Checking Beliefs in Dynamic Networks

Nuno P. Lopes, Nikolaj Bjørner, and Patrice Godefroid, *Microsoft Research;*
Karthick Jayaraman, *Microsoft Azure;* George Varghese, *Microsoft Research*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes

This paper is included in the Proceedings of the
12th USENIX Symposium on Networked Systems
Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Checking Beliefs in Dynamic Networks

Nuno P. Lopes *Microsoft Research*
Nikolaj Bjørner *Microsoft Research*
Patrice Godefroid *Microsoft Research*
Karthick Jayaraman *Microsoft Azure*
George Varghese *Microsoft Research*

Abstract Network Verification is a form of model checking in which a model of the network is checked for properties stated using a specification language. Existing network verification tools lack a *general specification language* and *hardcode* the network model. Hence they cannot, for example, model policies at a high level of abstraction. Neither can they model dynamic networks; even a simple packet format change requires changes to internals. Standard verification tools (e.g., model checkers) have expressive specification and modeling languages but do not scale to large header spaces. We introduce Network Optimized Datalog (NoD) as a tool for network verification in which both the specification language and modeling languages are Datalog. NoD can also scale to large to large header spaces because of a new filter-project operator and a symbolic header representation.

As a consequence, NoD allows checking for beliefs about network reachability policies in dynamic networks. A belief is a high-level invariant (e.g., "Internal controllers cannot be accessed from the Internet") that a network operator *thinks* is true. Beliefs may not hold, but checking them can uncover bugs or policy exceptions with little manual effort. Refuted beliefs can be used as a basis for revised beliefs. Further, in real networks, machines are added and links fail; on a longer term, packet formats and even forwarding behaviors can change, enabled by OpenFlow and P4. NoD allows the analyst to model such dynamic networks by adding new Datalog rules.

For a large Singapore data center with 820K rules, NoD checks if any guest VM can access any controller (the equivalent of 5K specific reachability invariants) in 12 minutes. NoD checks for loops in an experimental SWAN backbone network with new headers in a fraction of a second. NoD generalizes a specialized system, SecGuru, we currently use in production to catch hundreds of configuration bugs a year. NoD has been released as part of the publicly available Z3 SMT solver.

1 Introduction

> Manually discovering any significant number of rules a system must satisfy is a dispiriting adventure — *Engler et al. [13].*

Our goal is to catch as many latent bugs as possible by static inspection of router forwarding tables and ACLs without waiting for the bugs to trigger expensive live site incidents. In our operational network, we see roughly one customer visible operational outage of more than one hour every quarter across our major properties; these live site incidents are expensive to troubleshoot and reduce revenue and customer satisfaction. As businesses deploy services, bug finding using static verification will become increasingly essential in a competitive world.

We have already deployed an early version of our checker in our public cloud; it has regularly found bugs (§ 7). Operators find our checker to be indispensable especially when rapidly building out new clusters. However, we wish to go deeper, and design a more useful verification engine that tackles two well-known [2, 15] obstacles to network verification at scale.

O1. Lack of knowledge: As Engler and others have pointed out [12, 14], a major impediment is determining what specification to check. Reachability policies can be thought of intuitively as "who reaches who". These policies evolve organically and are in the minds of network operators [2], some of whom leave. How can one use existing network verification techniques [1, 22–24, 26, 35] when one does not know the pairs of stations and headers that are allowed to communicate?

O2. Network Churn: Existing network verification techniques assume the network is static and operate on a static snapshot of the forwarding state. But in our experience many bugs occur in buildout when the network is first being rolled out. Another particularly insidious set of bugs only gets triggered when failures occur; for example, a failure could trigger using a backup router that is not configured with the right drop rules. Beyond such short term dynamism, the constant need for cost reduction and the availability of new mechanisms like SDNs keeps resulting in new packet formats and forwarding behaviors. For example, in recent years, we have added VXLAN [25] and transitioned to software load balancers in our VM Switches [30]. The high-level point is that verification tools must be capable of modeling such dynamism and provide insight into the effect of such changes. However, all existing tools we know of including Anteater [26], VeriFlow [24], Hassel [23] and NetPlumber [22] assume fixed forwarding rules and fixed packet headers, with little or no ability to model faults or even header changes.

Our approach to tackling these obstacles is twofold, and is embodied in a new engine called Network Optimized Datalog (NoD).

A1. General specification language to specify beliefs: We use Datalog to specify properties. Datalog is far more general than the regular expression language used in NetPlumber [22], the only existing work to provide a specification language. For example, Datalog allows specifying differential reachability properties across load balanced paths. More importantly, Datalog allows specifying and checking higher-level abstract policy specifications that one can think of as operator beliefs [13]. Beliefs by and large hold, but may fail because of bugs or exceptions we wish to discover. For example, a common belief in our network is "Management stations should not be reachable from customer VMs or external Internet addresses". This is an instance of what we call a *Protection Set* template: "Stations in Set A cannot reach Stations in Set B".

For this paper, a belief is a Boolean combination of reachability predicates expressed using Datalog definitions. Rather than the operator enumerate the crossproduct of all specific reachability predicates between specific customer VM prefixes and all possible management stations, it takes less manual effort to state such beliefs. If the engine is armed with a mapping from the predicate "customer VM", "Internet", etc. to a list of prefixes, then the tool can unroll this more abstract policy specification into specific invariants between pairs of address prefixes. Of course, checks can lead to false positives (if there are exceptions which require refining our beliefs) or false negatives (if our list is incomplete), but we can start without waiting for perfect knowledge.

We have found five abstract policy templates (Table 1) cover every policy our operators have enforced and our security auditors check for: protection sets, reachability sets, reachability consistency, middlebox processing, and locality. An example of a reachability consistency belief and bug is shown in Figure 2. We will describe these in detail in the next section but Table 1 summarizes examples. Most are present in earlier work except for locality and reachability consistency. Despite this, earlier work in network verification with the exception of NetPlumber [22] does not allow reachability invariants to be specified at this level of abstraction. We make no attempt to learn these abstract policies as in [2, 13]. Instead, we glean these templates from talking to operators, encode them using NoD, and determine whether violations are exceptions or bugs by further discussion with operators.

A2. General modeling language to model networks: We provide Datalog as a tool not just to write the specification but also to write the router forwarding model. Thus it is easy for users to add support for MPLS or any new packet header such as VXLAN without changing internals. Second, we can model new forwarding behaviors enabled by programmable router languages such as P4 [4]. Third, we can model failure at several levels. The easiest level is not to model the routing protocol. For example, our links and devices are divided into availability zones that share components such as wires and power supplies that can fail together. Will an availability zone failure disconnect the network? This can be modeled by adding a predicate to each component that models its availability zone and state, and changing forwarding rules to drop if a component is unavailable.

Policy Template	Example	Datalog Feature Needed
Protection Sets	Customer VMs cannot access controllers	Definitions of sets
Reachable Sets	Customer VMs can access VMs	Definitions, Negation
Reachability Consistency	ECMP/Backup routes should have identical reachability/same path length	Negation, Non-determinism, Bit vectors
Middlebox processing	Forward path connections through a middlebox should reverse	Negation
Locality	Packets between two stations in the same cluster should stay within the cluster	Boolean combinations of reachability predicates

Table 1: 5 common policy templates.

At a second level, we can easily model failure response where the backup routes are predetermined as in MPLS fast reroute: when one tunnel in a set of equal cost tunnels fails, the traffic should be redistributed among the live tunnels in proportion to their weights. The next level of depth is to model the effect of route protocols like OSPF and BGP as has been done by Fogel et al. [15] for a university network using an early version of NoD. All of these failure scenarios can be modeled as Datalog rules, with routing protocol modeling [15] requiring that rules be run to a fixed point indicating routing protocol convergence. Our tool also can be used to ask analogous "what if" questions for reliability or security of network paths. By contrast, existing tools for network verification like VeriFlow and NetPlumber cannot model dynamism without changing internals. This is because the network model is hardcoded in these tools.

In summary, we need a verification engine that can specify beliefs and has the ability to model dynamism such as new packet headers or failures. Existing verification network tools scale to large networks at high speeds but with the exception of NetPlumber [22] do not have a specification language to specify beliefs. NetPlumber's regular expression language for reachability predicates, however, is less rich than Datalog; for example, it cannot model reachability consistency across ECMP routes as in Figure 2. More importantly, none of the existing tools including NetPlumber can model changes in the network model (such as failures) without modifying the internals.

On the other hand, the verification community has produced an arsenal of tools such as model checkers, Datalog, and SAT Solvers that are extremely extensible accompanied by general specification languages such as temporal logic. The catch is that they tend to work well only with small state spaces. For example, several Datalog implementations use relational backends that use tables as data structures. If the solutions are sets of headers and the headers can be 100 bits long, the standard implementations scale horribly. Even tools such as Margrave [27] while offering the ability to ask "what if" questions for firewalls do not scale to enumerating large header spaces. Thus, our contributions are:

1. Modeling Beliefs and Dynamism (§ **3):** We show how to encode higher-level beliefs that can catch concrete bugs using succinct Datalog queries.

2. Network Optimized Datalog (§ **4):** We modify the Z3 Datalog implementation by adding new optimizations such as symbolic representation of packets and a combined Filter-Project operator. While we will attempt to convey the high-level idea, the main point is that these optimizations are crucial to scale to large data centers with 100,000s of rules. Our code is publicly released as part of Z3 so that others can build network verification tools on top of our engine.

3. Evaluation (§ **6**): We show that the generality of Network Optimized Datalog comes at reasonable speeds using existing benchmarks, synthetic benchmarks, and our own operational networks. We also report briefly on the performance of other tools such as model checkers and SAT solvers.

4. Experience (§ **7**): We describe our experiences during the last year with an existing checker called SecGuru. We also describe our experience checking for beliefs on a large data center. Finally, we allude to Batfish [15] that checks for bugs in the face of failures using our tool as a substrate.

In addition, § 2 describes our Datalog model, and § 5 describes our benchmarks.

2 Datalog Model

2.1 Why Datalog

An ideal language/tool for network verification should possess five features:

1. All Solutions: We want to find all packet headers from *A* that can reach *B*. In other words, we need *all* solutions for a reachability query. Most classical verification tools such as model checkers [21] and SAT solvers [3] only provide single solutions; the naive approach of adding the negation of the solution and iterating is too slow.

2. Packet Rewrites: Among classical verification logics, Datalog does provide a native way to model routers as relations over input and output packets. Packet reachability is modeled as a recursive relation. Rewriting few selected bits or copying a range is also simple to model in this framework.

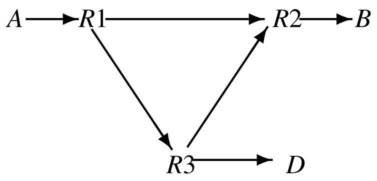
3. Large Header Spaces: Both Features 1 and 2 are challenging when the header space is very large; for example with packet headers of 80 bytes, the header space is of up to 2^{640} bits. Without a way to compress headers, naive solutions will scale poorly.

4. General Specification Language: Encoding beliefs minimally requires a language with Boolean operators for combining reachability sets, and negation to express differential queries.

5. General Modeling Language: Modeling failure response to protocols such as BGP and OSPF requires the ability to model recursion and running solutions to a fixed point as has been done by [15]. The language must also allow modeling simple failure scenarios such as availability zone failures and packet format changes.

Recent work in network verification including VeriFlow [24] and NetPlumber [22] provide all solutions, while allowing packet rewriting and scaling to large header spaces (Features 1-3). However, none of these domain-specific languages support Feature 5 (modeling any changes to network forwarding requires a change to tool internals) or Feature 4 (the specifications are typically hardcoded in the tool). While the FlowExp reachability language in NetPlumber [22] allows combination of reachability sets using regular expressions, it is fairly limited (e.g., it cannot do differential queries across paths) and is missing other higher-level features (e.g., recursion). By contrast, FlowExp queries can be encoded using Datalog, similar to the way finite automata are encoded in Datalog.

Thus, existing network verification languages [22–24, 26] fail to provide Features 4 and 5 while existing verification languages fail to provide Features 1, 2, and 3. FlowLog [28] is a language for programming controllers and seems less suited for verifying dataplanes. Out of the box Datalog is the only existing verification language that provides Features 1, 4, and 5. Natively, Datalog implementations struggle with Features 2 and 3. We deal with these challenges by overhauling the underlying Datalog engine (§ 4) to create a tool we call Network Optimized Datalog (NoD).



in	*dst*	*src*	*rewrite*	*out*
$R1$	$10\star$	$01\star$		$R2$
$R1$	$1\star\star$	$\star\star\star$		$R3$
$R2$	$10\star$	$\star\star\star$		B
$R3$	$\star\star\star$	$1\star\star$		D
$R3$	$1\star\star$	$\star\star\star$	$dst[1] := 0$	$R2$

Figure 1: $R1$ 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 $R3$. $R3$ has an ACL that drops packets from $1\star\star$. $R3$ also rewrites the middle bit in *dst* to 0. This ensures that rerouted packets reach B regardless of the value of $dst[1]$.

2.2 Modeling Reachability in NoD

NoD is a Datalog implementation optimized for large header spaces. At the language level, NoD is just Datalog. We model reachability in NoD as follows.

Figure 1 shows a network with three routers $R1$, $R2$, and $R3$, and three end-point nodes A, B and D. The routing tables are shown below the picture. For this simple example, assume packets have only 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. The last rule of Figure 1 includes a packet rewrite operation. We use $dst[1] := 0$ to indicate that position $dst[1]$ is set to 0. (Position 0 corresponds to the right-most bit.)

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.

For reachability, we only model a single (symbolic) packet starting at the source. The current location of a packet is modeled by a location predicate. For example, the predicate $R1(dst, src)$ is true when a packet with destination *dst* and source *src* is at router $R1$.

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*.

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

$$\begin{aligned}
& & & B(dst, src) \\
R1(dst, src) & &:- \quad & G_{12} \wedge Id \wedge R2(dst', src') \\
R1(dst, src) & &:- \quad & G_{13} \wedge Id \wedge R3(dst', src') \\
R2(dst, src) & &:- \quad & G_{2B} \wedge Id \wedge B(dst', src') \\
R3(dst, src) & &:- \quad & G_{3D} \wedge Id \wedge D(dst', src') \\
R3(dst, src) & &:- \quad & G_{32} \wedge Set0 \wedge R2(dst', src') \\
A(dst, src) & &:- \quad & R1(dst, src) \\
& & ? \quad & A(dst, src)
\end{aligned} \tag{2}$$

We use a shorthand for predicates that represent the matching condition in a router rule called a *guard* and for packet updates. The relevant guards and updates from Fig. 1 are in equation (1). Notice that G_{13} includes the negation of G_{12} to model the fact that the rule forwarding packets from $R1$ to $R3$ has lower priority than the one forwarding packets from $R1$ to $R2$. The update from the last rule (*Set0*) sets dst' to the concatenation of $dst[2]\ 0\ dst[0]$. Armed with this shorthand, the network of Fig. 1 can now be modeled as equation (2).

To find 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. Note how Datalog is used both as a modeling and a specification language.

Router FIBs and ACLs can be modeled by Datalog rules in a similar way. A router that can forward a packet to either $R1$ or $R2$ (load balancing) will have a separate (non-deterministic) rule for each possible next hop. We model *bounded* encapsulation using additional fields that are not used when the packet is decapsulated. Datalog queries can also check for cycles and forwarding loops. A loop detection query for an MPLS network is described below.

3 Beliefs and Dynamism in NoD

We now describe how to encode beliefs in NoD/Datalog, which either cannot be expressed succinctly, or at all, by previous work [23, 24, 26] without changing internals. While FlowExp in NetPlumber [22] allows modification of a reachability query *within a single path*, it cannot express queries *across paths* as Datalog can. We now show how to write Datalog queries for the belief templates alluded to in the introduction.

3.1 Protection sets

Consider the following belief: **Fabric managers are *not* reachable from guest virtual machines**. While this can be encoded in existing tools such as Hassel [23] and VeriFlow [24], since the guest VMs are set of size 5000, the fabric manager is a set of around 12 addresses, and the naive way to express this query is to explode the query to around 60,000 separate queries. While this can be reduced by aggregating across routers, it is still likely to take many queries using existing tools [23, 24]. While this could be fixed by adding a way to define sets in say Hassel, this requires subtle changes to the internals. Our point is that Datalog allows this to be stated succinctly in a single query by taking advantage of the power of definitions.

The compact encoding in Datalog is as follows. Let $VM(dst, src)$ denote the fact that a packet is at one of the guest virtual machines and destined to an address *dst* that belongs to the set of fabric managers. We now query (based on the rules of the network, encoded for example as in equation (2)) for a *violation* of the belief: whether the fabric manager can be reached (encoded by the predicate FM) from a customer VM.

$$\begin{array}{rcl} VM(dst, src) & :- & AddrOfVM(src), AddrOfFM(dst). \\ & ? & FM(dst, src). \end{array}$$

Datalog Features: definitions for sets of addresses.

3.2 Reachability sets

Consider the following belief: **All Fabric managers are reachable from jump boxes (internal management devices)**.

As before, we check for the corresponding violation of the belief (a bug). Namely, we can query for addresses injected from jump boxes J, destined for fabric manager FM that nevertheless do not reach FM.

$$\begin{array}{rcl} J(dst, src) & :- & AddrOfJ(src), AddrOfFM(dst). \\ & ? & J(dst, src) \wedge \neg FM(dst, src). \end{array}$$

Datalog Features: Definitions, negation.

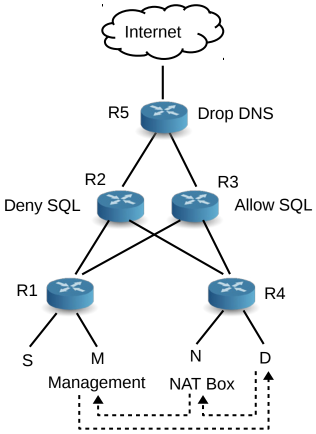


Figure 2: Bugs encountered in a cloud setting.

3.3 Equivalence of Load Balanced Paths

Consider the following bug:

Load Balancer ACL Bug: In Figure 2, an operator may set up two routers $R2$ and $R3$ that are load balancing traffic from a source S to destination D. $R2$ has an ACL entry that specifies that packets to the SQL port should be dropped but $R3$ does not. Assume that ECMP (equal cost multipath routing) [33] currently uses a hash function that routes SQL traffic via $R2$ where it is (correctly) dropped. However, the hash function can change to route packets via $R3$ and now SQL packets will (incorrectly) be let through.

In general, this bug violates a belief that *reachability across load balanced paths must be identical regardless of other variables such as hash functions*. We can check whether this belief is true by encoding a differential query encoded in Datalog. Is it possible that some packet reaches a destination under one hash function but not another? Such a query is impossible to answer using current network verification tools [22–24].

Datalog has the power needed for expressing reachability over different hashing schemes. We encode a hashing scheme as a bit vector h that determines what hashing choices (e.g., should $R1$ forward packets to D via $R2$ or $R3$ in Figure 2) are made at every router. We assume we have defined a built-in predicate *Select* that selectively enables a rule. We can then augment load balancing rules by adding *Select* as an extra guard in addition to the guards modeling the match predicate from the FIB. For example, if there are rules for routing from $R1$ to $R2$ and $R3$, guarded by G_{12}, G_{13}, then the modified rules take the form:

$$\begin{array}{rcl} R2(dst, h) & :- & G_{12} \wedge R1(dst, h) \wedge Select(h, dst). \\ R3(dst, h) & :- & G_{13} \wedge R1(dst, h) \wedge Select(h, dst). \end{array}$$

To check for inconsistent hashing, we pose a query that asks if there exists an intermediate node A at which packets to destination dst arrive using one hash assignment h_1 but are dropped using a different hash assignment h_2:

$$? \quad A(dst,h_1) \wedge \neg A(dst,h_2). \tag{3}$$

By adding an ordering across routers, the size of h can be encoded to grow linearly, not exponentially, with the path length.

Datalog Features: Negation, bit vectors, non-determinism, Boolean combinations of reachability predicates.

3.4 Locality

Consider the following bug:

Cluster reachability: In our Hong Kong data center, we found a locality violation. For example, it would be odd if packets from S to M in Figure 2 flowed through $R2$.

Putting aside the abstruse details of the wiring issues during buildout that caused this bug, the bug violates a belief that routing preserves traffic locality. For example, traffic within one rack must not leave the top-of-rack switch. Datalog definitions let us formulate such queries compactly.

Consider packets that flow between S and M in Figure 2. Observe that it would be odd if these packets flowed through $R2$, or $R3$, or $R5$ for that matter. The ability to define entire sub-scopes in Datalog comes in handy in this example. For example, we can define a predicate *DSP* (for *D*ata center *SP*ine) to summarize packets arriving at these routers:

$$\begin{aligned} DSP(dst) &\ :-\ R2(dst). \\ DSP(dst) &\ :-\ R3(dst). \\ DSP(dst) &\ :-\ R5(dst). \end{aligned}$$

Conversely, local addresses like S and M that can be reached via $R1$ can be summarized using another predicate L_{R1} (*L*ocal $R1$ addresses), that we assume is given as an explicit set of IP address ranges.

$$L_{R1}(dst) \ :-\ dst = 125.55.10.0/24.$$

Assume a packet originates at S and sends to such a local address; we ask if *DSP* is reached indicating that the packet has (incorrectly) reached the spine.

$$\begin{aligned} S(dst) &\ :-\ L_{R1}(dst). \\ &\ ?\ \ DSP(dst). \end{aligned} \tag{4}$$

Note the power of Datalog definitions. The query can also be abstracted further to check whether traffic between N and D (that sit on different racks and could be in a different cluster) have the same property in a single query, without writing separate queries for each cluster. For example, we could add rules, such as:

$$\begin{aligned} L_{R4}(dst) &\ :-\ dst = 125.75.10.0/24. \\ D(dst) &\ :-\ L_{R4}(dst). \\ N(dst) &\ :-\ L_{R4}(dst). \end{aligned}$$

This query can be abstracted further by defining locality sets and querying whether any two stations in the same locality set take a route outside their locality set.

Datalog Features: Scoping via predicates.

3.5 Dynamic Packet Headers

Going beyond encoding beliefs, we describe an example of dynamism. Network verification tools such as Hassel [23] and NetPlumber [22] support IP formats but do not yet support MPLS [31].

However, in Datalog one does not require *a priori* definitions of all needed protocols headers before starting an analysis. One can easily define new headers *post facto* as part of a query. More importantly, one can also *define new forwarding behaviors* as part of the query. This allows modeling flexible routers whose forwarding behavior can be metamorphosed at run-time [4, 5]

To illustrate this power, assume that the Datalog engine has no support for MPLS or the forwarding behavior of label stacking. A bounded stack can be encoded using indexed predicates. For example, if $R1$ is a router, then $R1^3$ encodes a forwarding state with a stack of 3 MPLS labels and $R1^0$ encodes a forwarding state without any labels. Using one predicate per control state we can encode a forwarding rule from $R5$ to $R2$ that pushes the label 2016 on the stack when the guard G holds as:

$$\begin{aligned} R2^1(dst,src,2016) &\ :-\ G, R5^0(dst,src). \\ R2^2(dst,src,l_1,2016) &\ :-\ G, R5^1(dst,src,l_1). \\ R2^3(dst,src,l_1,l_2,2016) &\ :-\ G, R5^2(dst,src,l_1,l_2). \\ Ovfl(dst,src,l_1,l_2,l_3) &\ :-\ G, R5^3(dst,src,l_1,l_2,l_3). \end{aligned}$$

We assume that l_1, l_2, l_3 are eight bit vectors that model MPLS labels. The first three rules model both MPLS label stacking procedure and format. Predicates, such as $R1^3$, model the stack size. The last rule checks for a misconfiguration that causes label stack overflow.

SWAN [19] uses an MPLS network updated dynamically by an SDN controller. To check for the belief that the SDN controller does not create any loops, we use standard methods [37]. We use a field to encode a partial history of a previously visited router. For every routing rule, we create two copies. The first copy of the rule sets the history variable h to the name of the router. The other copy of the rule just forwards the history variable. There is a loop (from *R*5 to *R*5) if *R*5 is visited again where the history variable holds *R*5. We omit details.

While router tables [19] may be in flux during updates, a reasonable belief is even during updates "rules do not overlap", to avoid forwarding conflicts. Similarly, another reasonable belief is that any rule that sends a packets out of the MPLS network should pop the *last* element from the MPLS label stack.

3.6 Middleboxes and Backup Routers

While ensuring that traffic goes through a middlebox *M* is a well-studied property [22], we found a more subtle bug across paths:

Incorrect Middlebox traversal: Once again, refer to Figure 2. A management box *M* in a newer data center in Brazil attempted to start a TCP connection to a local host *D*. Newer data centers use a private address space and internal hosts must go through a Network Address Translator (NAT) before going to external public services. The box *M* sent the TCP connection request to the private address of *D*, but *D* sends the packet to *M* via the NAT which translates *D*'s source address. TCP at *M* reset the connection because the SYN-ACK arrived with an unexpected source.

This bug violates a belief that packets should go through the same set of middleboxes in the forward and reverse path. This is a second example of an invariant that relates reachability across two different paths, in this case the forward and reverse paths, as was reachability across load balanced paths. This is easily encoded in Datalog by adding a fictitious bit to packets that is set when the packet passes through a middlebox. We omit details to save space.

Consider next the following bug:

Backup Non-equivalence: Two data centers *D*1 and *D*2 in the same region are directly connected through a border network by a pair of backup routers at the border of *D*1 and *D*2. The border routers are also connected to the core network. The intent is that if a single failure occurs *D*1 and *D*2 remain directly connected. However, we found after a single failure, because of an incorrect BGP configuration, the route from *D*1 to *D*2 went through a longer path through the core. While this is an inefficiency bug, if it is not fixed and then if the route to the core subsequently fails, then *D*1 and *D*2 lose connectivity even when there is a physical path.

This bug suggests the belief that all paths between a source and destination pair passing through any one of a set of backup routers should have the same number of hops. We omit the encoding except to note that we encode path lengths in Datalog as a small set of control bits in a packet, and query whether a destination is reached from the same source across one of the set of backup routers, but using two different path lengths. Once again this query appears impossible to state with existing network verification tools except NetPlumber [22]. NetPlumber, however, cannot handle the range of queries NoD can, especially allowing dynamic networks.

4 Network Optimized Datalog

While Datalog can express higher-level beliefs and model dynamism (Features 4 and 5 in Section 2.1) and computes all solutions (Feature 1), naive Datalog implementations struggle with Features 2 and 3 (scalably expressing large header spaces and packet rewrites). While we describe our experience with modifying μZ (the Datalog framework in Z3), there are two general lessons that may apply to other Datalog tools: the need for a new table data structure to compactly encode large header spaces, and a new Select-Project operator. We will try to convey the high-level ideas for a networking reader.

4.1 Compact Data Structures

One can pose reachability queries to μZ (our Datalog framework) to compute the set of packets that flow from *A* to *B*. Think of the Datalog network model as expressing relations between input and output packets at each router: the router relation models the forwarding behavior of the router including all forwarding rules and ACLs. The router is modeled not as a function but as a relation, to allow multicast and load balancing, where several output packets can be produced for the same input packet. Whenever the set of input packets at a router change, the corresponding set of output packets are recomputed using the router relation. Thus for a network, eventually sets of packets will flow from the inputs of the network to the endpoints of the network.

The main abstract data structure to encode a relation in Datalog is a table. For example, a table is used to store the set of input packets at a router, and a table is used to store output packets. To update the relationship between input and output tables at a router, under the covers, μZ executes Datalog queries by converting them into relational algebra, as described elsewhere [9]. Networking readers can think of μZ as providing a standard suite of database operators such as *select*, *project* and *join* to manipulate *tables* representing sets of packet headers in order to compute reachability sets. Figure 3 – which represents a single router that drops HTTP packets using an ACL that drops port 80 packets – makes this clearer.

In our toy example, Figure 3, assume that the set of packets that reach this router have source addresses whose first bit is 1 because of some earlier ACLs. Thus the set of packet headers that leave the router are those that have first bit 1 and whose TCP destination port is

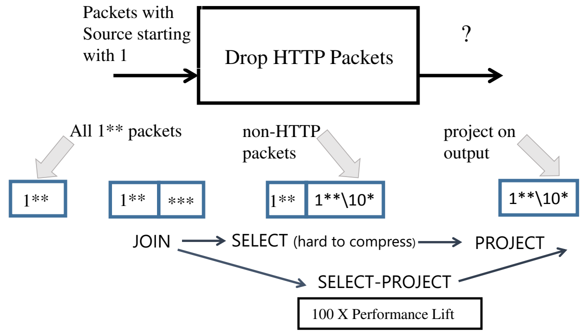


Figure 3: Manipulating tables of packet headers with a combined select-project operator.

not 80. As we have seen, the natural way for Datalog to represent a set of packet headers is as a *table*. Representing all source addresses that start with a 1 would require 2^{127} rows if 128-bit packet headers are represented by arrays. None of the existing table data structures (encapsulated in what are called backends) in μZ performed well for this reason. Hence, we implemented two new table backends.

The first backend uses BDDs (Binary Decision Diagrams [7]) for Datalog tables. BDDs are a classic data structure to compactly represent a Boolean function, and widely used in hardware verification to represent circuits [8]. A classic program analysis paper also augments Datalog with BDDs [34] for program analysis, so our use of BDDs for Datalog tables is natural.

The second backend is based on ternary bit vectors, inspired by Header Space Analysis (HSA) [23, 24], but placed in a much more general setting by adding a new data structure to Datalog. This data structure we added was what we call *difference of cubes* or DoC. DoC represents sets of packets as a *difference* of ternary strings. For example, $1\star\star \setminus 10\star$ succinctly represents all packets that start with 1 other than packets that start with 10. Clearly, the output of Figure 3 can be compactly represented as the set difference of all $1\star\star$ packets and all packets whose destination port is 80.

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 that have dependencies. For example, the second rule in Figure 1 takes effect only if the first rule does not match. More precisely, difference of cubes is particularly efficient at representing formulas of the form $\varphi \wedge \neg\varphi_1 \wedge \cdots \wedge \neg\varphi_n$, with φ and φ_i of the form $\bigwedge_i \phi_i$ and ϕ_i having no Boolean operators. This 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. Again, in networking terminology dependencies of a forwarding rule or ACL are all higher priority matching rules.

Code: The Datalog backends added to Z3 were implemented in C++. The BDD backend takes 1,300 LoC, and the difference of cubes backend takes almost 2,000 LoC.

4.2 Combining Select and Project

We needed to go beyond table compression in order to speedup Datalog's computation of reachability sets as follows. Returning to Figure 3, μZ computes the set of output packets 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 input packets and produces the set of output packets O.

Thus, in Figure 3, the output of the first *join* is the set of all possible input packets with source addresses that start with 1, together with all possible output packets. While this sounds like a very indirect and inefficient path to the goal, this is the natural procedure in μZ. Joins are the only way to create a new relation, and to avail of the powerful set of operators that work on relations. While the join with all possible output packets A appears expensive, A is compactly represented as a single ternary string/cube and so its cost is small.

Next, after the *select*, the relation is "trimmed" to be all possible input packets with source bit equal to 1, together with all output packets with source bit equal to 1 and destination port not equal to 80. Finally, in the last step, the *project* step removes all columns corresponding to the input packets, resulting in the correct set of output packets (Figure 3) as desired. Observe that the output of the *join* is easily compressible (a single ternary string) and the output of the final *project* is also compressible (difference of two ternary strings).

The elephant in the room is the output of the *select* which is extremely inefficient to represent as a BDD or as a difference of ternary strings. But the output of the *select* is merely a way station on the path to the output; so we do not need to explicitly materialize this intermediate result. Thus, we define a new combined *select-project* operator whose inputs and outputs are both compressible. This is the key insight, but making it work in the presence of packet rewriting requires more intricacy.

4.3 Handling Packet Rewriting

The example in Figure 3 does not include any packet rewrites. In Figure 1, however, rule R_3 rewrites the first destination address bit to a 0. Intuitively, all bits *except* the first are to be copied from the input packet to the output. While this can be done by a brute force enumeration of all allowed bit vectors for this 6-bit toy example, such an approach does not scale to 128 bit headers. We need, instead, an efficient way to represent copying constraints.

Back to the toy example, 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 using a generalized select). Since we know that the 5th rule in Figure 1 can only apply if 4th rule does not match, we know that in addition to the first destination address bit being 1, the "negation of the dependencies" requires that bit 2 of the source address should also be 0.

One might be tempted to believe that $1\star\star\star\star\star10\star0\star\star$ compactly represents the input-output relation as "All input packets whose second destination address bit is 1" (the first six bits) together with "All output packets for which bit 2 of destination address bit is 1, bit 1 of the destination address has been set to 0, and for which bit 2 of the source address bit is 0". But this incorrectly represents the copying relation! For example, it allows input packets where bit 1 of the source address of the input packet is a 0, but bit 1 of the source address of the output packet is a 1. Fortunately, we can rule out these exceptions fairly compactly using set differences which are allowed in difference of cubes notation. We obtain the following expression:

$$\begin{aligned}
&1\star\star\star\star\star10\star0\star\star \setminus\\
&(\star\star0\star\star\star\star\star1\star\star\star\cup\star\star1\star\star\star\star\star0\star\star\star\cup\\
&\ \star\star\star\star0\star\star\star\star\star1\star\cup\star\star\star\star1\star\star\star\star\star0\star\cup\\
&\ \star\star\star\star\star0\star\star\star\star\star1\cup\star\star\star\star\star1\star\star\star\star\star0)
\end{aligned}$$

While this looks complicated, *the key idea is efficiently representing copying using a difference of ternary strings*. The unions in the difference are ruling out cases where the "don't care" $\star$ bits are not copied correctly. The first term in the difference states that we cannot have bit 0 of destination address bit be a 0 in the *input* packet and bit 0 of 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* whose bits are being copied.

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 $10\star0\star\star$. The final result is significantly smaller than the intermediate result, and this effect is much more pronounced when we use 128 bit headers!

Generalizing: To make select-project efficient, we need to compute the projection implicitly without explicitly materializing intermediate results. 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 besides networking. In verification terminology, this operation corresponds to computing the strongest post-condition of the transition relation. However, it takes some delicacy to implement such an operator in a general verification engine such as Z3 so it can be used in other domains besides network verification.

Code: Besides select-project, we made several additional improvements to the Datalog solver itself, which were released with Z3 4.3, reducing Z3's memory usage in our benchmarks by up to 40% .

5 Benchmarks

We use four benchmarks:

Stanford: This is a publicly available [32] snapshot of the routing tables of the Stanford backbone, and a set of network reachability and loop detection queries. The core has 16 routers. The total number of rules across all routers is 12,978, and includes extensive NAT and VLAN support.

Generic Cloud Provider: We use a parameterizable model 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. Data centers are interconnected by an all-to-all one-hop mesh network. Parameters include replication factor, router ports, data centers, machines per data center, VMs per machine, and number of services. These parameters can be set to instantiate small, medium, or large clouds. This benchmark is publicly available [29].

Production Cloud: Our second source comes from two newer live data centers located in Hong Kong and in Singapore whose topology is shown in Figure 4. They

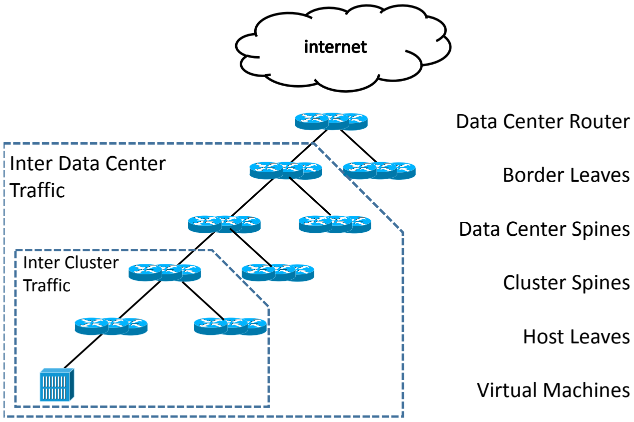


Figure 4: Production Cloud Layout

consist of a hundred routers each ranging over border leaves, data center and cluster spines, and finally top-of-rack switches. In the Hong Kong data center, each router has roughly 2000 ECMP forwarding rules adding up to a combined 200K rules. In the Singapore data center there are about 820K combined forwarding rules. The text files for these rules take from 25MB to 120MB for the respective data-centers.

This infrastructure services three clusters and thousands of machines. We extracted routing tables from the Arista and Cisco devices using a `show ip route` command. This produced a set of routing tables including the ECMP routing options in Datalog format. To handle longest prefix match semantics, the translation into Datalog uses a trie to cluster ranges with common prefixes, avoiding redundancy in the Datalog rules. The range of private and public IP addresses assigned to each cluster was extracted from a separate management device.

Experimental Backbone: We check an SDN backbone based on the SWAN design [19]. To maximize bandwidth utilization, the SDN controller periodically recomputes routing rules that encapsulate packets with an MPLS labels stack encoding tunnels. Our tool takes the output from the controller: a set of routing tables, a network topology and configurations that map IP addresses to end-points. We check selected beliefs, such as loop freedom and absence of stack overflow.

Experimental Toolkit: In addition to the publicly available Hassel C code [16], for comparison we used two classic model checking algorithms (BMC and PDR), a Datalog framework μZ [18], and a state-of-the-art SAT/SMT solver, all implemented in the Z3 [11] engine. Our engines are publicly available [29] so other researchers can extend our results.

6 Evaluation

We describe results for belief checking on a production cloud IP network and an experimental MPLS backbone, followed by differential reachability queries on a cloud benchmark. Finally, we compare NoD's performance with existing tools.

6.1 Protection Sets in a Production Cloud

We checked whether two policies based on the Protection sets template (see Table 1) hold in the Singapore data center. The two queries were to verify that neither Internet addresses or customer VMs can access the protected fabric controllers for security reasons.

Both experiments took around 12 minutes. While this may seem slow, the sets of addresses are very large. For the power of exploring a very general belief, the performance seems acceptable and easily incorporated in a checker that runs every hour.

Both queries failed; thus the beliefs of the network operators were incorrect. Closer inspection showed that these were not network bugs, but incorrect beliefs. There are two ranges of IPs of fabric controllers that are supposed to be reachable (and they are), and certain ICMP packets are also allowed to flow to fabric controllers. Although no network bugs were found, these queries allowed the operator to refine his beliefs.

6.2 Reachable Sets on a Production Cloud

As in the previous experiment, we also checked whether two policies from the reachable sets template hold in the same Singapore data center. The first query checked if all of "utility boxes" can reach all "fabric controllers". The second query is similar, but checks whether "service boxes" can reach "fabric controllers". The first query took around 4 minutes to execute, while the second took 6 minutes. Both queries passed successfully, confirming operator beliefs.

6.3 Locality on a Production Cloud

Figure 4 shows our public cloud topology which has more levels of hierarchy than the simple example of Figure 2. These levels motivate more general queries than the simple locality query (4) used in [§3.4].

Figure 4 also shows the boundaries of traffic locality in our public cloud. Based on these boundaries, we formulate the following queries to check for traffic locality. First, C2C requires that traffic within a cluster not reach beyond the designated cluster spines. Next, B2DSP, B$\overline{2}$DSP, B2CSP, and B$\overline{2}$CSP require that traffic targeting public addresses in a cluster must reach only designated data center spines, not reach other data center spines belonging to a different DC, must reach only designated cluster spines, and not reach cluster spines belonging to other clusters, respectively.

Query	Cluster 1	Cluster 2	Cluster 3
C2C	12 (2)	13 (2)	11 (2)
B2DSP	11 (2)	11 (2)	11 (2)
B$\bar{2}$DSP	3 (1)	4 (1)	4 (1)
B2CSP	11 (2)	11 (2)	11 (2)
B$\bar{2}$CSP	11 (2)	12 (2)	11 (2)

Table 2: Query times are in seconds, times in parentheses are for a snapshot where only one of the available ECMP options is part of the model (20% the size of the full benchmarks).

Table 2 shows the time spent for these five different kinds of queries over three clusters. They check correctness of routing configurations in all network devices.

For the Hong Kong cluster we identified some violated locality queries due to bugs during buildout. An example output for a C2C query was 100.79.126.0/23 suggesting that an address range got routed outside the cluster it was said to be part of. On the other hand, our tool produced public address ranges from a B2CSP query which were supposed to reach the cluster spine but did not: $192.114.2.62/32 \cup \ldots \cup 192.114.3.48/29$.

6.4 An Experimental MPLS backbone

We checked loop-freedom, absence of black holes, and rule disjointness on configurations for an experimental backbone based on the SWAN design [19]. Note the versatility of Datalog and its ability to model dynamic forwarding behaviors without changing tool internals. Both packet formats and forwarding behaviors were completely different from those in the production cloud queries. We report on a selected experiment with a configuration comprising of 80 routers, for a total of 448 forwarding rules. Modelling label stacks required 3400 Datalog rules over a header-space of 153 bits. The loop check takes a fraction of a second for the configurations we checked. Checking for black holes takes under 5 seconds, identifying 56 flows out of 448 as black holes. The same configuration had 96 pairs of overlapping rules, which were enumerated in less than 1 second. This experience bodes well for scaling validation to larger backbone sizes.

6.5 Differential Reachability

We investigate the performance of differential reachability queries on our synthetic cloud benchmarks (topology similar to Figure 2).

We performed two experiments by taking the Medium Cloud topology as baseline, and changing the ACLs at one of the core routers such that one of the links in a set of load balanced paths allowed VLAN 3 and blocked VLAN 1, while all other links blocked VLAN 1 and allowed VLAN 3. We then checked the difference in reachability across all load-balanced paths between the Internet and a host in the data center.

This query took 1.9 s, while the equivalent reachability query takes 1.0 s (90% run time increase). Repeating the same query, but measuring differential reachability between two hosts located in different data centers (resulting in longer paths) took 3.1 s, while the equivalent reachability queries takes 1.1 s (182% run time increase).

We note that the run time increase between vanilla reachability queries and differential queries will likely increase with the amount of differences, but in practice the number of differences (i.e., bugs) should be small.

6.6 Comparison with existing Tools

We ran the benchmarks with multiple tools to benchmark our speeds against existing work. Besides the difference of cubes backend described in § 4, we also used a BDD-based Datalog backend, a bounded model checker (BMC) [10], an SMT solver that uses an unrolled representation of the network as in [37], and a state-of-the-art solver based on the IC3 [6, 17] algorithm. The benchmarks were run on a machine with an Intel Xeon E5620 (2.4 GHz) CPU. We also used an SMT algorithm that was modified to return all solutions efficiently. The model checkers, however, return only one solution so the speed comparison is not as meaningful.

Table 3 is a small sampling of extensive test results in [29]. It shows time (in seconds) to run multiple tools on a subset of the Stanford benchmarks, including reachable and unreachable queries, and a loop detection query. The tests were given a timeout of 5 minutes and a memory limit of 8 GBs.

Note that the model checkers only compute satisfiability answers (i.e., "is a node reachable or not?"), while Datalog computes all reachable packets. For SMT, we provide results for both type of queries. All the SMT experiments were run 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 running time. We do not provide the SMT running times for the cloud benchmarks, since our AllSAT algorithm does not support non-determinism. We were unable to run Hassel C [16] on the cloud benchmarks, since Hassel C has hardwired assumptions, such as router port numbers following a specific naming policy.

The first takeaway is that NoD is faster at computing *all solutions* than model checkers or SAT solvers are at computing *a single solution*. Model checking performance also seems to degrade exponentially with path length (see row 3 versus row 2 where the model checkers run out of memory). Similarly, unrolling seems to exact

Test	Model Checkers		SMT		Datalog		Hassel C
	BMC	PDR	Reach.	All sols.	BDDs	DoC	
Small Cloud	0.3	0.3	0.1	–	0.2	0.2	–
Medium Cloud	T/O	10.0	0.2	–	1.8	1.7	–
Medium Cloud Long	M/O	M/O	4.8	–	7.4	7.2	–
Cloud More Services	7.2	8.5	12.5	–	5.3	4.8	–
Large Cloud	T/O	M/O	2.8	–	16.1	15.7	–
Large Cloud Unreach.	T/O	M/O	1.1	n/a	16.1	15.7	–
Stanford	56.2	13.7	11.5	1,121	6.6	5.9	0.9
Stanford Unreach.	T/O	12.2	0.1	n/a	2.6	2.1	0.1
Stanford Loop	20.4	11.7	11.2	290.2	6.1	3.9	0.2

Table 3: Time (in seconds) taken by multiple tools to solve network benchmarks. Model checkers only check for satisfiability, while Datalog produces reachability sets. T/O and M/O are used for time- and memory-out.

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.

Hassel C takes under a second for Stanford, faster than Datalog. NetPlumber [22] and VeriFlow [24] are even faster for incremental analysis. Further, Yang and Lam [36] use predicate abstraction to further speed up reachability testing. However, none of these tools have the ability to model higher-level beliefs and model dynamic networks as NoD can. NoD speeds are also acceptable for an offline network checker, the major need today.

7 Experience

SecGuru: The SecGuru [20] tool has been actively used in our production cloud. In continuous validation, SecGuru checks policies over ACLs on every router update as well as once a day, doing over 40,000 checks per month, where each check takes 150-600 ms. It uses a database of predefined common beliefs. For example, there is a policy that says that SSH ports on fabric devices should not be open to guest VMs. While these policies rarely change, IP addresses do change frequently, which makes SecGuru useful as a regression test. SecGuru had a measurable impact in reducing misconfigurations during build-out, raising an average of one alert per day, each identifying ∼16K faulty addresses. SecGuru also helped reduce our legacy corporate ACL from roughly 3000 rules to 1000 without any business impact.

Belief Refinement: As described in § 6.1, our operator's belief in a protection set policy (customer VMs cannot reach fabric controllers) was subtly incorrect. The correct belief required new reachability exceptions. We currently code this at the level of Datalog in lieu of a GUI interface to the 5 policy templates from Table 1.

Batfish: Batfish [15] can find whether two stations remain reachable across any set of single link failures by modeling OSI and BGP. Batfish uses NoD for reachability analysis because it is more expressive than other tools. Batfish also uses the Z3 constraint solver to find concrete packet headers, confirming our intuition that supplying NoD in a general tool setting allows unexpected uses.

8 Conclusion

Network Optimized Datalog (NoD) is more expressive than existing tools in its ability to encode beliefs to uncover true specifications, and to model network dynamism without modifying internals. NoD is much faster than existing (but equally expressive) verification tools such as model checkers and SMT solvers. Key to its efficiency are ternary encoding of tables and a Select-Project operator.

By contrast, current network verification tools operate at a low level of abstraction. Properties such as cluster scoping (Figure 4) expressible in a single Datalog query would require iterating across all source-destination pairs. Further, existing work cannot easily model new packet formats, new forwarding behaviors, or failures [15]. We were pleased to find we could model MPLS and label stacking succinctly. It took a few hours to write a query to find loops in SWAN.

As in [13], our work shows how fragile the understanding of the true network specification is. Even when working with an experienced network operator, we found that simple beliefs (e.g., no Internet addresses can reach internal controllers) had subtle exceptions. The belief templates in Table 1 abstract a vast majority of specific checks in our network and probably other networks. A GUI interface for belief templates will help operators.

If network verification is to mature into a networking CAD industry, its tools must evolve from ad hoc software into principled and extensible techniques, built upon common foundations that are constantly being improved. We suggest NoD as a candidate for such a foundation.

References

[1] E. Al-Shaer and S. Al-Haj. FlowChecker: configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

[2] T. Benson, A. Akella, and D. A. Maltz. Mining policies from enterprise network configuration. In *IMC*, 2009.

[3] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[5] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, 2013.

[6] A. R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, 2011.

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

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, 1990.

[9] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, Mar. 1989.

[10] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, 2001.

[11] L. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[12] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI*, 2000.

[13] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *SOSP*, 2001.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI*, 2002.

[15] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, 2015.

[16] Hassel C. `https://bitbucket.org/peymank/hassel-public/`.

[17] K. Hoder and N. Bjørner. Generalized property directed reachability. In *SAT*, 2012.

[18] K. Hoder, N. Bjørner, and L. De Moura. μZ: an efficient engine for fixed points with constraints. In *CAV*, 2011.

[19] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *SIGCOMM*, 2013.

[20] K. Jayaraman, N. Bjørner, G. Outhred, and C. Kaufman. Automated Analysis and Debugging of Network Connectivity Policies. Technical Report MSR-TR-2014-102, Microsoft Research, July 2014.

[21] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, Oct. 2009.

[22] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *NSDI*, 2013.

[23] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: static checking for networks. In *NSDI*, 2012.

[24] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: verifying network-wide invariants in real time. In *NSDI*, 2013.

[25] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. RFC 7348, Aug. 2014.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with Anteater. In *SIGCOMM*, 2011.

[27] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The Margrave tool for firewall analysis. In *LISA*, 2010.

[28] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *NSDI*, 2014.

[29] Network verification website. `http://web.ist.utl.pt/nuno.lopes/netverif/`.

[30] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *SIGCOMM*, 2013.

[31] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. RFC 3031, Jan. 2001.

[32] Stanford benchmark. `https://bitbucket.org/peymank/hassel-public/src/697b35c9f17ec74ceae05fa7e9e7937f1cf36878/hassel-c/tfs/`.

[33] D. Thaler and C. Hopps. Multipath issues in unicast and multicast next-hop selection. RFC 2991, Nov. 2000.

[34] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.

[35] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. G. Greenberg, G. Hjálmtýsson, and J. Rexford. On static reachability analysis of IP networks. In *INFOCOM*, 2005.

[36] H. Yang and S. Lam. Real-time verification of network properties using atomic predicates. In *ICNP*, 2013.

[37] S. Zhang, S. Malik, and R. McGeer. Verification of computer switching networks: An overview. In *ATVA*, 2012.