Machine A is connected via a trunk port to both the red and green networks while machine B is only connected to the green network. The tenant-level expectation for this topology is that all packets from B towards A reach their destination. However, if A uses the same MAC for both its trunked ports this prevents communication between A and B. This bug was reported on the OpenStack Neutron bug tracker [4]. `netdiff` found a number of failed states which indicated that all packets leaving machine B were being dropped in the forward path by an anti-spoofing rule in the *br-int* bridge connecting the two machines.

Allowed address pairs bug. An allowed address pair is an extra IPv4, MAC address pair specifically tailored to allow bridging at VM level. Thus, the expected tenant-level behavior is that traffic with destination addresses in the list of allowed address pairs for a VM be allowed on egress. [5]

However, an implementation bug in the firewall module stops traffic from getting through. Thus, traffic issuing from VM A towards VM B is correctly forwarded to VM B, but the reverse traffic from VM B to VM A is dropped. The issue is correctly traced by `netdiff` which indicates failed (non-equivalent) states between the tenant and the provider perspectives. In the tenant view, A and B have bidirectional connectivity, whereas in deployment connectivity is broken. `netdiff` correctly captures the error in the reverse packet run and successfully identifies the offending rule.

No firewall enforcement on ICMP Type/Code. Security group support for ICMP filtering was not implemented for older versions of Neutron [6]. `netdiff` showed how ICMP traffic that is meant to be blocked is allowed in the dataplane.

Filtering with security groups. Security groups are collections of ACL rules that apply to all VMs part of that group. When specifying connectivity outside the group, tenants can use prefixes or remote security groups to specify the external source of traffic. There was a bug in the implementation of filtering when remote security groups where used. In our setup, we had two groups called *green* and *blue* and a rule that all traffic from the *green* group should reach the blue group [7]. We instantiated three VMs: A in the *blue* group, B in the *red* group, and C in both. At runtime, C could not be reached by neither A or B, violating the tenant configuration.

Inconsistent connections in the tracker. This bug appears when connectivity is repeatedly enabled and disabled for the same host-pair. We ran our test with VMs A and B, and the tenant allows traffic from A to B. In Neutron, all ACL rules are directional, and they are implemented using the connection tracker. In this case, A can initiate connections to B and B can respond, but B cannot initiate a connection to A.

After instantiation, A starts a connection to B which creates per-connection state in the conntrack module. Immediately afterwards, the tenant disallows traffic from A and B, which marks the connection in the conntracker as "dead" but does not delete the conntrack entry. When packets of the same connection reach conntrack, they will be dropped as expected. The problem appears when the tenant re-enables traffic from A to B: the flow entry mark is not cleared, and all subsequent packets are incorrectly dropped. [8] We catch this bug by using symbolic conntrack state. The incorrect behavior is captured and reported, highlighting the offending rule and the conntrack conditions which trigger the behaviour.

`netdiff` captures the bug in less than 2 minutes in our simple deployment, but larger scale experiments indicate that using symbolic state variables signficantly increase execution time. This is why we typically use an initially empty conntrack state for all servers in the topology.

A network isolation bug was discovered solely using `netdiff` and was reported as a Neutron bug [9]. In this setup, we have two machines running on the same host as in Figure 3, each connected to distinct VLANs. Assume that B is completely isolated from the rest of the network. Then, the expected behavior at tenant level is that no traffic from B can ever reach A. Next, assume host A is part of a permissive security group whereby ingress HTTP traffic is allowed; further assume that B knows A's MAC and IP addresses. Then, HTTP traffic from B will reach A, breaking isolation. This bug exists even when B belongs to a different tenant.

`netdiff` successfully detects the erroneous behavior, providing a packet from B that can reach A. To validate the bug, we successfully reproduced the behavior in the deployment. We note that the bug is difficult to catch with standard testing because ARP traffic was correctly blocked, and a simple `http-ping` from B to A would fail. Because `netdiff` uses symbolic packets, it finds a valid packet which will reach B.

Old Linux Kernel. This is a configuration bug that we stum-

bled upon when deploying Openstack in our testbed. Neutron's OVS adapter needs kernel support to access the Netfilter's conntrack module; support exists since version 4.3.

In our deployment, we had a compute node with kernel version 4.2. We deployed two VMs with security groups to allow all traffic between them. However, since there is no kernel support for connection tracking (as required by the firewall module), the insertion of security rules silently fails, and all traffic is dropped. `netdiff` caught this behavior by reporting successful execution at tenant level while the same input packet was dropped in the deployed dataplane.

Tunnel endpoint listening on *localhost*. The issue arises when Puppet, a provisioning tool, erroneously binds a tunnel endpoint on a compute node to the *localhost* address. The effect is that the VMs hosted on the affected compute node will not be able to communicate with VMs running on different compute nodes when in principle they should.

Hosts behind a NAT reachable from the outside. This issue was highlighted solely by running `netdiff` in a public-private network scenario. *Tenant A* creates a private virtual network and connects it to a public network via a virtual router, configuring source NAT on the external gateway of the router. He then deploys some virtual machines within his own private network and enrolls them in a permissive security groups allowing all ingress traffic. *Tenant B* also creates a virtual machine that he plugs directly into the public network and manages to find the IP address of the router's external gateway and the VM's private address.

Because the router is configured in SNAT mode, *A* would expect that no traffic from outside his network can initiate connections to any of his machines. However, Neutron virtual routers perform routing between the external and the internal network regardless and thus *B* can reach both *A*'s VMs.

ARP spoofing. Assume a VM responds to an ARP request by stating that the queried IP address can be found at a different MAC than the one defined on the VM's interface. Neutron's network abstraction asserts that no spoofing should be possible. Spoofing prevention is implemented by having the integration bridges [10] perform port-based checking on ARP replies to ensure that IPv4 addresses cannot be modified (ARP_SPA field) and all L2 frames coming from the VM have expected L2 source addresses.

However, no explicit check is performed to ensure that the advertised L2 address (ARP_SHA) is the expected one. Thus, a malicious or corrupt ARP implementation in a VM may successfully transmit spoofed ARP pairs.

Unexpected interactions with libvirt. The following configuration bug arises when deploying *libvirt*-based NAT networking alongside Neutron's iptables-based security groups mechanism. The libvirt toolset automatically creates a default virtual network with some prefix *P* (usually 192.168.122.0/24) and installs a series of iptables rules in the NAT table, POSTROUTING chain in order to perform address translation for outgoing VM traffic. This issue was discovered in one of the production settings we evaluated.

Say a tenant creates a virtual network with prefix *P*, then all outgoing traffic from a VM in this network to other networks will be dropped. `netdiff` quickly discovers a non-equivalence whenever the IP source address is in *P* and the destination address is not in *P*. We reproduced the bug, validated the model in our testbed and discovered that indeed packets issued from network A were being NAT-ed due to the unwanted interference with the rules generated by libvirt.

Troubleshooting VM connectivity. A common configuration issue confirmed in production settings appears in tenant networks with many security groups. The tenant wishes to enable communication between two of its VMs but mistakenly adds them to different security groups (the groups may have similar names). By default, security groups are configured such that they allow ingress traffic from machines belonging to the same group, but not from other security groups. After deployment, the user notices that no traffic flows between its VMs.

Troubleshooting connectivity problems is difficult for tenants, as it requires manually checking the ports of a given VM, and the security groups which they belong to. With `netdiff`, the administrator is able to quickly assess that the tenant and provider perspectives are identical. Thus, there must be a misconfiguration at tenant level which does not meet the user's expectation. Symbolic execution of the tenant topology indicates that all failed outcomes are due to an ingress security group which is not matched at B's level.

Iptables optimizations. Llorente et al. [28] aim at shortening packet processing pipelines in order to enhance performance of Neutron's iptables driver. We set out to check whether the optimization algorithm works correctly on a few inputs, i.e. it preserves the same packet processing behavior.

To achieve this, we deploy a one-node OpenStack deployment with 9 running VMs connected to 9 different security groups. Since all iptables optimizations are localized within the VM access Linux bridges, we only test equivalence at this component level. `netdiff` takes an average of 9.4s per VM to show that the optimization algorithm doesn't break any of the underlying logic. The result shows little performance degradation with respect to normal symbolic execution of the same deployment using Symnet (7.3s).

6.2 Checking a large Neutron deployment

In order to test `netdiff`'s scalability and discover novel bugs, we used a snapshot from our department's Openstack deployment. It consists of 87 compute nodes, running a total of 243 virtual machines. The deployment contains 14960 iptables and 11375 openflow rules which implement the 17 tenant-administered virtual networks and 2 public networks.

In the previous section, `netdiff` caught Neutron bugs in seconds by checking equivalence of tenant-level and

Network	# ports in network				
	1	3	26	57	164
Virtual (s)	0.03	0.03	0.04	0.02	0.02
Physical (s)	0.23	0.14	0.14	1.2	3.2
`netdiff` (s)	2.1	1.65	2.28	12.8	31.8

Figure 4: Neutron L2 reachability

Security group	# rules in security group ACL				
	2	6	12	13	19
Virtual (s)	0.08	0.08	0.09	0.08	0.09
Physical (s)	0.12	1.09	1.51	2.48	1.57
`netdiff` (s)	0.24	1.74	6.87	8.51	2.47

Figure 5: Ingress security groups

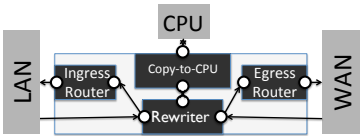


Figure 6: A modular P4 NAT

provider-level network models. In the departmental deployment this approach is not feasible. For instance, when testing L3 unicast reachability, `netdiff` failed to finish processing due to exponential state explosion, a common issue of symbolic execution. First of all, the generated SEFL code was introducing a lot of useless branching in lookup tables - such as ARP tables. We used a state-merging technique to mitigate this issue [24]; the results show a significant decrease in processing time in some scenarios, but increases in others - whenever the number of elements in the table is small.

To further tame the complexity we used a compositional approach: we test equivalence between corresponding parts of the two dataplanes instead. For Neutron we tested three equivalence checkpoints detailed below. As the size of each component is small, the time to check equivalence is reduced to seconds or tens of seconds. Since our implementation is compositional by design, identifying distinct components and running `netdiff` against them entailed a small amount of extra work - cca. 200LOC.

L2 reachability equivalence can be checked by using a symbolic ARP request in both the tenant and provider network. In table 4, we show average execution times to check equivalence for layer 2 reachability using ARP broadcast probes, contrasted with plain symbolic execution in either network. As the number of paths to be explored grows with the number of ports size of the virtual network, equivalence checking time goes up from 2.1 seconds to around 30 seconds. To put the results in perspective, parsing all the configuration files and code generation take around 15s each.

Security group compliance. For each VM in the deployment we test if its implementation is compliant to the security groups (both ingress and egress) defined by the tenant. `netdiff` checks egress security groups quickly (200ms on average), and takes longer for ingress security groups (table 5) because tenants tend to use default-on egress policies.

Virtual routers must behave correctly with respect to their expected functionality including routing, floating IPs and source NAT. The physical implementation of a layer 3 router is well delimited within the boundaries of a Linux network namespace, so we can simply *clog* outgoing corresponding interfaces in both virtual and physical topologies and inject packets at input interfaces. Depending on the number of router interfaces, as well as on the number of floating IPs defined therein, the time for equivalence checking goes up to 80s for a router with 48 floating IPs.

6.3 P4 equivalence

P4 [5] is a high level language that enables programming dataplanes and can also be efficiently implemented in hardware. Despite its apparent simplicity, coding P4 programs is tricky: unexpected behaviors may be accidentally introduced during the design or runtime phase. In this section we show how `netdiff` can be used to determine behavioral equivalence between different P4 programs with seemingly identical dataplane configuration and functionality.

Monolithic NAT vs modular NAT. One of the simplest P4 tutorials is a NAT that includes three distinct pieces of functionality: a *NAT rewriter*, which simply sets packet fields to given mappings, two *routers*, one for the LAN and one for the WAN, each of which performs longest prefix matching, assigns next hop address and selects the proper output port and a *CPU redirector*, which encapsulates a packet and sends it to a control-plane application if no NAT mapping is found. Even for a simple set of table rules, understanding the interactions between these different pieces is difficult.

To check whether the functionality of our P4 NAT works correctly, we wish to compare it to a modular design that runs different functionality in separate P4 programs which are connected. We still prefer the monolithic approach for deployment because it is cheaper to implement and performs better at runtime than our modular design that serializes and de-serializes packets between the interconnected boxes.

In Figure 6 we show how the modular NAT works. We used Vera [40] to generate models for both NATs and used `netdiff` to check whether they are equivalent. In around 5s, `netdiff` shows that the implementation of the monolithic NAT is not equivalent to the modular implementation. In the monolithic implementation, it is possible to translate a packet intended for the LAN and then send it on the LAN interface. The same behavior is not possible with the modular NAT because the routing tables corresponding to LAN and WAN networks are split into 2 distinct routers - one for *ingress* and one for *egress*.

Which is the correct order of table application? In our next example, we take the simple router P4 tutorial and we enhance it to enable ACL processing. Assume that the P4 programmer initially instructs the ingress pipeline to first route packets and then apply ACL. In a subsequent run, the programmer decides to reverse the order in which the two tables are applied in order to avoid routing packets that would be dropped by the ACL; this approach seems more efficient. In our program, the ACL table has two actions: one that drops packets and one that passes them through (see Fig. 7).

When we compare the two programs with `netdiff`, it is

ACL table:

```
table acl {
  reads { ipv4.srcAddr : exact; }
  actions { _drop; _nop; }
}
```

Entry:

```
table_add acl 10.0.0.3 _drop
```

Attempt 1:

```
apply(ipv4_lpm);
apply(acl);
```

Attempt 2:

```
apply(acl);
apply(ipv4_lpm);
```

Figure 7: Ways of adding an ACL to a P4 router.

surprising to find they are different. The first version matches our expectation that unwanted packets (10.0.0.3) are indeed dropped. The second version surprisingly allows all packets through. This is because the packet is not actually dropped in the ACL table. The P4 spec states that a drop action within the ingress pipeline only marks the packet for rejection and continues execution from that point on. When the dropped packet hits the ipv4_lpm table, the default action sets the egress spec to that of a valid interface, reviving the packet.

Trimming switch.p4 to size Recent work on verifying P4 programs [27, 40, 32, 10] highlights the difficulty of programming correct P4 programs. Instead of writing software from scratch. network operators can use existing catch-all implementations and adjust them to their needs. An example is `switch.p4` [45] which provides a full implementation of production-ready ToR switch. switch.p4 contains 131 tables and a total of 6KLOC; deploying this program is wasteful when one does not need all the functionality therein. If fewer tables are synthesized, they can hold more match-action rules allowing scope for specialization.

We used `netdiff` to help us trim switch.p4 while maintaining equivalence for IPv4 processing. We began with a working configuration of switch.p4, including concrete table entries for all entries. We then iterated by (1) removing functionality irrelevant for basic IPv4 routing and (2) generating the SEFL model for the resulting program using Vera's translator [40] (3) checking equivalence to the full program. In all our tests, equivalence testing took between between one and two minutes, depending on the number of table entries.

We confirmed that much processing was not needed for the correct functioning of v4 routing and can be safely removed: IP sourceguard, QOS processing, sFlow, Integrated services, Storm control, MPLS, etc. Removing these from the source code significantly reduces the total size of the tables (e.g. only sourceguard had space saved for 1500 entries, and INT processing makes up for 10% of the LOC in switch.p4). On the other hand, setting ingress port mappings, validating the outer header, handling VLANs, as well as the obvious IPv4 processing (longest prefix match, reverse path forwarding checks, etc.) are needed for correct functioning.

6.4 Monitoring FIBs in a production network

For the last six months we have been using `netdiff` to monitor routing in our university's network. For each of the 9 routers that make up the network core, we have have taken a FIB snapshot every 6 hours, and then checked the equivalence between FIB snapshots from the same router. We aim

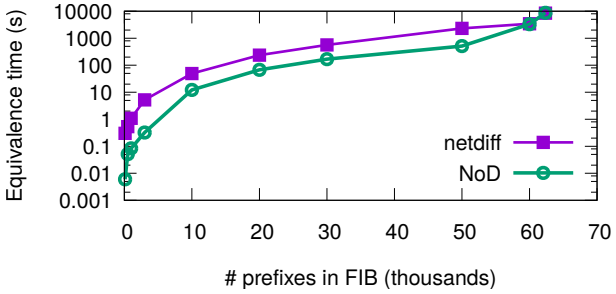


Figure 8: Router equivalence checking: `netdiff` vs. NOD

to help our admin understand how routing changes over time. The snapshots vary in size from about 700 entries up to 20K entries for the core router. Equivalence takes 100ms to compare the smallest FIBs, and up to 50s for the largest ones.

Only in rare cases (about 5%) two snapshots of the same router FIB are equivalent. However, most of the time, the differences were due to churn in directly connected hosts (92%). `netdiff` did uncover interesting differences: a few routers had empty FIBs after recovering from a failure (1%) or were missing certain routes due to link-failures (2%).

6.5 Is my datacenter network one big switch?

Fat-trees [2] are the de-facto standard datacenter network, and they aim to provide a *big-switch* abstraction to end-hosts. We use `netdiff` to check whether the myriad of interconnected switches is equivalent to a single switch to which end-hosts are directly connected. We used Batfish [8] to generate the data planes for a fat-tree network with 125 switches (50 edge, 50 aggregation and 25 core switches). The question we asked is whether every port of every edge device behaves as if it was a port of the corresponding big-switch.

The verification procedure generates SEFL models for both the fat-tree and the big switch from the FIBs given by Batfish. We then run equivalence by injecting a packet with symbolic destination address into equivalent server-facing switch ports in the two topologies; the check takes around 4 minutes.

`netdiff` found that the two models are not equivalent: every edge switch had at least one /32 prefix which wasn't advertised to the core (due to the configured routing policy), rendering the prefix unreachable from different edges. This contradicts the big switch assumption.

6.6 Scalability

Our experiments so far have highlighted the usefulness of `netdiff`, which works well in practice despite its poor theoretical complexity (§4) given by the exponential nature of symbolic execution. To better understand `netdiff`'s performance, we compare it against Network Optimized Datalog [29] when testing equivalence of two routers with small to medium-sized routing tables.

We generate NOD rules from router FIBs and measure the total equivalence checking time. We run tests involving reachability analysis and tests involving input, port and functional equivalence. Functional equivalence tests with NOD ran out of memory, even for small inputs.

Figure 8 shows the runtime of both tools against the number of entries in the FIB. Note that the NOD line corresponds to input and port equivalence; the `netdiff` line corresponds to full equivalence checking (Definition 2.1). As expected, the runtime grows exponentially for both NOD and `netdiff`. For unicast reachability, NOD proves cca. an order of magnitude faster than `netdiff`, but the difference diminishes for larger inputs; for the largest FIB `netdiff` is faster than NOD. Overall,these results show that `netdiff`'s more powerful equivalence comes at modest runtime costs.

Overhead breakdown. We also measured the time taken by each of `netdiff`'s components: we measured separately the symbolic execution for M_1 and M_2, their sieving (algorithm 2), and finally the equivalence testing time (Algorithm 3). The results show that M_2 symbex time grows linearly with the number of output path conditions from M_1. However, M_2 sieving time is constant, due to the fact that the number of feasible outcomes from M_2 is constant - most often equal to one in case when M_1 and M_2 are very similar. Finally, equivalence testing time is negligible.

We also noticed that for broadcast packets, our sieving algorithm is faster since the number of solver queries is linear in the number of output packets in the network. To reduce the sieving time, we disable it for unicast packets.

7 Related work

Our observation that equivalence checking is a simple form of specification is not novel: it has been used previously for program regression verification [44, 36] and to check compiler correctness [23, 12], among other applications.

Note that, in contrast to compiler verification, which attempts to show that compilation preserves semantic equivalence on all possible source programs, `netdiff` limits its scope to only showing equivalence between concrete dataplane snapshots - i.e. a single source program.

There exist a wide range of specification languages and verification tools for network dataplanes; we discuss here the ones not covered in section 2. Margrave is a tool that checks firewall configurations against user-specified policies in first-order logic [31]. Anteater [30] translates networks and reachability queries to SAT formulae, while NetPlumber [18] takes as input a graph and network boxes modeled as bitwise transfer functions, and uses HSA [19] to check for compliance. Finally, NetCheck [35] takes specifications written in CTL and uses symbolic execution with Symnet to check them. All these tools have merits, yet one of their biggest problems is the difficulty of specifying what the network is meant to do. In many cases, the spec underspecifies the behavior, meaning that potential problems are missed.

Another line of work focuses on more rigorous specifications which are first proven correct and then translated to dataplane rules. Examples here include Kinetic [21] which takes Finite State Machine descriptions of network functionality, FatTire [37] that takes regular expressions specifying paths to be taken by packets, and Cocoon [38] which enables iterative design and specification for networks. All of these tools offer much stronger correctness properties, but this comes at the expense of usability by non-experts. `netdiff` is complementary to the above in that it may serve as an extra validation step.

In programming languages, equivalence testing is not a novel concept. DECKARD [14], CCFinder [17] and [25] P-Miner look for syntactically similar pieces of code that are equivalent. EQMINER [15] detects functionally equivalent code via random testing but does not offer guarantees that two programs are equivalent because it does not cover all possible test cases. Another work that aims to achieve the same goal with symbolic execution, targets functional equivalence for simple arithmetic functions, in code that has no branches [13]. Neither tool is exhaustive, so they do not offer correctness guarantees. Our work aims to decide whether two network dataplane models process the packet in the same way, a much stronger definition of equivalence in the limited context of programmable dataplanes.

8 Conclusions

Checking equivalence of programmable dataplanes is a simple way to check program correctness or verify policy. We have presented `netdiff`, an algorithm that checks two network dataplanes for equivalence using symbolic execution. `netdiff` will be open-sourced soon.

We have used `netdiff` to uncover three previously-unknown Openstack Neutron bugs and four configuration errors. `netdiff` can be used to check P4 programs too: we have found bugs even in simple P4 programs, and have shown how `netdiff` can be used to help trim large P4 programs while preserving desired functionality. Overall, we find that while equivalence checking is more expensive than individual symbolic execution of the two programs, it scales well enough for most use-cases; compositional equivalence can be used to scale to large Neutron deployments.

In future work, we intend to further explore the applicability of `netdiff`. One particularly interesting avenue of research is to check the equivalence between SEFL models and the actual dataplane (in C), which requires integrating different symbolic execution engines (Symnet and Klee).

Acknowledgements

This work was funded by CORNET H2020, a research grant of European Research Council (no. 758815).

References

[1] Symnet Source Code Repository. `https://github.com/nets-cs-pub-ro/Symnet/`.

[2] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *Proceedings of ACM SIGCOMM 2008*.

[3] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. Netkat: Semantic foundations for networks. In *POPL'14*.

[4] BECKETT, R., GUPTA, A., MAHAJAN, R., AND WALKER, D. A general approach to network configuration verification. In *SIGCOMM* (2017).

[5] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014).

[6] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *Proc. TACAS'08*.

[7] DOBRESCU, M., AND ARGYRAKI, K. Software dataplane verification. In *Proc. NSDI'14*, NSDI'14.

[8] FOGEL, A., FUNG, S., PEDROSA, L., WALRAED-SULLIVAN, M., GOVINDAN, R., MAHAJAN, R., AND MILLSTEIN, T. A general approach to network configuration analysis. In *NSDI* (2015).

[9] FOSTER, N., KOZEN, D., MILANO, M., SILVA, A., AND THOMPSON, L. A coalgebraic decision procedure for netkat. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2015), POPL '15, ACM, pp. 343–355.

[10] FREIRE, L., NEVES, M., LEAL, L., LEVCHENKO, K., SCHAEFFER-FILHO, A., AND BARCELLOS, M. Uncovering bugs in p4 programs with assertion-based verification. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2018), SOSR '18, ACM, pp. 4:1–4:7.

[11] GEMBER-JACOBSON, A., VISWANATHAN, R., AKELLA, A., AND MAHAJAN, R. Fast control plane analysis using an abstract representation. In *SIGCOMM* (2016).

[12] GUO, S.-Y., AND PALSBERG, J. The essence of compiling with traces. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 563–574.

[13] HIETALA, K. Detecting Behaviorally Equivalent Functions via Symbolic Execution, 2016.

[14] JIANG, L., MISHERGHI, G., SU, Z., AND GLONDU, S. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering* (Washington, DC, USA, 2007), ICSE '07, IEEE Computer Society, pp. 96–105.

[15] JIANG, L., AND SU, Z. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (2009), ISSTA '09.

[16] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 103–115.

[17] KAMIYA, T., KUSUMOTO, S., AND INOUE, K. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng. 28*, 7 (July 2002).

[18] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *Proc. NSDI'13*.

[19] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *Proc. NSDI'12*.

[20] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. Veriflow: Verifying network-wide invariants in real time. In *Proc. NSDI'13*.

[21] KIM, H., REICH, J., GUPTA, A., SHAHBAZ, M., FEAMSTER, N., AND CLARK, R. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 59–72.

[22] KING, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (July 1976), 385–394.

[23] KUNDU, S., TATLOCK, Z., AND LERNER, S. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 327–337.

[24] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2012), PLDI '12, ACM, pp. 193–204.

[25] LI, Z., LU, S., MYAGMAR, S., AND ZHOU, Y. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (2004), OSDI'04.

[26] LIU, H. H., ZHU, Y., PADHYE, J., CAO, J., TALLAPRAGADA, S., LOPES, N. P., RYBALCHENKO, A., LU, G., AND YUAN, L. Crystalnet: Faithfully emulating large production networks. In *Proc. of the 26th Symposium on Operating Systems Principles (SOSP)*.

[27] LIU, J., HALLAHAN, W., SCHLESINGER, C., SHARIF, M., LEE, J., SOULÉ, R., WANG, H., CAŞCAVAL, C., MCKEOWN, N., AND FOSTER, N. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 490–503.

[28] LLORENTE, J., AND MAEL, K. Neutron firewall optimizations. `https://github.com/jllorente/neutron-firewall-optimization`.

[29] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking beliefs in dynamic networks. In *Proc. NSDI'15*.

[30] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P. B., AND KING, S. T. Debugging the data plane with anteater. In *Sigcomm* (2011).

[31] NELSON, T., BARRATT, C., DOUGHERTY, D. J., FISLER, K., AND KRISHNAMURTHI, S. The margrave tool for firewall analysis. In *Proceedings of the 24th International Conference on Large Installation System Administration* (Berkeley, CA, USA, 2010), LISA'10, USENIX Association, pp. 1–8.

[32] NÖTZLI, A., KHAN, J., FINGERHUT, A., BARRETT, C., AND ATHANAS, P. P4pktgen: Automated test case generation for p4 programs. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2018), SOSR '18, ACM, pp. 5:1–5:7.

[33] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, 2015), USENIX Association, pp. 117–130.

[34] PONTARELLI, S., BIFULCO, R., BONOLA, M., CASCONE, C., SPAZIANI, M., BRUSCHI, V., SANVITO, D., SIRACUSANO, G.,

CAPONE, A., HONDA, M., HUICI, F., , AND BIANCHI, G. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), USENIX Association.

[35] POPOVICI, M. Verifying large-scale networks using netcheck. In *2017 European Conference on Networks and Communications (EuCNC)* (June 2017), pp. 1–5.

[36] RAMOS, D. A., AND ENGLER, D. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., 2015), USENIX Association, pp. 49–64.

[37] REITBLATT, M., CANINI, M., GUHA, A., AND FOSTER, N. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking* (2013), HotSDN '13.

[38] RYZHYK, L., BJØRNER, N., CANINI, M., JEANNIN, J.-B., SCHLESINGER, C., TERRY, D. B., AND VARGHESE, G. Correct by construction networks using stepwise refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 683–698.

[39] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., MCKEOWN, N., AND LICKING, S. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 15–28.

[40] STOENESCU, R., DUMITRESCU, D., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 518–532.

[41] STOENESCU, R., DUMITRESCU, D., AND RAICIU, C. Openstack networking for humans: Symbolic execution to the rescue. In *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)* (June 2016), pp. 1–6.

[42] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. SymNet: scalable symbolic execution for modern networks. In *SIGCOMM* (2016).

[43] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Tech report: Debugging p4 programs with vera. Tech. rep., June 2018.

[44] STRICHMAN, O., AND GODLIN, B. *Regression Verification - A Practical Way to Verify Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 496–501.

[45] THE P4 CONSORTIUM. A p4 implementation of a tor switch. `https://github.com/p4lang/switch/blob/master/p4src/switch.p4`, 2018.

[46] ZAOSTROVNYKH, A., PIRELLI, S., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. A formally verified nat. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 141–154.

A Correctness of `netdiff`

Note the following useful properties of symbolic execution:

$$\forall (p_i,\pi_i) \in DataplaneSymbex(M,k,p), S(p_i) \subseteq S(p) \quad (1)$$

$$\forall (p_i,\pi_i),(p_j,\pi_j) \in DataplaneSymbex(M,k,p) i \neq j, \; S(p_i) \cap S(p_j) = \emptyset \quad (2)$$

$$\bigcup_{(p_i,\pi_i)\in DataplaneSymbex(M,k,p)} S(p_i) = S(p) \quad (3)$$

The notation (p,π) denotes a pathset, where p is a predicate - which we refer as path condition, describing a subset in the input packet space, and π is a set of paths with the same path condition. $S(p)$ is the set of packets described by predicate p.

Lemma 1 *The set of pathsets computed by algorithm 2 satisfies the symbolic execution properties 1, 2, 3.*

Proof: Property 1 is satisfied by the design of symbolic execution. We need to prove that the set of pathsets built by the algorithm 2 satisfies properties 2 and 3. Line 2 of the algorithm creates the set Q that contains all the pathsets obtained by running symbolic execution with the symbolic packet described by predicate p_0, not necessarily obeying property 2. To reinforce this property, we build a new set L, that will contain only the pathsets with disjoint path conditions. We will prove the following invariant holds each time the algorithm reaches line 14: the set L satisfies 2 and 3.

We iterate through all the pathsets (q,π) in Q and (l,s) in L, lines 4 and 5, and check for overlapping path conditions. We replace the overlapping pathset in (l,s) in L with two pathsets: one that adds the union of paths s and π with the overlapping path condition $(q \wedge l)$ and one that keeps the pathset (l,s) with the non-overlapping path condition $(l \wedge \neg q)$, lines 7 and 8. Based on set theory, these operations keep the invariant for the local iteration. Line 9 guarantees that the invariant is satisfied for all iterations through L, by updating the pathset (q,π) to $(q \wedge \neg l,\pi)$. If after iterating through all pathsets in L the path condition q is non empty then we add it to L, line 13, reinforcing property 3. □

Theorem A.1 EQUIVALENCE(M_1,M_2,i_1,i_2,p_0) *is true iff* M_1, M_2 *are equivalent w.r.t.* $Q = \{x \in Packet | p_0(x) = true\}$ *and input ports* i_1, i_2.

Proof: We show that when EQUIVALENCE(M_1,M_2,i_1,i_2,p_0) is true, there is a bijection χ mapping each successful *located* packet produced by M_1 to one produced by M_2, c.f. Def. 2.2.

In order to determine if such a bijection can be built, we symbolically execute the program M_1 for the input port i_1 and a symbolic packet specified by predicate p_0. p_0 describes the set of packets Q for which we decide the equivalence of programs c.f. Def. 2.2. The result is a set of pathsets (q_{i1}, π_{i1}). According to the symbolic execution properties listed above, each path condition q_{i1} implies the initial path condition p_0 (Property 1), the sets of packets described by the path conditions are disjoint (Property 2) and their union over all path conditions is the set of packets specified by p_0 (Property 3). For each pair (q_{i1}, π_{i1}) of M_1 we symbolically execute program M_2 with path condition q_{i1} and decide on the equivalence of paths.

For a path condition M_1 there might be several pathsets (q_{j2}, π_{j2}) of M_2. The main point is that the symbolic execution property 3 holds, therefore the set of packets described by condition q_{i1} is the union of the sets of packets described by conditions q_{j2} over all j. Consequently, equivalence must hold between (q_{i1}, π_{i1}) and each pathset (q_{ij2}, π_{ij2}) over all j, under the constraints imposed by $\mathscr{R}$ and ω. To decide their equivalence we must look into the pathset definition.

The above essentially means that $\forall p \in S(p_0).\exists! q_{ij2} \; s.t. \; p \in S(q_{ij2})$ (because of property 2). Thus, if the bijection condition holds true for all q_{ij2}, then it holds for their union. But due to property 3, $\bigcup_{(i,j)} S(q_{ij2}) = S(p_0)$, which implies that the bijection can be found on all input packets.

A pathset is a set of input packets and located output packets. Two pathsets are equivalent if (i) they have the same cardinality, (ii) the ports are in correspondence c.f. relation $\mathscr{R}$ and (iii) the packet headers on the corresponding ports satisfy the relation ω c.f. Def. 2.2 in the context of the current path condition. Our approach consists in reducing the equivalence decision problem to that of the maximum bipartite matching (MBM). The conditions of the MBM define a bijection between workers and jobs, which maps to a bijection χ between programs' outcomes in our case. We need to prove that our equivalence algorithm implements the conditions of the MBM problem, therefore deciding on the existence of the bijection.

The first condition is satisfied since the ports are different being defined in the namespace of each program. The second condition reinforces the equality of the cardinalities of the sets of workers and jobs, O_1 and O_2 in our case. We verify it in the line 3 of the algorithm 3. The next condition imposes that worker i is qualified for job j, meaning that we can create the edge representing the equivalence between outcomes described as pairs of (port, packet). The association between ports is checked in line 5 of algorithm 4. Line 7 insures that the relation between packets in the context of the current path condition hold. The existence of the bijective association between programs' outcomes is checked by the algorithm *MaxBipartiteMatching*. The outcome of the MBM is true iff the two path are equivalent. □

B Notes on equivalence

An equivalence relation is a binary relation satisfying the reflexivity, symmetry and transitivity conditions. The core of our definition of equivalence 2.2 is the bijection between sets of packet and output port pairs, meaning that equivalence conditions are satisfied. The algorithm `netdiff` determines if a bijection can be computed, therefore verifying the equivalence. It is worth mentioning that two dataplane programs written in SEFL have their own name spaces therefore the algorithm `netdiff` can be applied to check equivalence between a program and itself.

C Notes on complexity of `netdiff`

The complexity of `netdiff` depends strongly on the complexity and number of pathsets output by *DataplaneSymbex*. First of all, we take into account the time of the symbolic execution of the first program (line 2 in Alg. 1). Assume that the number of pathsets produced as a result is n. Similarly, the number of pathsets produced by executing line 5 is m. Therefore, the complexity is $C(DataplaneSymbex_1) + n \cdot (C(DataplaneSymbex_2) + m \cdot C(EQP))$

Now, we turn our attention to computing the complexity of *EQP*. The dominating operation is represented by computing the adjacency matrix of the correspondence graph. This involves at most p^2 calls to the SMT solver, where p is the maximum number of paths in each pathset.

The total complexity of *EQP* is: $C(DataplaneSymbex_1) + n \cdot \big(C(DataplaneSymbex_2) + m \cdot p^2 \cdot C(SMT)\big)$

Since path conditions are usually simple and path length through the network is not large, we assume the complexity of invoking the SMT solver for one path condition to be constant in the size of the network.

Notice that even though the networks under equivalence test may be of similar size, the complexity of the first symbolic execution is considered significantly larger, especially when networks exhibit a high degree of similarity. This is because the path conditions coming out of the first symbolic execution reduce state explosion in the second. Even though in theory both n and m are exponential in the size of the network analyzed, we claim that in practice $m << n$. Therefore, we need to stress that the complexity of *EQP* is strongly dominated by the complexity of the symbolic execution of M_1 and the number of outcomes thereof.

Packet cloning is not widely used in the network dataplanes we have examined, which means that p is one or a small integer. Even when cloning is used, for instance in L2 processing, the specifics of the network forwarding fabric constrain p to be smaller than the number of terminals connected to the L2 segment - which is evidently much smaller than the size of the entire network.

Notes

[1] After taking pains to actively test it.

[2] it requires enumerating all possible values in the field range

[3] a join between all possible input packets and output packets is used to model a router

[4] https://bugs.launchpad.net/neutron/+bug/1626010

[5] https://bugs.launchpad.net/neutron/+bug/1697593

[6] https://bugs.launchpad.net/neutron/+bug/1708358

[7] https://bugs.launchpad.net/neutron/+bug/1708092

[8] https://bugs.launchpad.net/neutron/+bug/1715789

[9] https://bugs.launchpad.net/neutron/+bug/1736739

[10] which aggregate traffic from all machines on a compute node