Checking Cloud Contracts in Microsoft Azure

Nikolaj Bjørner[1] and Karthick Jayaraman[2]

[1] Microsoft Research
nbjorner@microsoft.com
[2] Microsoft Azure
karjay@microsoft.com

Abstract. *Cloud Contracts* capture architectural requirements in data-centers. They can be expressed as logical constraints over configurations. Contract violation is indicative of miss-configuration that may only be noticed when networks are attacked or correctly configured devices go off-line. In the context of Microsoft Azure's data-center we develop contracts for (1) network access restrictions, (2) forwarding tables, and (3) BGP policies. They are checked using the SecGuru tool that continuously monitors configurations in Azure. SecGuru is based on the Satisfiability Modulo Theories solver Z3, and uses logical formulas over bit-vectors to model network configurations. SecGuru is an instance of applying technologies, so far developed for program analysis, towards networks. We claim that *Network Verification* is an important and exciting new opportunity for formal methods and modern theorem proving technologies. Networking is currently undergoing a revolution thanks to the advent of highly programmable commodity devices for network control, the build out of large scale cloud data-centers and a paradigm shift from network infrastructure as embedded systems into software controlled and defined networking. Tools, programming languages, foundations, and methodologies from software engineering disciplines have a grand opportunity to fuel this transformation.

1 Introduction

Modern large-scale cloud infrastructures are inherently complex to configure and deploy: Network access restrictions are enforced at multiple points, forwarding and filtering policies are programmed or configured in various formats targeting devices that span different vendors and generations. Access restrictions evolve with organizational changes and operators come and leave. The general problem of analyzing networks is daunting, inhumane, and can really only be solved using rigorous tools.

As part of an arsenal of tools to take on this challenge we show in very recent work [11] how policies can be checked through a set of *beliefs*. Beliefs are confirmed, refuted or refined by posing queries about the network. Checking beliefs in packet switched networks without having any architectural knowledge of the network requires solving Boolean combinations of reachability properties. To make general belief checking practical we developed a general purpose Network

Optimized Datalog engine that scales to data-center sized networks and used in many different contexts: Datalog over bit-vectors is very general and not confined to reasoning about IPv4, IPv6 networks, but applies also to Multi-protocol Label Switching (MPLS) networks, other arbitrary packet formats and is adaptable to many scenarios. The Network Optimized Datalog engine is also available with Z3.

This paper takes a narrower perspective to a specialized but very important deployment scenario. We describe a system currently used in production, based on technologies that have been matured for some time. Our starting point is a carefully designed infrastructure, Microsoft Azure, where relevant properties are articulated already in well-motivated design goals. These principles can be captured using a set of high-level *contracts* that are enforced throughout the life-cycle of a deployment. This scenario allows us to take advantage of the properties we know of the data-center architecture to pose queries that are solved using specialized logics and efficient, well-established, reasoning engines. The flip-side is that the contracts we present in this paper do not translate to arbitrary scenarios. The scale and economic significance of Microsoft Azure, however, motivates our specialized solution, and we postulate that many of the techniques we describe here are of general interest. The three sources of cloud contracts for Azure's data-center networks are: (1) network access restrictions, (2) forwarding tables, and (3) Border Gateway Protocol (BGP) policies.

Many contracts can be captured in fragments of first-order logic. In this context, we describe the SecGuru tool that checks cloud contracts in the Microsoft Azure public cloud infrastructure. The tool is based on the Satisfiability Modulo Theories solver Z3 [6]. SecGuru models network configurations using quantifier-free logical formulas over bit-vectors. SecGuru's checking of network access restrictions is a subject of a separate paper. It is described in detail in [8]. We will here recall the highlights of network access restriction checking, and then develop our newer extensions for checking forwarding tables and BGP policies.

Network Verification is an exciting new area for both modern networking and verification technologies. In the broader context, networking is undergoing a revolution thanks to new highly programmable commodity devices for network control, the build out of large scale cloud data-centers and a paradigm shift from network infrastructure as embedded systems into software controlled and defined networking. The latter begs for the attention of techniques developed hitherto for software engineering and CAD disciplines. Many techniques can be adapted in a straight-forward way to modern packet switched networking, but many more problems require new techniques and new ideas. It is also useful to appreciate the differences of the correctness, business and deployment models for networking, hardware and software. Bugs that ship in silicone are in the best case fixed by firmware updates, and worst case by a costly recall; software bugs can be addressed by periodic updates, but is vulnerable to the update distribution process and adaption; cloud networking, in our context, is run as a (web) service and bugs lead to outages and missed service level agreements (SLAs). The dynamics are different: hardware designs are driven by advances in circuitry that

enables more complex designs; large software systems have to deal with features, legacy and interoperability; cloud services are constrained by capacity, energy requirements and current complexities are in part due to a heavy churn in new technologies and the challenges of deploying large scale distributed monitoring services. Lessons from the last two decades of static software analysis also include the often referenced obstacles of the *false positives*, or even for *true positives*, the practical obstacles and business impacts of fixing bugs[3]. The class of bugs we address in networking seems to have a somewhat different flavor: each alert that SecGuru raises is directly actionable. The cost of ignoring alerts translates to opening a network to attacks, missed revenue (by breaking SLAs), decreased performance, increased costs, and missing out of a competitive advantage. The advantages of SecGuru are on the other hand, reduced time for building out new data-centers, allowing more sophisticated and hence complex policies, making the service auditable and even using a successful report to save precious time by ruling out miss-configuration as a culprit during live site incidents.

Sections 2, 3 and 4 describe cloud contracts for access control lists, routing tables and BGP policies, respectively. The material in Section 2 is described in more detail in [8]. Section 3 extends the summary from [3] to discuss the impact of checking routing configurations. Section 4 describes select BGP contracts for configurations that we check statically. Contracts for the three configuration sources are now checked on a continuous basis in Microsoft Azure by the SecGuru tool. Section 5 provides background on the SMT solver Z3, reflects on broader opportunities around Network Verification.

2 Access Control Lists

The SecGuru [8] tool has been actively used in our production cloud in the past years for continuous monitoring and validation of Azure. It has also been used for maintaining legacy edge ACLs. In continuous validation, SecGuru checks policies over ACLs on every router update as well as once a day. This is more than 40,000 checks per month, where each check takes 150-600 ms. It uses a database of predefined contracts. For example, there is a policy that says that SSH ports on fabric devices should not be open to guest virtual machines. While these policies themselves rarely change, IP addresses do change frequently, which makes using SecGuru as a regression test useful. SecGuru had a measurable positive impact in prohibiting policy misconfigurations during build-out, raising in average of one alert per day; each identifies 3-5 buggy address ranges in the /20 range, e.g., ˜16K faulty addresses. We also used SecGuru to reduce our legacy corporate ACL from roughly 3000 rules to 1000 without any derived outages or business impact.

In more detail, the Azure architecture enforces network access restrictions using ACLs. These are placed on multiple routers and firewalls in data-centers and on the edge between internal networks and the internet. Miss-configurations, such

[3] A classical issue in the in context of static tools, such as Prefix and Coverity, e.g., see http://popl.mpi-sws.org/2014/andy.pdf for an insightful discussion.

as miss-configured ACLs, are a dominant source of network outages. SecGuru translates the ACLs into a logical predicate over packet headers that are represented as bit-vectors. These predicates are checked for containment and equivalence with contracts that are represented as other bit-vector formulas. For illustration, representative contracts are of the form:

Cloud Contract 1 *DNS ports on DNS servers are accessible from tenant devices over both TCP and UDP.*

Cloud Contract 2 *The SSH ports on management devices are inaccessible from tenant devices.*

The routers that are dedicated to connect internal networks to the Internet backbone are called Edge routers and they enforce restrictions using ACLs. Figure 1 provides a canonical example of an Edge ACL. The ACL in this example is authored in the Cisco IOS language. It is basically a set of rules that filter IP packets. They inspect header information of the packets and the rules determine whether the packets may pass through the device.

```
1   remark Isolating private addresses
2   deny ip 10.0.0.0/8 any
3   deny ip 172.16.0.0/12 any
4   deny ip 192.0.2.0/24 any
5   ...
6   remark Anti spoofing ACLs
7   deny ip 128.30.0.0/15 any
8   deny ip 171.64.0.0/15 any
9   ...
10  remark permits for IPs without
11         port and protocol blocks
12  permit ip any 171.64.64.0/20
13  ....
14  remark standard port and protocol
15         blocks
16  deny   tcp any any eq 445
17  deny   udp any any eq 445
18  deny   tcp any any eq 593
19  deny   udp any any eq 593
20  ...
21  deny   53 any any
22  deny   55 any any
23  ...
24  remark permits for IPs with
25         port and protocol blocks
26  permit ip any 128.30.0.0/15
27  permit ip any 171.64.0.0/15
28  ...
```

Fig. 1. An Edge Network ACL configuration.

Each rule of a policy contains a packet filter, and typically comprises two portions, namely a traffic expression and an action. The traffic expression specifies a range of source and destination IP addresses, ports, and a protocol specifier. The expression 10.0.0.0/8 specifies an address range 10.0.0.0 to 10.255.255.255. That is, the first 8 bits are fixed and the remaining 24 (= 32-8) are varying. A wild card is indicated by *Any*. For ports, *Any* encodes the range from 0 to $2^{16} - 1$. The action is either *Permit* or *Deny*. They indicate whether packets matching

the range should be allowed through the firewall. This language has the first-applicable rule semantics, where the device processes an incoming packet per the first rule that matches its description. If no rules match, then the incoming packet is denied by default.

The meaning of network ACLs can be captured in logic as a predicate ACL over variables *src*, a source address and port, *dst*, a destination address and port, and other parameters, such as protocol and TCP flags. For our example from Figure 1, we can capture the meaning as the predicate:

$$
\begin{array}{l}
ACL \equiv \\
\quad \textbf{if } src = 10.0.0.0/8 \ \wedge \ proto = 6 \textbf{ then } false \textbf{ else} \\
\quad \textbf{if } src = 172.16.0.0/12 \ \wedge \ proto = 6 \textbf{ then } false \textbf{ else} \\
\quad \textbf{if } src = 192.0.2.0/24 \ \wedge \ proto = 6 \textbf{ then } false \textbf{ else} \\
\quad \ldots \\
\quad \textbf{if } dst = 171.64.64.0/20 \ \wedge \ proto = 6 \textbf{ then } true \textbf{ else} \\
\quad \ldots \\
\quad \textbf{if } proto = 4 \ \wedge \ dstport = 445 \textbf{ then } false \textbf{ else} \\
\quad \ldots
\end{array}
$$

For ease of readability, we re-use the notation for writing address ranges. In bit-vector logic we would write the constraint $src = 10.0.0.0/8$ as $src[31:24] = 10$, e.g., a predicate that specifies the 8 most significant bits should be equal to the numeral 10 (the bit-vector 00000110).

Traffic is permitted by an ACL if the predicate ACL is true. Traffic permitted by one ACL and denied by another is given by $ACL_1 \oplus ACL_2$ (the exclusive or of ACL_1 and ACL_2). The SecGuru tool uses the encoding of ACLs into bit-vector logic and poses differential queries between ACLs to find differences between configurations. It also checks contracts of ACLs by posing queries of the form $ACL \ \Rightarrow \ Property$, where an example property is that UDP ports to DNS servers are allowed. The main technological novelty in SecGuru is an enumeration algorithm for compactly representing these differences. Compact representation of differences help network operators understand the full effect of a miss-configuration.

3 Routing Tables

We recently added new capabilities to SecGuru. Most notably, using lessons from our work on the more powerful Network Optimized Datalog engine, we developed techniques for checking reachability properties for routing tables in Azure. Routers in Azure are configured to follow a specific layered data-center architecture that we describe in more detail below. Figure 2 shows a schematic overview of Azure's data-centers are configured. It is a variant of the VL2 architecture [7]. In this architecture, each rack (at the bottom) is a top-of-rack switch that relays packets from the rack to a hierarchy of routers above. The hierarchy provides redundant routes to other racks within the a group called a *cluster* and to other racks belonging to other clusters, and external traffic is routed to and from the internet.

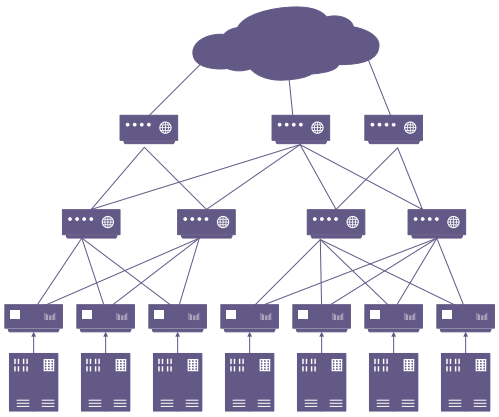

Fig. 2. Schematic overview of data-center routes.

While the architecture is fixed, each data-center is deployed using a different number of machines and clusters. Data-centers grow and shrink when tenants are migrated between machines and assumptions on the topologies change when new technologies are deployed. Thus, there are ample of opportunities for miss-configuring routers in spite of the overall fixed design. It may be entirely possible to miss-configure all but one router in a redundancy group and only observing the mistake when the correctly configured router goes off-line. Our tool checks routers from Azure networks. It catches any such miss-configurations and at the same time provides a certificate when routers are configured correctly. The latter is indispensable for operators when trouble-shooting live site incidents - knowing which parts are healthy saves precious time and resources. SecGuru retrieves a very significant amount of data from routers: Each router has a few thousand rules and each data-center can have between dozen and a few hundred routers. Some 500GB of routing tables are retrieved and checked for contracts on a daily basis.

Figure 3 shows an excerpt of a routing table from an Arista network switch Similarly to ACLs we can model routing tables as relations *Router* over destination addresses and next-hop ports that can be represented as atomic Boolean predicates. Each rule in the routing table is either provisioned based on static

```
1    B E    0.0.0.0/0 [200/0] via 100.91.176.0, n1
2                              via 100.91.176.2, n2
3
4    B E    10.91.114.0/25 [200/0] via 100.91.176.125, n3
5                                   via 100.91.176.127, n4
6                                   via 100.91.176.129, n5
7                                   via 100.91.176.131, n6
8    B E    10.91.114.128/25 [200/0] via 100.91.176.125, n3
9                                     via 100.91.176.131, n6
10                                    via 100.91.176.133, n7
11   ...
```

Fig. 3. A BGP routing table.

configurations specified in the device, or derived based on BGP network announcements that the device receives.

We here choose an encoding of *Router*, such that for each destination address dst and next-hop address n:

$$\begin{array}{l} \quad \mathit{Router}[\mathit{dst} \mapsto \mathsf{dst}, \mathsf{n} \mapsto \mathit{true}] \text{ is true} \\ \text{iff} \\ \quad \mathsf{n} \text{ is a possible next hop for address } \mathsf{dst} \end{array}$$

The routing tables have an ordered interpretation, wherein rules whose destination prefixes are the longest applies first. The default rule with mask 0.0.0.0/0, listed first, applies if no other rule applies. For our example, our chosen encoding of the predicate *Router* is of the form:

$$\begin{array}{l} \mathit{Router} \equiv \\ \quad \textbf{if } \ldots \\ \quad \textbf{if } \mathit{dst} = 10.91.114.128/25 \textbf{ then } n_3 \vee n_6 \vee n_7 \textbf{ else} \\ \quad \textbf{if } \mathit{dst} = 10.91.114.0/25 \textbf{ then } n_3 \vee n_4 \vee n_5 \vee n_6 \textbf{ else} \\ \quad n_1 \vee n_2 \end{array}$$

Each Azure data-center is built up around a hierarchy of routers that facilitate high-bandwidth traffic in and out as well as within the data-center. Traffic that leaves and enters the data-center traverses four layers of routers, while traffic within the data-center may traverse only one, two or at most three layers depending on whether the traffic is within a rack, a physical partition called a *cluster*, or between clusters. Figure 4 illustrates the hierarchies employed in Azure. Routers

Fig. 4. Hierarchies in Azure.

close to the host machines belong to one of the clusters. Traffic in a correctly configured data-center is routed without loops and along the shortest path for cluster-local traffic. Sample (slightly simplified from the ones checked for Azure) contracts are:

Cloud Contract 3 *Traffic from a host leaf directed to a different cluster from the leaf is forwarded to a router in a layer above. In other words, suppose that Router belongs to a cluster given as a predicate Cluster, and that RouterAbove is the set of routers above Router, then*

$$\neg Cluster(dst) \wedge Router(dst) \Rightarrow \bigvee_{n} RouterAbove(n)$$

On the other hand,

Cloud Contract 4 *Traffic from a host leaf directed to the same cluster is directed to the local VLAN or a router in the layer above that belongs to the same cluster as the host leaf router:*

$$Cluster(dst) \wedge Router(dst) \Rightarrow VLAN(dst) \ \vee \ \bigvee_{n} RouterAbove(n) \wedge Cluster(n)$$

The routing behavior of routers at the same level from the same cluster should also act uniformly for addresses within the cluster (they can behave differently for addresses outside of a cluster range).

Cloud Contract 5 *Let $Router_1, Router_2$ be two routers at the same layer within the cluster Cluster, then*

$$Cluster(dst) \Rightarrow Router_1(dst) \equiv Router_2(dst)$$

4 BGP: Border Gateway Protocol policies

The Azure network comprises several routing domains, and uses the BGP protocol to announce routing and reachability information between them. The combination of static policies configured in the device and the route information the device hears from its neighbors from the other routing domains determines (1) the forwarding rules enforced in the device, and (2) the BGP announcements that the device can make. These policies are critical to assure the availability and stability of the network. For example, policies are configured in the devices to avoid several hundreds of routes when they can be succinctly summarized as a single route. As another example, policies are configured to reject routes that are not reachable within the origin network. Such policies critical to enforce that nobody can impersonate an address. The intent of these policies can be captured as contracts, as we will illustrate with some examples.

Aggregate network statements (ANS) are used to specify a coarse aggregate of address ranges Given a device, aggregate network statements (ANS) are

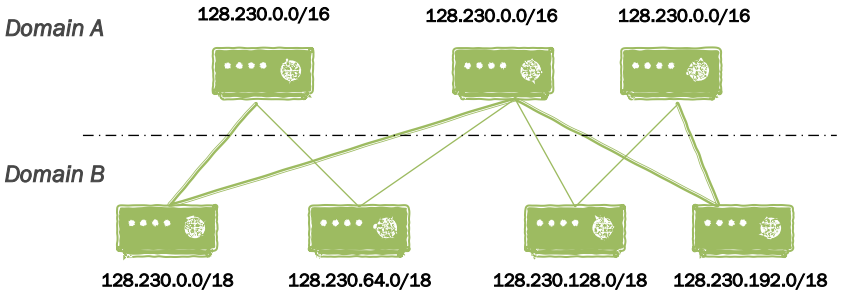

Fig. 5. BGP Aggregate Addresses.

used to specify a that are accessible from the device. Figure 5 provides an example scenario. In this figure, domain A comprises 3 routers, and domain B comprises 4 routers. Each router The addresses reachable from the routers in domain A are a union of all the addresses reachable in domain B. All the four address ranges reachable in domain B, namely 128.230.0.0/18, 128.230.64.0/18, 128.230.128.0/16, and 128.230.192.0/18, can be merged into a single address range, namely 128.230.0.0/16. Thus, we could configure an aggregate network statement in the routers in domain A to announce the combined aggregate 128.230.0.0/16 instead of announcing each of the four address ranges. Other routers receiving the announcements from the routers in domain A thus have to store only one route instead of four, thus saving memory consumption in the device. In real large IP networks such as Azure, the savings from these statements are in the order of several hundreds of rules. The contract for the ANS is that the set of configured tenant addresses that are handled by a given router

Given a device, aggregate network statements (ANS) are used to specify a coarse aggregate of address ranges that are accessible from the device. Figure 5 provides an example scenario. Domain A comprises 3 routers, and domain B comprises 4 routers. The addresses reachable from the routers in domain A are a union of all the addresses reachable in domain B. All the four address ranges reachable in domain B, namely 128.230.0.0/18, 128.230.64.0/18, 128.230.128.0/16, and 128.230.192.0/18, can be merged into a single address range, namely 128.230.0.0/16. Thus, we could configure an aggregate network statement in the routers in domain A to announce the combined aggregate 128.230.0.0/16 instead of announcing each of the four address ranges received from the routers in domain B. Routers receiving the announcements from the routers in domain A thus have to store only one route instead of four, thus saving memory consumption in the device. In large IP networks such as Azure, the savings from these statements are in the order of several hundreds of rules. The contract for the ANS is that the set of configured tenant addresses that are handled by a given router coincides with the configured ANS address ranges. In other words:

Cloud Contract 6 *The set of designated tenant addresses reachable from a router coincides with the address ranges summarized in the BGP aggregate network statements.*

Safety contracts for route announcements are enforced using a construct called route maps. Route maps specify policies for filtering or transforming route annoucements before either redistributing them or incorporating them as forwarding rules. For example, a safety contract for this configuration is that the devices in domain A reject any route announcements for an address range that is not contained in 128.230.0.0/16. In other words:

Cloud Contract 7 *The device rejects any BGP route announcement with an address range that is contained in the complement of tenant addresses reachable from the router.*

5 Z3, SMT, Model Checking and Network Verification

The Satisfiability Modulo Theories [2, 5] (SMT) solver Z3 [6], from Microsoft Research, is a core of several advanced program analysis, testing and model-based development tools. Figure 6 highlights the functionality and use of Z3. Z3 determines *satisfiability* of logical formulas. Furthermore, Z3 can provide witnesses for satisfiable formulas. This is useful for analysis tools that rely on components using logic for describing states and transformations between system states[4]. Consequently, they require a logic inference engine for reasoning about the state transformations. Z3 is particularly appealing because it combines specialized solvers for domains that are of relevance for computation and it integrates crucial innovations in automated deduction. It is tempting to build custom ad-hoc solvers for each application, but extending and scaling these require a high investment and will inevitably miss advances from automated deduction. New

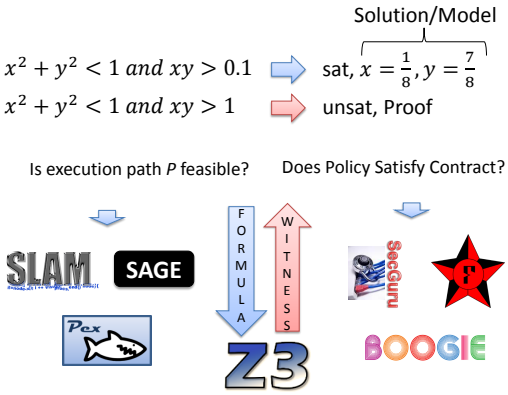

Fig. 6. Symbolic analysis with Z3

[4] Online demonstrations of the tools mentioned in the bottom of Figure 6 are available from http://rise4fun.com

applications introduce new challenges for Z3 and provide inspiration for improving automated deduction techniques. It is not uncommon that when improvements to Z3 are made based on one application, other applications benefit as well.

The source code, as well as nightly builds of Z3 is available online from http://z3.codeplex.com. There are several online resources around Z3. An interactive tutorial on using Z3 is available from http://rise4fun.com/z3/tutorial.

Z3 has recently been applied in a number of contexts related to *Network Verification*. These contexts require quite different capabilities. The use we described in this paper only relies on the quantifier-free theory of bit-vectors. On the other hand, the Network Optimized Datalog engine that we use in other work [11] requires optimized data-structures to maintain sets of reachable states. Checking firewall configurations is central to securing networks. Several other tools address checking firewall configurations. These include Margrave [14], which provides a convenient formalism for expressing rich properties of networks and firewalls (but counter-examples are only available for one address at a time), and the firewall testing tool in [4], which builds upon Isabelle/HOL and Z3 for generating test-cases. Z3 is also used in a very different twist for verifying compilers for software defined networks [15, 1].

More broadly, SAT, QBF (Quantified Boolean Formula) and other SMT solvers are actively purused for network data plane verification [18, 12]. It is beneficial to use specialized data-structures for analyzing IP networks, and this is explored in [10, 9, 17]. Model checkers are also currently being developed for verifying controller code for software defined networks [16, 13].

6 Conclusion

We developed *Cloud Contracts* to capture main architectural constraints in Microsoft Azure. The SecGuru tool is used to check these contracts on a continuous basis. By handling ACLs, routing tables and BGP policies, we covered the main sources of how IP networking is managed in Azure. It provides indispesible value for both more rapidly building out correctly configured data-centers, during the life-cycle of data-centers, as part of certifying isolation boundaries in data-centers and for analyzing live site incidents. SecGuru leverages the SMT solver Z3 for checking cloud contracts. Many software analysis, testing and verification tools already rely on Z3 and other SMT solvers to handle logical queries. The experience with SecGuru illustrates that the domain of engineering modern networks is a fresh new area where many problems can be reduced to logical queries and solved using advanced software engineering tools.

Acknowledgment We would like to express our gratitude to George Varghese and Charlie Kaufman for numerous interactions that have shaped this work. Our perspective on directions in current networking is influenced by conversations with Nick McKeown. Our experiences with network verification is based on joint work with several collaborators, including: Mooly Sagiv, Geoff Outhred, Nuno Lopes, Mingchen Zhao, Jeff Jensen, Monika Machado, Garvit Juniwal, Ratul

Mahajan, Ari Fogel, Jim Larus, Thomas Ball, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Michael Schapira and Asaf Valadarsky.

References

1. Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. VeriCon: towards verifying controller programs in software-defined networks. In Michael F. P. O'Boyle and Keshav Pingali, editors, *PLDI*, page 31. ACM, 2014.
2. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.
3. Nikolaj Bjørner and Karthick Jayaraman. Network Verification: Calculus and Solvers. In *SDN and FSI: The Next Generation Networking Infrastructure, Moscow*, 2014.
4. Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. hol-TestGen/fw - An Environment for Specification-Based Firewall Conformance Testing. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *ICTAC*, volume 8049 of *Lecture Notes in Computer Science*, pages 112–121. Springer, 2013.
5. Leonardo de Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction & Applications. *Comm. ACM*, 2011.
6. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
7. Albert G. Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. *Commun. ACM*, 54(3):95–104, 2011.
8. Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. Automated analysis and debugging of network connectivity policies. Technical Report MSR-TR-2014-102, Microsoft Research, July 2014.
9. Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: static checking for networks. In *NSDI*, 2012.
10. Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying Network-wide Invariants in Real Time. *SIGCOMM Comput. Commun. Rev.*, pages 467–472, September 2012.
11. Nuno Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Dna pairing: Using differential network analysis to find reachability bugs. Technical Report MSR-TR-2014-58, Microsoft Research, April 2014.
12. Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, New York, NY, USA, 2011. ACM.
13. Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. Kuai: A model checker for software-defined networks. In *FMCAD*, 2014.
14. Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave tool for firewall analysis. In *LISA*, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

15. Sudip Roy, Lucja Kot, Nate Foster, Johannes Gehrke, Hossein Hojjat, and Christoph Koch. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency. *CoRR*, abs/1403.2307, 2014.
16. Divjyot Sethi, Srinivas Narayana, and Sharad Malik. Abstractions for model checking SDN controllers. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 145–148. IEEE, 2013.
17. Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. In *2013 21st IEEE International Conference on Network Protocols, ICNP 2013, Göttingen, Germany, October 7-10, 2013*, pages 1–11. IEEE, 2013.
18. Shuyuan Zhang and Sharad Malik. SAT Based Verification of Network Data Planes. In Dang Van Hung and Mizuhito Ogawa, editors, *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 496–505. Springer, 2013.