Network Verification in the Light of Program Verification

Nuno P. Lopes
INESC-ID / IST U. Lisboa

Nikolaj Bjørner
Microsoft Research

Patrice Godefroid
Microsoft Research

George Varghese
Microsoft Research

Abstract

The fastest tools for network reachability queries use ad-hoc algorithms to compute all packets from a source *S* that can reach a destination *D*. This paper examines whether network reachability can be solved efficiently using existing verification tools. While most verification tools only compute reachability ("Can S reach D?"), we efficiently generalize them to compute all reachable packets. Using new and old benchmarks, we compare model checkers, SAT solvers and various Datalog implementations. The only existing verification method that worked competitively on all benchmarks in seconds was Datalog with a new composite Filter-Project operator and a Difference of Cubes representation. While Datalog is slightly slower than the Hassel C tool, it is far more flexible. We also present new results that more precisely characterize the computational complexity of network verification. This paper also provides a gentle introduction to program verification for the networking community.

1 Introduction

Modern networks are complex and contain multiple devices such as routers, bridges, firewalls, and load balancers each of which forward packets differently based on different packet fields. Device diversity together with erroneous manual entry often leads to bugs that cause large downtimes in which valid packets are dropped, or invalid packets are let through [16, 18]. For example, an erroneous firewall rule might only drop certain packets.

Much recent work has focused on static checkers that find violations of assertions (forwarding bugs) by inspecting router tables and configuration files, assuming the routing protocol is correct. Data plane static verification began with Xie et al. [24] who computed a formula for reachability between a source and a destination while handling IP forwarding and ACLs. Later, Anteater [18] used SAT solving techniques to evaluate a reachability formula and applied their analysis to the UIUC network to find black holes, loops, and router bugs. Independently, Al-Shaer [1] represented network routing tables as BDDs and computed reachability predicates using model checking.

Later, the Header Space Analysis (HSA) technique [16] moved beyond finding reachability predicates (e.g., "Can A reach B") to reachability sets ("What are all the packets from A that can reach B") and for arbitrary protocols. Reachability sets are important for two reasons: *incremental computation* and *intelligibility*. Veriflow [17] and NetPlumber [15] showed techniques to incrementally compute reachability and reachability sets for Anteater [18] and HSA [16], respectively. Incremental computation is useful because when a manager adds a new rule (e.g., an ACL), the tool can check in real-time for bugs; both Veriflow and NetPlumber leverage the fact that a single rule change does not significantly change the underlying "network state machine". By remembering the reachability sets before the change, only small modifications need to be made to incorporate the rule change. Computing reachability *sets* seem useful for any incremental approach.

Second, reachability sets provide greater intelligibility for managers. If an ACL erroneously drops all packets sent to the prefix $128.55.0.0/16$, the reachability predicate can fail with a counterexample such as $128.55.8.7$. Much more insight can be gleaned if the tool outputs the *set* of packets being dropped (i.e., packets with destinations that match $128.55.*.*$ which suggests the ACL bug more directly). HSA represents packet headers by ternary strings drawn from $1, 0, \star$ where the wildcard character is used to abstract header bits that are irrelevant to forwarding. HSA computes reachability sets by propagating these ternary strings through each router or device modeled as a state machine that transforms and forwards ternary strings.

Zhang et al. [27] provide an excellent survey of existing work from the lens of program verification, and make the following observations. Network verification is essentially state machine verification because each router is a state machine. Reachability in a finite network of finite state machines is, in general, PSPACE-complete. HSA is akin to ternary symbolic simulation in hardware verification, which has been abandoned because of large growth in the state space. The techniques used in Anteater [18] and HSA [16] work well in practice because there are structural properties of networks that make the problem closer to being NP-complete. Structural properties conjectured in [27] include small depth (small diameter, TTL limitations), and the fact that the network transition function operates on comparatively few header bits.

While these observations [27] are intuitively appealing, there is currently no *quantitative insight* on the relation between network verification and standard verification. For example, there is no precise definition of the structural network parameters that impact complexity or a study (either analytic or experimental) of how the hardness of reachability varies with such parameters. Further, [27] and its successor [26] focus on reachability predicates and not reachability sets; as we have seen, reachability sets are crucial for incremental analysis and rapid debugging. For reachability sets, the field of program verification has a less rich suite of well-established tools. Reachability sets are akin to the AllSAT problem (all satisfying assignments) while reachability predicates are akin to SAT solving (one satisfying assignment). Modern SAT solvers are optimized for speed for SAT, not AllSAT.

The best techniques for static reachability verification [15, 17] are currently fast enough (microseconds) and scalable (to large networks such as the Google network). Why then should the networking community care about techniques from program verification? First, there is the scientific benefit; what precise aspects of networks make the problem different from verifying a hardware circuit or a program? Second, there is the need for a more comprehensive set of tools for networking beyond static checking including testing (e.g., [25]), new languages and synthesis [9], and control plane verification [8]. This vision for "network CAD tools" was articulated by McKeown [20]. Current hardware/software CAD tools leverage common verification tools such as theorem provers. Similarly, it seems prudent to leverage general purpose building blocks if network verification is to move forward from a set of ad hoc tools to an engineering discipline. Finally, establishing common terminology and common benchmarks, allows the verification and network communities to work together to build better tools for networks.

While the field of verification is vast, a few exemplars immediately seem relevant. A network is a large state machine [27]; state machine reachability has been successfully handled by model checking [1]. In particular, symbolic model checking [3] keeps track of sets of states using compact data structures (BDDs [2]); while most model checkers do not provide reachability sets it seems feasible to expose their internal data structures and repurpose them for reachability sets. How do BDDs compare with ternary representations [16, 17]? [1] Second, Datalog allows queries to be rewritten at a high level of abstraction, can handle recursion and rewriting in a natural way, and computes reachability sets out of the box. Third, while SAT/SMT solvers are fairly low-level, modern SAT solvers are scalable and flexible and have been used in past work [18, 26, 27]. If existing verification techniques can provide comparable functionality to say Veriflow [17] and NetPlumber [15], it seems easier to find a foundation to build further aspects of network CAD such as synthesis.

The combination of properties we wish for (reachability sets, incremental recomputation, and eventually automated synthesis) are on the *frontiers* of verification research today and considered hard problems — but ones that may have good solutions in specific domains. This is why verification researchers are interested in networking. But this dialogue requires a characterization of domain aspects that can be leveraged and agreement on standardized data sets and queries. Thus, our contributions are:

Complexity: We show that network verification problem is NP-complete if either there are no complex loops, no rewriting, or no cycles in the topology. Detecting complex loops is always NP-complete.

Computing Reachability Sets: While Datalog natively computes reachability sets, to scale to large networks we had to introduce a combined Filter-Project operator and a Difference of Cubes representation. We also show a way to adapt any SAT/SMT solver to compute reachability sets by leveraging bounded rewriting.

Benchmarks: We use a standard (Stanford) benchmark and introduce a new benchmark with parameters for rewriting, topology depth etc. that can be varied. We also introduce a standard set of queries in Datalog. All our benchmarks and code are public (see [21]).

Experiments: We compare various combinations of Datalog, model checking, and SMT solving in the Z3 theorem prover [6]. While Z3 provides only one implementation, other implementations should have comparable performance; the Z3 tool chain provides a uniform setting for comparing techniques.

Although our experimental results point to some clear insights, we do not claim to provide the final word on the topic. In fact, we hope that our paper and publicly available code and benchmarks will stimulate further work. We view bridging this gap between the network and ver-

[1]Note that [1] uses BDDs to encode router forwarding tables and not reachability sets.

$$\begin{tikzcd}
A \arrow[r] & R_1 \arrow[rr] \arrow[dr] & & R_2 \arrow[r] & B \\
& & R_3 \arrow[ur] \arrow[r] & D &
\end{tikzcd}$$

in	*dst*	*src*	*rewrite*	*out*
R_1	$10\star$	$01\star$		R_2
R_1	$1\star\star$	$\star\star\star$		R_3
R_2	$10\star$	$\star\star\star$		B
R_3	$\star\star\star$	$1\star\star$		D
R_3	$1\star\star$	$\star\star\star$	$dst[1] := 0$	R_2

Figure 1: R_1 has a QoS routing rule that routes packets from a video source on the short route and other packets to destination $1\star\star$ along a longer path that traverses R_3. R_3 has an ACL that drops packets from $1\star\star$. R_3 also rewrites the middle bit in *dst* to 0. This ensures that re-routed packets reach B regardless of the value of $dst[1]$.

ification communities as as another valuable contribution of this paper.

The paper is organized as follows. Section 2 formalizes network reachability. Section 3 provides complexity results. Section 4 surveys the basic tools we use. Section 5 shows how we scale our Datalog implementation. Section 6 describes a simple algorithmic adaptation of any SAT/SMT solver to compute reachability sets. Section 7 describes our benchmarks and Section 8 does performance comparisons.

2 Formal Model

While there are a variety of formalisms to model networks, we choose to use Datalog because our most efficient realization uses a Datalog implementation. We start with a simple example.

2.1 Reachability Set Example

Figure 1 shows a network with three routers R_1, R_2, and R_3, three end-point nodes A, B and D, and an associated reachability set question: what packets can reach B from A? The routing tables are shown below the picture. Packets are composed of two fields *dst* and *src*, each a bit-vector of 3 bits.

When there are multiple rules in a router, the first matching rule applies. For example, if a packet reaches R_1 and the first two bits in the destination address match 10 and the first two bits in the source IP address match 01, then the packet gets forwarded to router R_2. All other packets are handled by the next rules. For R_1, the next rule forwards packets whose first destination bit is 1 to router R_3. All other packets are silently dropped.

The last rule of Figure 1 includes a packet rewrite operation, as may happen in real networks. We use $dst[1] := 0$ to indicate that position $dst[1]$ is set to 0. (We assume position 0 corresponds to the right-most bit and so on.) Most router rules do not rewrite destination IP addresses but other fields such as MAC addresses. This small example is only for illustrative purposes.

The goal is to compute the set of packets that can reach from A to B. For this example, the answer is easy to compute by hand and is the set of 6-bit vectors

$$10\star01\star \cup (10\star\star\star\star \setminus \star\star\star1\star\star)$$

where each packet is a 6-bit vector defined by a 3-bit value for *dst* followed by a 3-bit value for *src*, $\star$ denotes either 0 or 1, and $\setminus$ denotes set difference.

2.2 Datalog Network Model

A network is a set of routers R_1, R_2, etc. and a set of end-points A, B, etc. that can send and receive packets. In what follows, routers and end-points will sometimes be referred to as *network nodes*. We model the header of a packet as a set of variables representing packet fields, each of which is a bit-vector. Our example used only one destination and one source fields. Both forwarding and rewriting operations are supported. We can model *bounded* encapsulation by having additional fields that are not used when the packet is decapsulated. The body of packets is not modeled. For reachability, we will only model a single (symbolic) packet starting at the source. For reachability we are interested in the "state" of a packet as its current location. The current location of a packet is modeled by a Boolean location predicate. For example, the predicate $R_1(dst, src)$ is true when a packet with destination *dst* and source *src* is at router R_1.

Operations in the network (i.e., forwarding and rewriting) are modeled as Datalog rules. Forwarding changes the location of a packet, and rewriting changes packet fields. A Datalog rule consists of two main parts separated by the :- symbol. The part to the left of this symbol is the `head`, while the part to the right is the `body` of the rule. A rule is read (and can be intuitively understood) as "`head` holds if it is known that `body` holds". The initial state/location of a packet is a *fact*, i.e., a rule without a `body`. For example, $A(dst, src)$ states that the packet starts at location A with destination address *dst* and source address *src*.

To model router rules as in Figure 1, it is convenient to use some shorthand for predicates that represent the matching condition, called a *guard*. Updates can be represented as relations over current state values dst, src and next state values dst', src' of the packet headers. Both guards and updates can thus be written as relations over dst, src and their next state values dst', src'. Most of the updates do not change anything, and are identity functions. The update from the last rule sets dst' to the concatenation of $dst[2]\ 0\ dst[0]$.

Thus, the relevant guards and updates from Fig. 1 are:

$$
\begin{aligned}
G_{12} &:= dst = 10\star \wedge src = 01\star \\
G_{13} &:= \neg G_{12} \ \wedge \ dst = 1\star\star \\
G_{2B} &:= dst = 10\star \\
G_{3D} &:= src = 1\star\star \\
G_{32} &:= \neg G_{3D} \ \wedge \ dst = 1\star\star \\
Id &:= src' = src \ \wedge \ dst' = dst \\
Set0 &:= src' = src \ \wedge \ dst' = dst[2]\ 0\ dst[0]
\end{aligned}
$$

Notice that G_{13} includes the negation of G_{12} to model the fact that the rule forwarding packets from R_1 to R_3 has lower priority than the one forwarding packets from R_1 to R_2. Armed with this shorthand, the network of Fig. 1 can now be modeled as:

$$
\begin{array}{rcl}
 & & B(dst, src) \\
R_1(dst, src) & :- & G_{12} \wedge Id \wedge R_2(dst', src') \\
R_1(dst, src) & :- & G_{13} \wedge Id \wedge R_3(dst', src') \\
R_2(dst, src) & :- & G_{2B} \wedge Id \wedge B(dst', src') \\
R_3(dst, src) & :- & G_{3D} \wedge Id \wedge D(dst', src') \\
R_3(dst, src) & :- & G_{32} \wedge Set0 \wedge R_2(dst', src') \\
A(dst, src) & :- & R_1(dst, src) \\
 & ? & A(dst, src)
\end{array}
$$

Since we want to know all the packets leaving A that could reach B, we pose the Datalog query $?A(dst, src)$ at the end of all the router rules. The symbol ? specifies that this is a query, not a rule or a fact.

Network router FIBs and ACLs can be modeled by Datalog rules in a similar way. Networks are deterministic, but load balancing and failover can be encoded using nondeterminism. For instance, a router that can forward a packet to either R_1 or R_2 (load balancing) will have a separate rule for each possible next hop. This is sufficient because we are only interested in reachability questions.

2.3 Network Queries

We define three basic network verification problems.

Reachability analysis: given two network nodes n and n', does there exist some packet P that can reach n' from n?

Cycle detection: does there exist some packet P that starts at node n and reaches the same node n as P despite possible rewriting in between?

Forwarding loop detection: does there exist some packet P that can reach some node n from n (while possibly being re-written as new packet P')?

Forwarding loop detection in an approximation to the more general problem of cycle detection. It checks whether a packet can ever visit a same node more than once, which is usually undesirable in practice for performance reasons.

In addition to these three core verification problems, the following variants are clearly interesting in a networking context [15, 18]: all packets that reach a destination; non-reachability / black holes; all packets flow through a middle-box; disjoint paths for different protocols; consistency of backup routers; maximum path length; isolation between VPNs; packets guaranteed to flow between A and B. Each of these properties can be expressed as a Datalog query. We show some examples in [21]. Unlike [16–18] if one wishes to modify the reachability question (for example, can A reach B through C or D), one can do so by adding a few lines to a Datalog query without changing the backend code. By comparison, the FlowExp reachability language in NetPlumber [15] is less powerful and well-established than Datalog.

2.4 Connections to EFSMs and SAT

Checking the properties above can be reduced to checking various reachability queries in the *graph* defined by the network's topology and its routing rules. It is therefore useful to conceptually view the data plane of a network as an *Extended Finite-State Machine (EFSM)*. An EFSM is a Finite-State Machine (FSM) extended with a finite set $\mathscr{V}$ of Boolean variables. While a conventional FSM can only specify transitions between states, an EFSM can also specify a *guard* and a *command* with each transition. A guard is a Boolean condition on $\mathscr{V}$, while a command is an assignment to one or several variables in $\mathscr{V}$. In an EFSM, a transition can be taken only when its guard it satisfied; when the transition is taken, the command of the transition is executed, possibly modifying the values of variables in $\mathscr{V}$.

Given a Datalog network model M as defined in Section 2.2, it is easy to translate it (in linear time and logarithmic space) into an equivalent EFSM M': network nodes of M correspond to states of M', each bit of a packet P is represented by a Boolean variable in $\mathscr{V}$, and (Datalog) routing rules are encoded by transitions with corresponding guards and commands on the corresponding Boolean variables. The set of all constraints labeling all the transitions outgoing from a state s_n represents the set of packet routing rules used in node n. A packet P arriving at node n is routed next to successor node n' iff there is a transition t from s_n to $s_{n'}$ such that the guard of

t evaluates to true with the variable assignment defined by P for all the variables in $\mathcal{V}$. When a packet P enters a node n and is routed to another successor node n', it can be partially *rewritten* into a new packet P'. Even though P' is different from P, we will still refer to it as the same packet in what follows.

A Datalog network model, as well as any EFSM, can also be encoded as a *propositional logic formula*, and reachability queries can then be solved with a SAT (or SMT) solver, as we will discuss later. In the presence of loops in a network, a standard technique is simply to "unroll" the transition relation at most TTL times, as done in [18] and [27], where TTL ("Time-To-Live") is a given upper-bound on the number of hops a packet can go through in a network. The details are crucial, however, and our encoding is different from [27] and [18] and is described in Section 6.1.

3 The Complexity of Network Verification

In this section, we provide a more precise characterization than in earlier work. We show that if a network verification problem exhibits any one of 3 characteristics (absence of complex loops, no rewriting, no cycles in topology) then the problem is NP-complete. We also show that detecting forwarding loops in arbitrary networks is always NP-complete (not PSPACE-complete). These results are obtained by reduction to related problems on EFSMs (e.g., see [10]).

3.1 Complexity of Reachability

In its full generality, network reachability analysis is a hard problem.

Theorem 1 *The worst-case complexity of network reachability analysis and cycle detection are both PSPACE-complete in the size of the description of the network.*

However, we now define specific classes of networks for which reachability analysis is easier. We start by strengthening a result proved in [18]: reachability analysis in networks without packet rewriting and where each rule may test at most 1-bit is only NP-complete. But perhaps testing n-bits at each rule could make the complexity worse. Thanks to known results on EFSMs, it is easy to show that this is not the case.

Theorem 2 *For networks where packets are never rewritten, network reachability analysis is NP-complete.*

The next result also follows immediately from earlier results on EFSMs.

Theorem 3 *For networks whose topology does not include any loop, network reachability analysis is NP-complete.*

The previous theorem is not very interesting since most real networks are bi-directional (i.e., A can send packets to B and vice-versa) and therefore do have loops in their topology.

However, as pointed out in [27], real IP networks often use TTLs to protect against loops in practice, and reachability then becomes NP-complete.

Theorem 4 *Reachability analysis in a network without any forwarding loop is NP-complete, including networks with loops in their topology and packet rewriting.*

In other words, network reachability analysis and cycle detection are PSPACE-complete problems in general, but the worst-case complexity is reduced to NP-complete if the network topology is loop-free, if there are no forwarding loops, or if packets cannot be rewritten.

While these seem to cover most "real" networks, it is worth noting that MAC networks do not have TTL protections, and combinations of VLANs can sometimes be configured into loops where packets can, in theory, cycle though each VLAN is a spanning tree [16].

3.2 Complexity of Loop Detection

By contrast to reachability, we now show that the *forwarding loop detection* problem is "only" NP-complete even in the general case. This is slightly surprising as one might think loop detection is as hard as (or even harder than) reachability.

Theorem 5 *Forwarding loop detection is NP-complete for any network, including networks with loops in their topology and packet rewriting.*

By comparison, [27] points out that for networks *without* packet rewriting, forwarding loop detection is equivalent to cycle detection, and a forwarding loop indicates the presence of an infinite cycle. In contrast, for networks *with* packet rewriting, a forwarding loop does not necessarily imply a returning packet will return forever (see [16] for a description of three kinds of loops), but this case is still undesirable since there is usually no reason for a packet to return to any past location. The existence of a "forwarding loop" in a network/EFSM is a necessary condition for the existence of a cycle in its state space, but not the other way round; this explains why the former may be easier to detect than the latter.

Observe that allowing packet duplication or broadcasting, i.e., nondeterministic networks, does not change the worst-case complexity of any of the problems above.

4 Experimental toolkit

We transition to *experimentally* studying the difficulty of reachability using standard verification tools. In addition to the publicly available Hassel C code [11], we used two classic model checking algorithms (BMC and PDR) and a Datalog framework, all of which are implemented in the Z3 [6] engine, also publicly available [21]. Other researchers can replicate and extend our results. We experiment with:

BMC: BMC denotes a classic bounded model checking algorithm [5]. Given our view of networks as extended finite state machines, applying model checking to network reachability is very natural.

PDR: PDR stands for Property Directed Reachability [7, 12] and is considered state-of-the-art in hardware model checking [7].

SAT/SMT Solver: SMT (Satisfiability Modulo Theories) extend SAT to a richer modeling language; in particular, the theory of bit-vectors nicely models headers. We use Z3's SAT/SMT solver which is considered state of the art, and extended it to compute reachability sets as described later.

Z3 Datalog Framework: Z3 provides a Datalog framework called μZ [13] with several backends. We added three new backends to [13] to efficiently compute reachability sets.

5 Datalog Algorithms

In this section, we present the main changes we implemented in μZ [13] to make it scale to the domain of network verification.

5.1 Data Structures

None of the existing backends in μZ performed well because sets of packets are tables; without compression these tables take too much storage. Hence, we implemented three new backends.

The first backend uses BDDs (Binary Decision Diagrams [2]) to represent Datalog tables. BDDs are a classic data structure to compactly represent a boolean function. A classic paper augments Datalog with BDDs [23] for reachability analysis in sequential programs, so our use of BDDs for Datalog is not surprising.

The other two backends are based on ternary bit-vectors, inspired by Header Space Analysis (HSA) [16], but placed in a much more general setting by merely adding a new data structure to Datalog. The simplest backend is a list of ternary strings we call a *union of cubes*. However, in both HSA [16] and Veriflow [17], a key optimization is to represent sets of packets as a *difference* of ternary strings. For example, $1**\setminus 10*$ succinctly represents all packets that start with 1 other than packets that start with 10. Thus the third data structure we added was what we call *difference of cubes*. More precisely, for ternary bit-vectors v_i and v_j, a difference of cubes represents a set

$$\bigcup_i \left(v_i \setminus \bigcup_j v_j \right)$$

The difference of cubes representation is particularly efficient at representing router rules where there are dependencies. For example the second rule in Figure 1 takes effect only if the first rule does not match. More precisely, in verification terms, difference of cubes is particularly efficient at representing formulas of the form $\varphi \wedge \neg\varphi_1 \wedge \cdots \wedge \neg\varphi_n$, with formulas φ and φ_i being of the form $\bigwedge_i \phi_i$ and ϕ_i having no boolean operators. This formula form is precisely what we obtain in the transfer functions of routing rules, with φ being the route matching formula, and the $\neg\varphi_i$ being the negation of the matching formula of the dependencies of the rule.

Code: All datalog backends added to Z3 were implemented in C++. The BDD backend takes 1,300 LoC (`src/muz_qe/dL_bdd_relation.*`), and the union of cubes and difference of cubes backends take almost 2,000 LoC (`src/muz_qe/dl_hassel_*`).

5.2 Combining Select and Project

One can pose reachability queries as in Figure 1 to μZ and find the set of packets that flow from A to B. Under the covers, μZ executes Datalog queries by converting them into relational algebra, described elsewhere [4].

Intuitively, a set of packet headers entering a router (say R_1 in Figure 1) should be thought of logically as a Datalog table (implemented say as a BDD). The set of output packets going to say R_2 are computed by finding a relation between input packets and corresponding output packets. The relation is computed in two steps: first, μZ *joins* the set of input packets I to the set of all possible output packets A to create a relation (I, A). Next, it *selects* the output packets (rows) that meet the matching and rewrite conditions to create a pruned relation (I, O). Finally, it *projects* away the input packets to be left with the set of output packets O. While this sounds like a very indirect and inefficient path to the goal, this is the natural procedure in Datalog. We will show that we can make it efficient for networking and other domains where there are equality constraints between variables.

While the join with all possible output packets A appears expensive, A is compactly represented as a single cube. The bigger expense is that the intermediate table produced by the select is often significantly larger than the table after the projection. This is particularly due to our representations which are inefficient at representing equality constraints between bits in input and output packets; for example, union of cubes is exponential in the number of equalities of don't care columns, and difference of cubes is doubly linear. To avoid this inefficiency, we had to combine selection and projection. An example will make this clear.

In Figure 1, consider an input packet $1\star\star\star\star\star$ at router R_3 that is forwarded to router R_2. Recall that the first 3 bits in this toy example are the destination address, the next 3 are the source address. We first join the table representing input packets with a full table (all possible output packets), obtaining a table with the row $1\star\star\star\star\star\star\star\star\star\star\star$, where the first six bits correspond to the input packet at R_3, and the remaining six bits belong to the output destined to R_2.

Then, we apply the guard and the rewrite formulas and the negation of all of the rule's dependencies (i.e., we perform a generalized select), and we obtain the following expression (in difference of cubes notation; the expression in union of cubes notation would be even larger):

$$
\begin{aligned}
&1\star\star\star\star\star 10\star 0\star\star \setminus (\\
&\ \star\star 0\star\star\star\star\star 1\star\star\star \cup \star\star 1\star\star\star\star\star 0\star\star\star \cup\\
&\ \star\star\star\star 0\star\star\star\star\star 1\star \cup \star\star\star\star 1\star\star\star\star\star 0\star \cup\\
&\ \star\star\star\star\star 0\star\star\star\star\star 1 \cup \star\star\star\star\star 1\star\star\star\star\star 0))
\end{aligned}
$$

While this looks complicated, the unions in the difference are simply ruling out cases where the "don't care" $\star$ bits are not copied correctly. The first term states that we can't have the third destination address bit be a 0 in the *input* packet and the third destination address bit in the *output* packet be a 1; the next term disallows the bits being 1 and 0 respectively. And so on for all the bit positions in *dst* and *src* where both are $\star$.

After the select operation, we perform a projection to remove the columns corresponding to the input packet (the first 6 bits) and therefore obtain a table with only the output packets. Again, in difference of cubes representation, we obtain simply $10\star 0\star\star$. The final result is significantly smaller than the intermediate result, and this effect is much more pronounced when we use 128 bit headers!

Therefore, we created a combined select-project operation that performs the equivalent of a generalized select followed by a projection. In verification terminology, this operation corresponds to computing the strongest post-condition of the transition relation.

We implemented the combined select-project operation in all the three of our Datalog backends. To make it efficient, we needed to find a way to compute the projection implicitly without explicitly materializing the intermediate results after the selection. For both the union of cubes and difference of cubes backends, we did this using a standard union-find data structure to represent equivalence classes (copying) between columns. When establishing the equality of two columns, if both columns (say bit 3 of the Destination address in both input and output packets) contain "don't care" values and one of them (bit 3 in the input packet) will be projected out, we aggregate the two columns in the same equivalence class. While this suffices for networking, we added two more rules to generalize this construction soundly to other domains. First, if only one of the columns contains a don't care value, then we set that column and all the others in the same equivalence class to the value of the column without a don't care value. Second, at most one column in an equivalence class may be a column that will not be projected out.

We now show how the combined select-project operation works in our example. The input we receive is the table after the join operator, i.e., $1\star\star\star\star\star\star\star\star\star\star\star$. Then we apply the rewriting (see last rule in Figure 1), which states that all bits should be copied, with the exception of the second bit, which should be set to 0. For the first two bits, it is easy to see that the resulting table is: $1\star\star\star\star\star 10\star\star\star\star$. The third bit, however, is a don't care and will be projected out.

Therefore, instead of explicitly representing the equality in the table, we keep it implicitly by stating that columns c_2 and c_8 are in the same equivalence class. We are assuming each ternary bit is a column, and that we count columns left-to-right, starting at 0, so the third destination address bit in the input is c_2 and the corresponding bit in the output is c_8 (because we use 6 bit headers).

If later, there is another equality mentioning any of these two columns, either we join the corresponding equivalence classes if the other column is also a don't care, or we need to set all columns in the equivalence class to the equated value (either a constant or a non-projected out don't care column). The table after the rewriting is: $1\star\star\star\star\star 10\star 0\star\star$, plus the equivalence classes $\{c_2, c_8\}$, $\{c_4, c_{10}\}$, and $\{c_5, c_{11}\}$.

The final step is to do the projection, where we discard the equivalence classes altogether, since they are not relevant anymore, and thus we obtain: $10\star 0\star\star$. In this example, the equivalence classes are not even useful but in more general cases they are needed for soundness. Finally, beyond these basic ideas, we implemented a number of other optimizations that we found crucial, some of which we describe briefly in Section 8.3.

Code: We also made several improvements to the Datalog solver itself, which were released with Z3 4.3.2. These improvements include, for example, reducing the worst case complexity of some algorithms, as well as a constant propagation-like optimization in the relational algebra compiler. These optimizations significantly reduced Z3's memory usage in our benchmarks by up to 40% .

6 Modifying SAT/SMT Solvers to Compute Reachability Sets

Unlike Datalog, SAT and SMT solvers only compute one solution, called a *model*, for a given formula. In this section, we present an algorithm to compute reachability sets (or, equivalently, all models of a formula) using a SAT/SMT solver as a black-box. We first describe how to encode network reachability as a SAT/SMT formula. We then show how to efficiently generalize the single solution to compute all packets from A that can reach B by exploiting the limited rewriting done at routers.

6.1 Our SMT Encoding

Given a TTL bound and a specific query, we encode a network as a logic formula using a query-driven procedure similar to symbolic simulation/execution of network paths up to length TTL, as also done in [18]. Roughly speaking, we evaluate the query as it passes through the routers and produce a formula that encodes all possible paths through the routers. At each step, the added formulae are proportional to the overall number of router rules. The size of the encoding has an upper-bound of the TTL times the number of router rules (but usually considerably smaller).

In contrast, the size of the encoding of [26] is always proportional to the number of routing rules. This encoding has the advantage that the same network encoding can be used to answer any query, but it does not support cycle detection (only forward loop detection).

Note that unlike the encodings in [17, 18, 26], our encoding can handle load balancing and replication which we will use in our cloud benchmarks.

We perform several optimizations on the generated formulas. For example, for each set of rules with the same output port, we aggregate the rules by their input port, which allows sharing dependencies across rules, reducing the size of the overall formula.

Reducing formula lengths does appear to be important in practice. On the Stanford network benchmark, [26] reports 100 secs for a SAT query (A can reach B) and 5 secs for an UNSAT query (A cannot reach B). By contrast, our encoding took 11.5 sec (8.7× faster) and 0.7 sec (7.1× faster) for SAT and UNSAT queries, respectively. We could not compare our results with [18] and [17] because their code is not public and they do not report results on the Stanford benchmark.

6.2 Our All SAT algorithm for Networking

Our SMT encoding finds a single packet that can reach a destination from a source in less than a second. But that is not good enough. The Hassel C code that implements HSA [16] finds *all* solutions in less than a second. We now show how we extend finding a single reachable packet using any SAT encoding to efficiently finding all reachable packets.

The classic method to generalize a single SAT solution to all solutions is to do a "block the model" loop. Each time a solution is found, we form a conjunction of the original formula and the negation of the solution, and iterate, as shown in Algorithm 1.

```
Algorithm 1: All-SAT
while $\varphi$ is SAT do
    $sol \leftarrow \mathsf{Generalize}(\mathsf{GetModel}(\varphi))$;
    $\varphi \leftarrow \varphi \wedge \neg sol$;
end
```

Unfortunately, the classic method may not converge quickly if there are millions of solutions as it can take millions of iterations, each of which requires a call to a SAT/SMT solver. To speed this process, we can try to *generalize* solutions in order to cover the set of all possible solutions as fast as possible. Consider the special case where the network does not rewrite packets. Intuitively, we can guess in that case that the value of many bits of many solutions will actually be irrelevant and can be viewed as wildcards. How can we spot these bit positions?

Using verification terminology, in the special case where there are no rewrites, we will generalize a model m (specific assignments of values to header bits) of a reachability formula φ by the following linear search algorithm:

```
Algorithm 2: Generalize
Input: $\varphi$ and model $m$
foreach $\ell \in m$ do
    if $\bigwedge(m \setminus \{\ell\}) \Rightarrow \varphi$ then
        $m \leftarrow m \setminus \{\ell\}$
    end
end
```

Intuitively, for each bit ℓ in the solution m we have found, we try and discard bit ℓ from the solution. The $m \leftarrow m \setminus \{\ell\}$ statement effectively makes ℓ a wildcard bit position by discarding it from the set of header bits considered in solution m. At each iteration, we execute one call to a SAT/SMT solver to prove whether a given bit can be wildcarded or not.

This is a linear search algorithm to discard bits that are "don't care" in the solution m. Algorithm 1 alone will take at least 2^w iterations if there are w wildcarded bits in the solution. By contrast, Algorithm 2 takes n iterations to discover the same 2^w solutions, where n is the length of the header in bits (in our experiments we use $n = 128$ to model a packet header).

There are standard techniques known in the verification community to do even better than linear search. For example, SAT solvers do not just return that a formula is unsatisfiable but can also output a subset of variables that caused the unsatisfiability, called an *unsat core*. An unsat core can provide hints to minimize the set of variables to search. Other standard ideas are to use some form of non-linear or binary search as expressed in techniques such as QuickXplain [14] or Progression [19]. However, in our experiments we found these techniques made little difference suggesting that speed was dominated not by the number of iterations but by the cost of solution representation.

Unfortunately, Algorithm 2 does not work if we wish to compute the set of *destination packets*. (Algorithm 2 does work to compute the set of source packets.) This is because we are now trying to compute all satisfying assignments of the projection of a relation between input packets and output packets, where the output packets are a function of the input packets. Some bits may be rewritten and some stay equal (as in Example 1). Thus even if bit 15 is a wildcard in the output packet, if there is a "copying" constraint stating that bit 15 in the output packet is equal to bit 15 in the input packet, bit 15 in the output will not be a wildcard on its own in the overall formula (since, e.g., having the input bit true and the output bit false is not a model of the formula).

What we need to do, then, in the face of copying, is not only to discover the wildcard bits in the generalized solution but also to *discover the copying constraints between corresponding input packet bits and output packet bits*. We can exploit the fact that rewriting in networking is not arbitrary (from a set of 128 source bits to a set of 128 arbitrary destination bits in the relation); instead large sets of bits stay the same because routers set only a few fields like MAC addresses.

In our networking domain, we have 3 cases for the output variable (the packet that arrives at the destination). Each bit is either equal to the input bit at the source, it is a constant, or it is a don't care bit (arising from the non-determinism present in the network). We take advantage of this insight to extend Algorithm 2 to compute the set of all reachable destination packets as follows.

Assuming we have $m = m_{in} \cup m_{out}$ and for every $\ell_{out} \in m_{out}$ there is a matching $\ell_{in} \in m_{in}$, the algorithm becomes:

Algorithm 3: Generalize$_=$

```
Input: $\varphi$ and model $m = m_{in} \cup m_{out}$
foreach $\ell_{out} \in m_{out}$ do
    if $\bigwedge(m \setminus \{\ell_{out}\}) \Rightarrow \varphi$ then
        $m \leftarrow m \setminus \{\ell_{out}\}$
    else if $\bigwedge(m \setminus \{\ell_{in}, \ell_{out}\}) \wedge (\ell_{in} \leftrightarrow \ell_{out}) \Rightarrow \varphi$ then
        $m \leftarrow (m \setminus \{\ell_{in}, \ell_{out}\}) \cup \{\ell_{in} \leftrightarrow \ell_{out}\}$
    end
end
```

What has changed from Algorithm 2 is that now we do a search among pairs of bits and ask the SAT/SMT solver whether the particular generalization is a solution assuming this pair of bits is copied ($\ell_{in} \leftrightarrow \ell_{out}$). Doing it for all pairs of bits would be quadratic and doing it for all subsets of bits would be exponential. However, doing it for corresponding bits in the input and output takes linear time.

In summary, Algorithm 3 does a pairwise linear search over the set of input/output variables of a relation, and tests if they are "don't care" under equality. While this works well for networking, this algorithm can be extended to other domains where the "shape" of the solution space is known. In other words, if one has some idea of the potential relations between subsets of variables (e.g., the copying relationship between input and output bits), the loop can be extended to check for these relationships to allow efficient generalization.

While Algorithm 3 is indeed specialized to the networking domain and exploits limited rewriting, it is still general because it allows generalizing from any specific SAT/SMT solution in any SAT/SMT encoding to find all reachability sets. For example, it can be used to generalize the SAT solutions in [18] and [26]. It can even be used to generalize the solutions produced by model checkers such as BMC and PDR, and could even be applied to the BDD based verification methods of [1].

7 Benchmarks

We use two sets of benchmarks: one real enterprise network and one parametrizable model of a cloud.

Stanford: This is a publicly available [22] snapshot of the routing tables of the backbone network of Stanford University, and a set of network reachability and loop detection queries. The core has 16 routers, organized in a star topology, where the central node is replicated across two routers. The total number of rules across all routers is 12,978. About 62% of these rules have no dependencies with other rules. The maximum number of dependencies of a rule is 910 rules, with an average of 11.6 dependencies per rule. 9% of the router rules are multicast rules which specify up to a maximum of 15 different ports, with the common case being 2 to 5 ports. Almost 11% of the rules have no destination port ("drop" rules). About 26% of the rules rewrite at least one bit of packet header, often rewriting 32 bits of source IP address, possibly for NAT. Note that we use 128 bit headers. This network includes extensive VLAN support but has no non-determinism unlike our next benchmark.

Cloud provider: Our second benchmark is a parameterizable model of of a cloud provider network with multiple data centers, as used by say Azure or Bing. We use a fat tree as the backbone topology *within* a data center and single top-of-rack routers as the leaves. Each router other than the leaves in the fat tree is connected to k other routers above and k routers below, where k is the replication factor. The number of ports for backbone routers and top-of-rack generators is configurable. Data centers are interconnected by an all-to-all one-hop mesh network. In addition to replication factor and router ports, other parameters include the number of data centers, the number of machines per data center, the number of IP addresses per machine (each modeling a VM or virtual machine), and the number of services s provided by each machine and by each data center.

Each data center is randomly assigned s services (destination TCP Port numbers) that it provides from a set of M possible services. As a consequence, more than one data center may provide the same service. Each service receives a randomly assigned public IP address, as well as an randomly assigned internal port number (so that we can run multiple services on the same machine). Each service is also randomly assigned a list of other services to which it can communicate, plus the internet (not all services are accessible from the internet). In each data center, a service is provided by a set of machines with contiguous IPs from a prefix assigned to each data center.

Each service has a public IP address. When a packet reaches a data center, the packet is rewritten to have an internal IP as destination and the internal TCP port of the service as is done in many clouds such as Azure. We use non-deterministic routing to model load-balancing between data centers (if two data centers provide the same service), between machines that provide the same service, and also across fat-tree paths to the same machine. When a packet leaves a data center, it is rewritten to have a public IP as the source IP.

Multiple ACL rules are deployed. For example, the entry-point from the internet to a data center performs multiple checks for security purposes. These include dropping packets that have local source IPs (e.g., `127.0.0.1/8`), packets that use the cloud provider's public IP range as the source IP, or are malformed (i.e., not using TCP/IP protocols). These are checks we have observed within Microsoft data centers. We also deploy ACL rules in top-of-rack routers to isolate problems in case a machine is compromised. The packets received from the host machines are checked to ensure that the source IP corresponds to the assigned range, and that the service sending the packet is allowed to contact the destination.

Code: The cloud benchmark generator was implemented in Python and consists in about 550 LoC, and it uses the Hassel-Py library to generate the topology files. The Python script to generate Datalog files from network topologies takes 350 LoC. The program that generates SMT formulas from network topologies consists in about 600 LoC of Python. Details can be found in [21].

8 Evaluation

We ran the benchmarks with multiple tools, including two model checkers (a bounded model checker, and the state-of-the-art PDR [12]), an SMT solver, and a Datalog solver coupled with different backends (BDDs, union of cubes, and difference of cubes). All the used tools are from Z3 4.3.2 [6]. The input to the BMC and PDR model checkers is the same as for the Datalog solver. The benchmarks were run on a machine with an Intel Xeon E5620 (2.4 GHz) CPU, and running Linux 2.6.32. The tests were given a timeout of 5 minutes and a memory limit of 8 GBs.

8.1 Run Time Performance

Table 1 is a small sampling of extensive test results that can be found in [21]. It shows the time (in seconds) to run multiple tools on a representative subset of the Cloud and Stanford benchmarks, which includes reachable and unreachable queries, as well as one loop detection query.

All cloud benchmarks in the table have 5 data centers, assign 4 IPs per host, and use a router replication factor of 2. Most benchmarks have 25 services, with each data center providing at most 10 services, and each host proving up to 4 services, with the exception of "Cloud More Services" benchmark, which has 300, 64, and 48, respectively. Most benchmarks use core and leaf routers with 32 ports, while the "Medium Cloud Long" benchmark uses 8 and 4 ports, respectively. Both "Small Cloud" and "Cloud More Services" benchmarks have 200 hosts per data center, "Medium Cloud" and "Medium Cloud Long" benchmarks have 1,000, and "Large Cloud" benchmarks have 2,000.

Note that the model checker tools only compute satisfiability answers (i.e., "is a node reachable or not?"), while Datalog computes all reachable packets at the destination. For the SMT technique, we provide results for both type of queries. All the SMT experiments were run

Test	Model Checkers		SMT		Datalog			Hassel C
	BMC	PDR	Reach.	All sols.	BDDs	Cube union	Diff. of cubes	
Small Cloud	0.3	0.7	0.1	–	0.2	0.6	0.2	–
Medium Cloud	T/O	12.9	0.2	–	2.2	3.2	2.2	–
Medium Cloud Long	M/O	M/O	4.8	–	13.3	57.4	13.0	–
Cloud More Services	7.3	70.4	12.5	–	6.2	7.6	5.8	–
Large Cloud	T/O	M/O	2.8	–	62.2	59.2	58.2	–
Large Cloud Unreach.	T/O	M/O	1.1	n/a	59.0	59.4	58.6	–
Stanford	56.7	23.5	11.5	1,121	13.2	T/O	6.3	0.9
Stanford Unreach.	T/O	23.2	0.1	n/a	T/O	T/O	2.4	0.1
Stanford Loop	23.0	22.1	11.2	290.2	11.4	T/O	4.1	0.2

Table 1: Time (in seconds) taken by multiple tools to solve network reachability benchmarks. Model checkers only compute satisfiability answers, while Datalog produces reachability sets. T/O and M/O mean, respectively, that the tool exceeded the maximum allowed time or memory.

with the minimum TTL (i.e., an unrolling) for each test; for example, the TTL for Stanford was 3 for reachability and 4 for loop detection. Higher TTLs significantly increase the running time. We do not provide the running time for reachability sets with SMT for the cloud benchmarks, since our prototype AllSAT algorithm does not support non-determinism. We were unable to run the Hassel C tool [11] on the cloud benchmarks, since HasselC has a few hardwired assumptions, such as router port numbers following a specific naming policy.

The first takeaway is that Z3's implementation of Datalog is faster at computing reachability sets (all solutions) than its model checkers or SAT solvers are at computing a single solution. This is not surprising as the model checkers make multiple copies of each router transfer function and stitch them together which is quite wasteful. The performance of model checkers also seems to degrade exponentially with path length (see row 3 versus row 2 where the model checkers timeout). Similarly, unrolling seems to exact a price for SMT solvers. Even our efficient AllSAT generalization algorithm is around 200× slower than Datalog (row 7). Datalog with difference of cubes is the most competitive implementation we have tested. Datalog with a BDD backend shows good performance as well except in row (Stanford Unreach.) where it timed out. The Hassel C tool takes under a second for Stanford, faster than Datalog, but at the expense of reduced flexibility.

8.2 Compactness of Representations

A natural hypothesis is that speed depends on the compactness of intermediate representations of sets of packet headers as each implementation progresses. Table 2 shows the sizes (in bytes and number of disjunctions) required by BDDs and Difference of Cubes to represent the largest intermediate set of packets. For the Stanford benchmarks, the sizes are comparable. However, the Cloud More Services benchmark (Row 4) shows that the Difference of Cubes representation can be 20 times larger than BDDs and yet (see Table 1) has faster run time. One explanation is that the operations on Difference of Cubes are faster than the corresponding BDD operations, possibly because the "glue" we added to paste BDDs into Z3 had some inefficiency.

Test	BDDs	Diff. of Cubes	
	Bytes	# Disjs.	Bytes
Small Cloud	7,664	4	2,008
Medium Cloud	2,004	8	6,872
Medium Cloud Long	2,004	2	680
Cloud More Services	2,644	240	26,904
Large Cloud	2,004	8	6,872
Large Cloud Unreach.	1,104	8	6,872
Stanford	150,744	332	158,368
Stanford Unreach.	(T/O)	253	36,224
Stanford Loop	150,744	332	158,368

Table 2: Size (in bytes and number of disjunctions) taken by the BDD and difference of cubes Datalog backends to represent the largest intermediate results.

8.3 Effect of Optimizations

Table 3 shows the impact of disabling optimizations that we implemented in the difference of cubes Datalog backend. We measured the speedup provided by each individual optimization using a timeout of 10 minutes.

The optimizations we tested were as follows. In Section 5.2, we described a combined select-project rela-

Test	Select-Project	Guard split	Dep. Simp.	de-CNF	Back. subsumption	Fwd. subsumption
Large Cloud	0.9	0.9	1.1	1.0	0.9	0.9
Large Cloud Unreach.	1.0	1.0	1.2	1.0	1.0	1.0
Stanford	> 96	14.0	1.2	> 96	0.9	0.8
Stanford Unreach.	> 113	11.0	1.2	> 113	0.9	0.9
Stanford Loop	> 143	3.0	1.3	> 143	0.9	1.2
Average	> 71	6.0	1.2	> 71	0.9	1.0

Table 3: Impact (given as speedup) of optimizations implemented in the Datalog with difference of cubes engine in a few representative benchmarks. Values higher than 1.0 mean that the technique reduced the running time.

tional operation. We tested Guard split, which takes a rule formula and splits into a guard (which can be efficiently represented in difference of cubes), and the rest (rewrites) The guard is then cached in the difference of cubes format and is directly ANDed with the relations whenever needed. Then, we have dependency simplification, which amounts to simplifying the input in SMT format with Z3's simplification tools. For example, if there are three router rules that match $111\star$, $11\star\star$, and $1\star\star\star$, the guard for the third rule can be simplified from the negation of the first two rules to the negation of just the second rule.

Next, the de-CNF optimization reverts the transformation that Z3 applies to formulas to put them in CNF in order to arrange the formulas in a more convenient form for the difference of cubes representation. Finally, we have backward and forward subsumption, which discard previously derived or newly derived facts, respectively, which can potentially reduce the memory usage. For example, if the reachability set is currently $10\star$ and we decide to union it with $1\star\star$, the $1\star\star$ subsumes the $10\star$, and therefore the result of the operation would be just $1\star\star$.

We observe that the crucial optimizations were not the generic optimizations that apply in any setting such as subsumption or dependency simplification. The essential optimizations were select-project and de-CNF. Select-project gets around artifacts of the Datalog implementation, and de-CNF works around the fact that Difference of Cubes does not work well with the canonical SMT representation in terms of CNF (Conjunctive Normal Form). While dependency simplification mostly improves performance, subsumption provides no gain or even hurts performance!

[21] also describes the best configuration settings for each tool. For example, the Datalog implementations work best with a so-called "unbound compression" option disabled. Unbounded compression is needed for Datalog table implementations that are typically not compressed; this is unnecessary for backends like Difference of Cubes that are inherently compressed.

9 Conclusion

Tools such as Veriflow [17] and NetPlumber [15] use adhoc custom code, and are therefore hard to extend. If network verification is to mature into a networking CAD industry, its tools must evolve into more standard, principled and extensible techniques, building upon common foundations developed over many years, constantly being improved, and used in other application domains.

Our paper shows that the efficiency of state-of-the-art network verification tools can be matched by re-using existing verification machinery optimized for the network verification domain. Adding incremental analysis as in Veriflow and NetPlumber [15,17] can result in even faster speeds. Specifically, we showed that this result can be achieved using a general-purpose Datalog engine specialized using a difference-of-cubes data representation and combined with a select-and-project operator that efficiently capture frequent bit-copying constraints from input to output packets.

Our work allows the full power of Datalog to be used in the network domain. For example, we are building a synthesis tool for Azure firewalls from a high-level specification language (instead of operators typing low-level ACLs in error-prone fashion). Given the results of this paper, we plan to use Datalog to compute reachability sets and add additional constraints for minimality (e.g., router X has no more than 10 rules). By building on solid foundations, standard components, publicly available code and benchmarks, network verification can benefit from future improvements that will undoubtedly take place.

Conversely, the study of network verification problem instances provides insights into exploitable aspects of the domain (e.g., formulas with one level of negation). Our new backends are available for use in other verification domains. In fact, many of our algorithms including the select-and-project implementation using equivalence classes were designed for general-purpose usage. Some of this machinery is not strictly needed for networking, but helps advance the craft and science of verification.

References

[1] AL-SHAER, E., AND AL-HAJ, S. FlowChecker: configuration analysis and verification of federated openflow infrastructures. In *SafeConfig* (2010).

[2] BRYANT, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. 35*, 8 (Aug. 1986), 677–691.

[3] BURCH, J. R., CLARKE, E. M., MCMILLAN, K. L., DILL, D. L., AND HWANG, L. J. Symbolic model checking: 10^{20} states and beyond. In *LICS* (1990).

[4] CERI, S., GOTTLOB, G., AND TANCA, L. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng. 1*, 1 (Mar. 1989), 146–166.

[5] CLARKE, E. M., BIERE, A., RAIMI, R., AND ZHU, Y. Bounded model checking using satisfiability solving. *Formal Methods in System Design 19*, 1 (2001), 7–34.

[6] DE MOURA, L., AND BJØRNER, N. Z3: an efficient SMT solver. In *TACAS* (2008).

[7] EÉN, N., MISHCHENKO, A., AND BRAYTON, R. Efficient implementation of property directed reachability. In *FMCAD* (2011).

[8] FEAMSTER, N., AND BALAKRISHNAN, H. Detecting BGP configuration faults with static analysis. In *NSDI* (2005).

[9] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: a network programming language. In *ICFP* (2011).

[10] GODEFROID, P., AND YANNAKAKIS, M. Analysis of boolean programs. In *TACAS* (2013).

[11] HASSEL C. `https://bitbucket.org/peymank/hassel-public`.

[12] HODER, K., AND BJØRNER, N. Generalized property directed reachability. In *SAT* (2012).

[13] HODER, K., BJØRNER, N., AND DE MOURA, L. μZ: an efficient engine for fixed points with constraints. In *CAV* (2011).

[14] JUNKER, U. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *AAAI* (2004).

[15] KAZEMIAN, P., CHANG, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[16] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: static checking for networks. In *NSDI* (2012).

[17] KHURSHID, A., ZOU, X., ZHOU, W., CAESAR, M., AND GODFREY, P. B. Veriflow: verifying network-wide invariants in real time. In *NSDI* (2013).

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

[19] MARQUES-SILVA, J., JANOTA, M., AND BELOV, A. Minimal sets over monotone predicates in boolean formulae. In *CAV* (2013).

[20] MCKEOWN, N. Mind the gap. In *SIGCOMM* (2012). `http://youtu.be/Ho239zpKMwQ`.

[21] NETWORK VERIFICATION WEBSITE. `http://web.ist.utl.pt/nuno.lopes/netverif/`.

[22] STANFORD BENCHMARK. `http://goo.gl/FtzxRr`.

[23] WHALEY, J., AND LAM, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI* (2004).

[24] XIE, G. G., ZHAN, J., MALTZ, D. A., ZHANG, H., GREENBERG, A. G., HJÁLMTÝSSON, G., AND REXFORD, J. On static reachability analysis of IP networks. In *INFOCOM* (2005).

[25] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic test packet generation. In *CoNEXT* (2012).

[26] ZHANG, S., AND MALIK, S. SAT based verification of network data planes. In *ATVA* (2013).

[27] ZHANG, S., MALIK, S., AND MCGEER, R. Verification of computer switching networks: An overview. In *ATVA* (2012).

A Encoding Reachability in Datalog

In this section, we show how to encode reachability queries using Datalog. Note that Datalog offers a simpler and powerful higher level way of specifying verification questions. Existing networking verification tools do not have any such a higher level language or if they do as in FlowExp [15] they are ad hoc (no clear semantics) and less powerful. We also how the Datalog queries can be simply extended to allow for faults which would require new algorithmic machiney and recoding in say [15].

A.1 All reachable packets in destination

We simply reverse the direction of rules.

$$\begin{array}{rcl} & & A(dst,src) \\ R_1(dst,src) & :- & A(dst,src) \\ R_2(dst',src') & :- & G_{12} \wedge Id \wedge R_1(dst,src) \\ R_3(dst',src') & :- & G_{13} \wedge Id \wedge R_1(dst,src) \\ B(dst',src') & :- & G_{2B} \wedge Id \wedge R_2(dst,src) \\ D(dst',src') & :- & G_{3D} \wedge Id \wedge R_3(dst,src) \\ R_2(dst',src') & :- & G_{32} \wedge Set0 \wedge R_3(dst,src) \\ & ? & B(dst,src) \end{array}$$

A.2 All packets that reach a destination

We add all destinations as additional facts. For example, we add the fact $D(dst,src)$ besides the fact $B(dst,src)$.

A.3 Non-reachability / black holes

The complement to the All-SAT answer are packets that are dropped or loop without reaching the specified destinations.

A.4 Forwarding loop detection

To determine if there are any packets leaving A and revisit the same node twice add an extra argument to every predicate. The argument remembers (non-deterministically) a node on the route. If the node reappears, then there is a loop:

$$\begin{array}{rcl} & & A(dst,src,0) \\ \widehat{R}_2(dst',src',\ell) & :- & G_{12} \wedge Id \wedge R_1(dst,src,\ell) \\ R_2(dst',src',\ell) & :- & \widehat{R}_2(dst',src',\ell) \\ R_2(dst',src',r_2) & :- & \widehat{R}_2(dst',src',\ell) \\ Loop(dst',src') & :- & \widehat{R}_2(dst',src',r_2) \\ & \vdots & \\ & ? & Loop(dst,src) \end{array}$$

A.5 Cycle detection

To determine if there are packets leaving A and getting trapped in an infinite loop we take a snapshot of both the node *and the contents of the packet*.

$$\begin{array}{rcl} & & A(d,s,d,s,0) \\ \widehat{R}_2(d',s',d_0,s_0,\ell) & :- & G_{12} \wedge Id \wedge R_1(d,s,d_0,s_0,\ell) \\ R_2(d,s,d_0,s_0,\ell) & :- & \widehat{R}_2(d,s,d_0,s_0,\ell) \\ R_2(d,s,d,s,r_2) & :- & \widehat{R}_2(d,s,d_0,s_0,\ell) \\ Loop(d,s) & :- & \widehat{R}_2(d,s,d,s,r_2) \\ & \vdots & \\ & ? & Loop(d,s) \end{array}$$

A.6 All packets flow through a middle-box

We add a 0-1 indicator variable v (for *visit*) to track if the middle-box was traversed. For example, if the middle box is R_3 we have

$$\begin{array}{rcl} & & A(dst,src,0) \\ R_1(dst,src,v) & :- & A(dst,src,v) \\ R_2(dst',src',v) & :- & G_{12} \wedge Id \wedge R_1(dst,src,v) \\ R_3(dst',src',v) & :- & G_{13} \wedge Id \wedge R_1(dst,src,v) \\ B(dst',src',v) & :- & G_{2B} \wedge Id \wedge R_2(dst,src,v) \\ D(dst',src',1) & :- & G_{3D} \wedge Id \wedge R_3(dst,src,v) \\ R_2(dst',src',1) & :- & G_{32} \wedge Set0 \wedge R_3(dst,src,v) \\ & ? & B(dst,src,0) \end{array}$$

A.7 Disjoint paths for different protocols

Similar to loop detection, instrument the predicates with one more variable that non-deterministically remembers a node on a path from A to B. We then pose a query of the form:

$$\begin{array}{rl} ? & A(dst_0,src_0,\ell),A(dst_1,src_1,\ell), \\ & Proto_0(dst_0,src_0),Proto_1(dst_1,src_1) \end{array}$$

It returns the set of packets from the two protocols that flow through the same node ℓ on the way to B.

A.8 Fault resilience

The encodings so far use non-determinism to model load balancing and failover (broadcast). Suppose we would like to check reachability in the presence of both faults as well as broadcasts.

We assume that faults are manifested by disabled routers and we are interested in checking fault tolerance up to a fixed number k of disabled routers. So if there are n routers we can use a logical formula over an n-bit vector to set at most k bits, indicating disabled routers. The bit-vector is passed as an additional argument and rules are only enabled if the fault bit, corresponding to the router being simulated, is not set. Thus, for an encoding into Datalog we have a vector d of disabled routers. For example, if $d[1]$ is the position used by router R_1, then our original encoding uses rules of the form:

$$R_1(dst, src, d) \quad :- \quad G_{12} \wedge Id \wedge R_2(dst', src', d) \wedge \neg d[1]$$

We can constrain d to have at most k bits set using a logical formula $AtMostK(d)$. The set of packets that cannot reach B from A with d faults are specified in Datalog with negation:

$$? \quad AtMostK(d) \wedge \neg A(dst, src, d)$$

We can of course strengthen the query to check if there are packets that reach the destination without faults, but are blocked in the presence of faults:

$$? \quad A(dst, src, d') \wedge AtMostK(d) \wedge \neg A(dst, src, d)$$

And we can enumerate packets that reach B under all legal faults:

$$\begin{array}{rcl} nA(dst, src) & :- & AtMostK(d) \wedge \neg A(dst, src, d) \\ & ? & \neg nA(dst, src) \end{array}$$

A.9 Maximum path length

We can query for packets that traverse some path that exceeds a threshold by adding a parameter that counts the number of nodes.

$$\begin{array}{rcl} & & A(dst, src, 0) \\ R_1(dst, src, k+1) & :- & A(dst, src, k) \\ R_2(dst', src', k+1) & :- & G_{12} \wedge Id \wedge R_1(dst, src, k) \\ & \vdots & \\ & ? & B(dst, src, k) \wedge k > threshold \end{array}$$