Reachability Analysis for AWS-Based Networks

John Backes[1], Sam Bayless[1,4], Byron Cook[1,2], Catherine Dodge[1], Andrew Gacek[1(✉)], Alan J. Hu[4], Temesghen Kahsai[1], Bill Kocik[1], Evgenii Kotelnikov[1,3], Jure Kukovec[1,5], Sean McLaughlin[1], Jason Reed[6], Neha Rungta[1], John Sizemore[1], Mark Stalzer[1], Preethi Srinivasan[1], Pavle Subotić[1,2], Carsten Varming[1], and Blake Whaley[1]

[1] Amazon, Seattle, USA
gacek@amazon.com
[2] University College London, London, UK
[3] Chalmers University of Technology, Gothenburg, Sweden
[4] University British Columbia, Vancouver, Canada
[5] TU Wien, Vienna, Austria
[6] Semmle Inc, San Francisco, USA

Abstract. Cloud services provide the ability to provision virtual networked infrastructure on demand over the Internet. The rapid growth of these virtually provisioned cloud networks has increased the demand for automated reasoning tools capable of identifying misconfigurations or security vulnerabilities. This type of automation gives customers the assurance they need to deploy sensitive workloads. It can also reduce the cost and time-to-market for regulated customers looking to establish compliance certification for cloud-based applications. In this industrial case-study, we describe a new network reachability reasoning tool, called Tiros, that uses off-the-shelf automated theorem proving tools to fill this need. Tiros is the foundation of a recently introduced network security analysis feature in the *Amazon Inspector* service now available to millions of customers building applications in the cloud. Tiros is also used within Amazon Web Services (AWS) to automate the checking of compliance certification and adherence to security invariants for many AWS services that build on existing AWS networking features.

1 Introduction

Cloud computing provides on-demand access to IT resources such as compute, storage, and analytics via the Internet with pay-as-you-go pricing. Each of these IT resources are typically networked together by customers, using a growing number of virtual networking features. Amazon Web Services (AWS), for example, today provides over 30 virtualized networking primitives that allow customers to implement a wide variety of cloud-based applications.

Correctly configured networks are a key part of an organization's security posture. Clearly documented and, more importantly, verifiable network design

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11562, pp. 231–241, 2019.
https://doi.org/10.1007/978-3-030-25543-5_14

is important for compliance audits, *e.g.* the Payment Card Industry Data Security Standard (PCI DSS) [10]. As the scale and diversity of cloud-based services grows, each new offering used by an organization adds another dimension of possible interaction at the networking level. Thus, customers and auditors increasingly need tooling for the security of their networks that is accurate, automated and scalable, allowing them to automatically detect violations of their requirements.

In this industrial case-study, we describe a new tool, called Tiros, which uses off-the-shelf automated theorem proving tools to perform formal analysis of virtual networks constructed using AWS APIs. Tiros encodes the semantics of AWS networking concepts into logic and then uses a variety of reasoning engines to verify security-related properties. Tools that Tiros can use include Soufflé [17], MonoSAT [3], and Vampire [23]. Tiros performs its analysis statically: it sends no packets on the customer's network. This distinction is important. The size of many customer networks makes it intractable to find problems through traditional network probing or penetration testing. Tiros allows users to gain assurance about the security of their networks that would be impossible through testing.

Tiros is used directly today by AWS customers as part of the Amazon Inspector service [11], which currently checks six Tiros-based network reachability invariants on customer networks. The use of Tiros is especially popular amongst security-obsessed customers, *e.g.,* the world's largest hedge fund Bridgewater Associates, an AWS customer, recently discussed the importance of network verification techniques for their organization [6], including their usage of Tiros.

Related Work. Several previous tools using automated theorem proving have been developed in an effort to answer questions about software defined networks (SDNs) [1,2,5,12,13,16,19,25]. Similar to our approach, these tools reduce the problems to automated reasoning engines. In some cases, they employ over-approximative static analysis [18,19]. In other cases, they use general purpose reasoning engines such as Datalog [12,15], BDD [1], SMT [5,16], and SAT Solvers [2,25]. VeriCon [2], NICE [8], and VeriFlow [19] verify network invariants by analyzing software-defined-network (SDN) programs, with the former two applying formal software verification techniques, and the latter using static analysis to split routes into equivalence classes. SecGuru [5,16] uses an SMT solver to compare the routes admitted by access control lists (ACLs), routing tables, and border gateway protocol (BGP) policies, but does not support full-network reachability queries. In our approach we employ multiple encodings and reasoning engines. Our SMT encoding is similar in design to Anteater [25] and ConfigChecker [1]. Anteater performs SAT-based bounded model checking [4], while ConfigChecker uses BDD-based fixed-point model checking [7]. Previous work has applied Datalog to reachability analysis in either software or network contexts [12–14,24]. The approach used in Batfish [13,24] and SyNET [12] is similar to our Datalog approach; they allow users to express general queries about whole-network reachability properties using an expressive logic language.

Batfish presents results for small but complex routing scenarios, involving a few dozen routers. SyNET [12] also uses a similar Datalog representation of network reachability semantics, but rather than verifying network reachability properties, they provide techniques to synthesize networks from a specification. The focus in Tiros's encoding is expressiveness and completeness; it encodes the semantics of the entire AWS cloud network service stack. It scales well to networks consisting of hundreds of thousands of instances, routers, and firewall rules.

2 AWS Networking

AWS provides customers with virtualized implementations of practically all known traditional networking concepts, *e.g.* subnets, route tables, and NAT gateways. In order to facilitate on-demand scalability, many AWS network features focus on elasticity, *e.g.* Elastic Load Balancers (ELBs) support autoscaling groups, which customers configure to describe when/how to scale resource usage. Another important AWS networking concept is that of Virtual Private Cloud (VPC), in which customers can use AWS resources in an isolated virtual network that they control. Over 30 additional networking concepts are supported by AWS, including Elastic Network Interfaces (ENIs), internet gateways, transit gateways, direct connections, and peering connections.

Figure 1 is an example AWS-based network that consists of two subnets "Web" and "Database". The "Web" subnet contains two instances (sometimes called virtual machines) and the "Database" subnet contains one instance. Note that these machines are in fact virtualized in the AWS data center. The "Web" subnet's route table has a route to the internet gateway, whereas the "Database" subnet's route table only has local routes (within the VPC). In addition, each of the subnets has an ACL that contains security access rules. In particular, one of the rules forbids SSH access to the database servers.

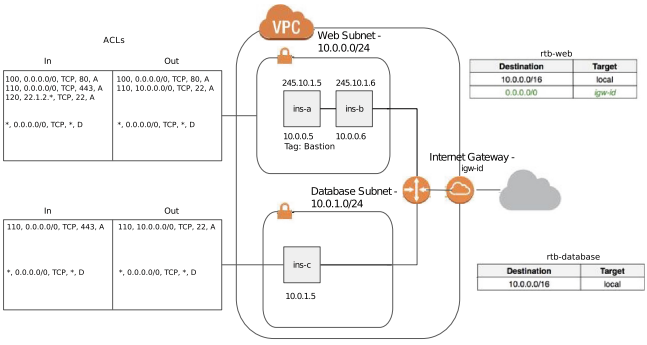


Fig. 1. An example VPC network

AWS-based networks frequently start small and grow over time, accumulating new instances and security and access rules. Customers or regulators want to

make sure that their VPC networks retain security invariants as their complexity grows. A customer may ask *network configuration questions* such as:

1. *"Are there any instances in subnet 'Web' that are tagged 'Bastion'?"* or *network reachability questions* such as:
2. *"Are there any instances that can be accessed from the public internet over SSH (TCP port 22)?"*

To answer such questions we must reason about which network components are accessible via feasible paths through the VPC, either from the internet, from other components in the VPC, or from other components in a different VPC via a peering connection or transit gateway.

3 AWS Networking Semantics as Logic

Tiros statically builds a model of an AWS network architecture to check reachability properties. The model of the network consists of two parts, the *formal specification* and the *snapshot* of the network. The specification formalizes the semantics of the AWS networking components, *e.g.*, how a route table directs traffic from a subnet, in which order a firewall applies rules in a security group, and how load balancers route traffic. The snapshot describes the topology and details of the network. For example, the snapshot contains the list of instances, subnets, and their route tables in a particular VPC (or set of VPCs). To answer reachability questions, Tiros combines the formal specification, the snapshot, and a query into a formula that represents the answer. Tiros uses up to three reasoning engines to answer queries: the Datalog solver Soufflé [17], the SMT solver MonoSAT [3], or the first-order theorem prover Vampire [23]. Due to the differing limitations and capabilities of each of these tools, we maintain three independent encodings of network semantics into logic, one for each of solver.

Datalog Encoding. In the Datalog encoding, a network model is a set of Datalog clauses (stratified, possibly recursive or negated Horn clauses without function symbols) using the theory of bit vectors to describe ports, IPv4 addresses, and subnet masks. The *specification* part of the network model contains types, predicates, constants, and rules that describe the semantics of the networking components in Amazon VPCs. The specification of Amazon VPC networks maps to approximately 50 types, 200 predicates, and over 240 rules. For example, a specification of the semantics of SSH tunneling is defined recursively: An instance can SSH tunnel to another instance iff it can either SSH to it directly, or through a chain of intermediate instances. We express this with predicates *canSshTunnel* and *canSsh*, of the type Instance × Instance, and rules:

$$canSshTunnel(I_1, I_2) \leftarrow canSsh(I_1, I_2).$$
$$canSshTunnel(I_1, I_2) \leftarrow canSshTunnel(I_1, I_3) \wedge canSshTunnel(I_3, I_2).$$

The *snapshot* part of the network model contains constants and *facts* (ground clauses with no antecedents) that describe the configuration of a specific AWS

network. Constants have the form type_{id}. For example, the snapshot of a network with an instance with id 1234 in a subnet with id *web* consists of the constants instance_{1234} and $\text{subnet}_{\text{web}}$, and the fact $hasSubnet(\text{instance}_{1234}, \text{subnet}_{\text{web}})$.

We illustrate the Datalog encoding using examples from Sect. 2. The network configuration question, $q(I)$, is encoded as $q(I) \leftarrow hasSubnet(I, \text{subnet}_{\text{web}}) \wedge hasTag(I, \text{tag}_{\text{bastion}})$. The network reachability question, $r(I, E)$, is encoded as:

$$r(I, E) \leftarrow hasEni(I, E) \wedge isPublicIP(Address) \wedge \\ reachPublicTcpUdp(\text{dir}_{\text{ingress}}, \text{proto}_6, E, \text{port}_{22}, Adress, \text{port}_{40000}).$$

In our Datalog encoding, we use the theory of bitvectors to reason about ports, IP addresses, and CIDRs. We use Souffl\'e as our Datalog solver, but in principle other Datalog solvers could also be used, so long as they also support bitvectors. We direct the reader to our co-author's dissertation (cf. Chapter 7 [28]) for a more detailed explanation of the Datalog encoding.

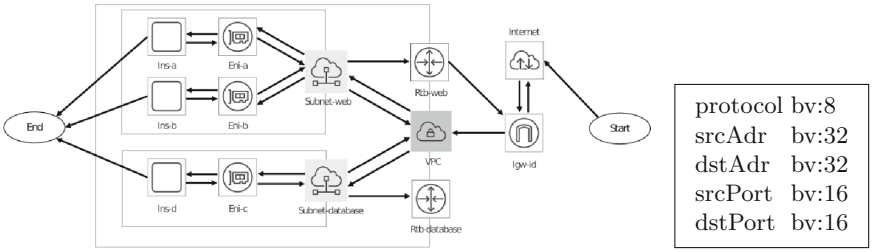


Fig. 2. (Left) The symbolic graph corresponding to the VPC in Fig. 1. (Right) A simplified symbolic packet, composed of bitvectors.

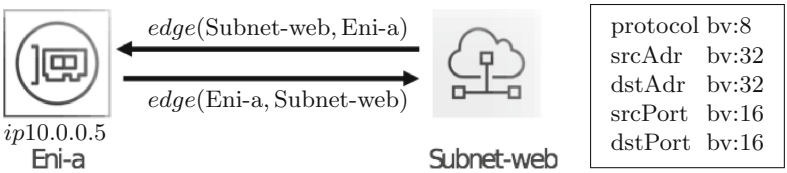
SMT Encoding. Our SMT encoding models network reachability as a *symbolic graph* of network components, along with one or more symbolic packet headers consisting of bitvectors for the source and destination addresses and ports. A symbolic graph consists of a set of nodes and directed edges, where the edges may be traversable or untraversable. Predicate $edge(u, v)$, where u and v are nodes, is true iff the corresponding edge is traversable. The assignment of the $edge(u, v)$ atoms in the formula determines which paths exist in the graph.

Figure 2 shows a symbolic graph corresponding to the VPC from Fig. 1. In our encoding, nodes represent networking components (such as instances, network interfaces, subnets, route tables, or gateways), and edges represent possible paths that packets may take between those components (such as between an instance and its network interface). Constraints between edge atoms and bitvectors in the packet headers define the routes that a packet can take.

For example, our encoding introduces an edge between each network interface node, Eni-a, and its containing Subnet-web node, $edge(\text{Eni-a}, \text{Subnet-web})$. As shown in Fig. 3, we also introduce constraints that force $edge(\text{Eni-a}, \text{Subnet-web})$

to be false if the packet's source address does not match the ENI's IP address. This ensures that packets leaving the ENI must have that ENI's IP address as their source address. Similar constraints ensure that packets entering the ENI must have that ENI's IP address as their destination address.

We encode reachability constraints into this graph using the SMT solver MonoSAT [3], which supports a theory of finite graph reachability. Specifically, we add a *start* and *end* node to the graph, with edges to the source components of the query and from the destination components of the query, and then we enforce a graph reachability constraint $reaches(start, end)$, which is true iff there is a start-end path under assignment to the edge literals. To encode the query "Are there any instances that can be accessed from the public internet over SSH?", we would add an edge from the *start* node to the internet, and from each EC2 instance to the *end* node. Additionally, we would add bitvector constraints forcing the protocol of the symbolic packet to be exactly 6 (TCP), and the destination port to be exactly 22.



$$((srcAdr \neq 10.0.0.5) \implies \neg edge(\text{Eni-a}, \text{Subnet-web}))$$
$$((dstAdr \neq 10.0.0.5) \implies \neg edge(\text{Subnet-web}, \text{Eni-a}))$$

Fig. 3. A small portion of the VPC graph, with constraints over the edges between an ENI and its subnet enforcing that packets entering or leaving the ENI have that ENI's source or destination address.

The SMT encoding described above is intended specifically for answering network reachability queries, and does not currently take into account other properties (such as tags) that would be required to model the more general network configuration queries supported by our datalog encoding.

First-Order Encoding. In our encoding for superposition solvers such as Vampire [23], we translate each network configuration question into a many-sorted first order logic problem that is unsatisfiable iff the answer to the question is true, and each network reachability question into a FOL problem that only has finite models, each corresponding to an answer to the question. For this encoding, we assume that network configuration questions have strictly yes/no answers, while network reachability questions return lists of solutions. In addition to its default saturation mode, Vampire implements a MACE-style [26] finite model builder for many-sorted first-order logic [27]. Thus we use Vampire both as a

saturation-based theorem prover and a finite model builder, running both modes in parallel and recording the result of the fastest successful run.

Our encoding begins with the same set of facts as were generated from the network model by our Datalog encoding, represented here by the symbols $(A_1, A_2, \ldots)$. From there, we handle network configuration and network reachability questions differently, with network-configuration encodings optimized for proof-by-contradiction, while reachability configurations are optimized for model-building. Proof-by-contradiction for yes/no questions is potentially faster than model-building, as intermediate variables need not be enumerated.

We encode a network configuration question φ in negated form: $A_1 \wedge \ldots \wedge A_n \Rightarrow \neg\varphi$. If Vampire can prove a contradiction in the negated formula, then φ holds. We encode a network reachability question φ into a formula of the form $A_1 \wedge \ldots \wedge A_n \wedge (\forall \bar{z})(q(\bar{z}) \Leftrightarrow \varphi) \Rightarrow (\forall \bar{z})q(\bar{z})$, where q is a fresh predicate symbol, and $\bar{z}$ are free variables of the network question φ. Each substitution of $\bar{z}$ that satisfies q corresponds to a distinct solution to the reachability question.

Our encoding targets Vampire's implementation of many-sorted first-order logic with equality, extended with the theory of linear integer arithmetic, the theory of arrays [22], and the theory of tuples [20]. We encode types, constants, and predicates using Clark completion [9]. We direct the reader to our co-author's dissertation (cf. Chapter 5 [21]) for a more detailed explanation of the Vampire encoding, including a detailed analysis of the performance trade-offs considered in this encoding.

4 Usage and Performance

In this section we describe the performance of the various solvers when used by Tiros in practice. Recall that our MonoSAT implementation can only answer reachability questions, whereas the other implementations also answer more general network configuration questions (such as the examples in Sect. 2).

In our experiments with Vampire, we found that the first order logic encoding we used does not scale well. As we were not able to obtain good performance from our Vampire-based implementation, in what follows we only present the experimental results for MonoSAT and Soufflé. We explain the poor performance of the Vampire encoding mainly by the fact that large finite domains, routinely used in network specifications, are represented as long clauses coming from the domain closure axioms. Saturation theorem provers, including Vampire, have a hard time dealing with such clauses.

Amazon Inspector. To compare the performance of Soufflé and MonoSAT in the context of the Tiros-based Amazon Inspector feature we randomly selected 10,000 network snapshots evaluated in December 2018. On these queries Soufflé required 4.1 s in the best-case, 45.1 s in the worst case, with 50th-percentile runtime of 5.1 s and 90th-percentile runtime of 5.5 s. MonoSAT required 0.8 s in the best case, 2.6 s in the worst case, with a 50th-percentile runtime of 1.39 s and 90th-percentile runtime of 1.79 s. To give the reader an

idea of the relative size of the constraint systems solved, in the smallest case our Soufflé encoding consisted of 2,856 facts, and the MonoSAT encoding consisted of 609 variables, 21 bitvectors, and 2,032 clauses. In the largest case, our Soufflé encoding consisted of 7517 facts, and the MonoSAT encoding consisted of 2,038 variables, 21 bitvectors, and 17,731 clauses.

Scalability Tests. MonoSAT and Soufflé scale to all queries evaluated using Amazon Inspector. To help understand the limits of the Soufflé and MonoSAT-based backends on larger networks, in Fig. 4 we compare the performance of the solvers on a series of artificially generated networks of increasing size, with 100, 1000, 10,000, and 100,000 instances. In each case, the query is *"list all open paths from the Internet to any instance in the VPC"*. We can see from the figure that neither approach dominates. In most cases the Datalog encoding is able to scale to 10,000 instances, but in no cases can it scale to 100,000 instances. In most cases the SMT encoding is able to scale to networks with 100,000 instances, but for the 'benchmark-2' networks, MonoSAT requires almost a full hour to solve the 10,000 instance network that Soufflé solves in 81 s. The SMT encoding performs poorly on 'benchmark-2' because that benchmark has a vast number of distinct feasible paths through the network, each requiring a separate SMT solver call. Other benchmarks have fewer distinct paths.

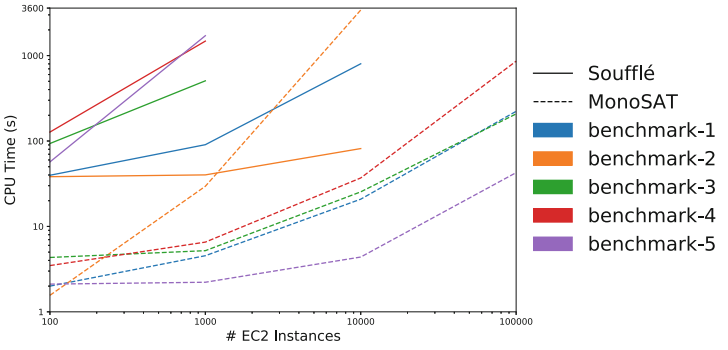


Fig. 4. Comparison of runtime in seconds for the different solver backends. Each benchmark uses a different color, *e.g.* Soufflé on benchmark-1 is a solid blue line, and MonoSAT on benchmark-1 is a dashed blue line. In these experiments, Soufflé recompiles each query before solving it, which adds ≈ 45 s to the runtime of each Soufflé query. In practice this cost can be amortized by caching compiled queries. (Color figure online)

Automating PCI Compliance Auditing. Many AWS services are built using other AWS services, *e.g.* AWS Lambda is built using AWS EC2 and the various AWS networking features. Thus within AWS we are using Tiros to prove the correctness of our own internal requirements. As an example, we use Tiros to

partially automate evidence generation for compliance audits of Payment Card Industry Data Security Standard (PCI DSS) [10]. Tiros is used across the many customer-facing AWS services that are built using AWS networking to establish controls supporting PCI DSS requirements 1.2, 1.3.1, 1.3.2, 1.3.4, and 1.3.7a.

Custom Application. AWS's Professional Services team works with some of the most security-obsessed customers to use advanced tools such as Tiros to achieve custom-tailored solutions. For example, as discussed in a public lecture [6], Bridgewater Associates worked with AWS Professional Services to build a Tiros-based solution which proves invariants of new AWS-based network designs before they are deployed in Bridgewater's AWS environment. Proof of these invariants assures the absence of possible data exfiltration paths that could be leveraged by an adversary.

5 Conclusion

We have described the first complete formalization of AWS networking semantics into logic. For customers of AWS services, Tiros provides deep insights into AWS networking. Via the incorporation of Tiros into the Amazon Inspector service, millions of AWS customers are able to automatically and continuously maintain their network-based security posture. They can now show compliance with security requirements at a scale that was impossible before. Internally within AWS, we are also able to automate some aspects of compliance evidence generation, which lowers our costs and increases our ability to quickly launch new features and services.

References

1. Al-Shaer, E., Marrero, W., El-Atawy, A., Elbadawi, K.: Network configuration in A box: towards end-to-end verification of network reachability and security. In: Proceedings of the 17th Annual IEEE International Conference on Network Protocols, 2009. ICNP 2009, Princeton, NJ, USA, 13–16 October 2009, pp. 123–132 (2009). https://doi.org/10.1109/ICNP.2009.5339690
2. Ball, T., et al.: VeriCon: towards verifying controller programs in software-defined networks. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, Edinburgh, UK, 9–11 June 2014, pp. 282–293 (2014). https://doi.org/10.1145/2594291.2594317, http://doi.acm.org/10.1145/2594291.2594317
3. Bayless, S., Bayless, N., Hoos, H.H., Hu, A.J.: SAT modulo monotonic theories. In: Proceedings of AAAI, pp. 3702–3709 (2015)
4. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
5. Bjørner, N., Jayaraman, K.: Checking cloud contracts in Microsoft azure. In: Natarajan, R., Barua, G., Patra, M.R. (eds.) ICDCIT 2015. LNCS, vol. 8956, pp. 21–32. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14977-6_2

6. Bridgewater Associates: Bridgewater's model-based verification of AWS security controls. AWS New York Summit (2018). https://www.youtube.com/watch?v=gJhV35-QBE8
7. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 1020 states and beyond. Inf. Comput. **98**(2), 142–170 (1992)
8. Canini, M., Venzano, D., Peresini, P., Kostic, D., Rexford, J.: A nice way to test openflow applications. In: Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI). No. EPFL-CONF-170618 (2012)
9. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Springer, Boston (1977). https://doi.org/10.1007/978-1-4684-3384-5_11
10. CSS Council. Payment Card Industry (PCI) Data Security Standard Requirements and Security Assessment Procedures Version 3.2.1. PCI Security Standards Council (2018)
11. Dodge, C., Quigg, S.: A simpler way to assess the network exposure of EC2 instances: AWS releases new network reachability assessments in amazon inspector. AWS Security Blog (2018). https://aws.amazon.com/blogs/security/amazon-inspector-assess-network-exposure-ec2-instances-aws-network-reachability-assessments/
12. El-Hassany, A., Tsankov, P., Vanbever, L., Vechev, M.: Network-wide configuration synthesis. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 261–281. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_14
13. Fogel, A., et al.: A general approach to network configuration analysis. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, NSDI 2015, pp. 469–483. USENIX Association, Berkeley (2015). http://dl.acm.org/citation.cfm?id=2789770.2789803
14. Hajiyev, E., Verbaere, M., de Moor, O.: *codeQuest*: scalable source code queries with datalog. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 2–27. Springer, Heidelberg (2006). https://doi.org/10.1007/11785477_2
15. Hoder, K., Bjørner, N., de Moura, L.: μZ– an efficient engine for fixed points with constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 457–462. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_36. http://dl.acm.org/citation.cfm?id=2032305.2032341
16. Jayaraman, K., Bjørner, N., Outhred, G., Kaufman, C.: Automated analysis and debugging of network connectivity policies. In: Microsoft Research, pp. 1–11 (2014)
17. Jordan, H., Scholz, B., Subotić, P.: Soufflé: on synthesis of program analyzers. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 422–430. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_23
18. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: static checking for networks. In: NSDI, vol. 12, pp. 113–126 (2012)
19. Khurshid, A., Zou, X., Zhou, W., Caesar, M., Godfrey, P.B.: Veriflow: verifying network-wide invariants in real time. In: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, 2–5 April 2013, pp. 15–27 (2013). https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/khurshid
20. Kotelnikov, E., Kovács, L., Voronkov, A.: A FOOLish encoding of the next state relations of imperative programs. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 405–421. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_27

21. Kotelnikov, E.: Checking network reachability properties by automated reasoning in first-order logic. In: Kotelnikov, E. (ed.) Automated Theorem Proving with Extensions of First-Order Logic, chap. 5, pp. 114–131. Chalmers University of Technology, Gothenburg (2018). https://research.chalmers.se/publication/504640/file/504640_Fulltext.pdf
22. Kotelnikov, E., Kovács, L., Reger, G., Voronkov, A.: The vampire and the FOOL. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs 2016, pp. 37–48 (2016). https://doi.org/10.1145/2854065.2854071, http://doi.acm.org/10.1145/2854065.2854071
23. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
24. Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K., Varghese, G.: Checking beliefs in dynamic networks. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, NSDI 2015, pp. 499–512. USENIX Association, Berkeley (2015). http://dl.acm.org/citation.cfm?id=2789770.2789805
25. Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, B., King, S.T.: Debugging the data plane with anteater. In: Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, 15–19 August 2011, pp. 290–301 (2011). https://doi.org/10.1145/2018436.2018470, http://doi.acm.org/10.1145/2018436.2018470
26. McCune, W.: A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, Argonne National Laboratory (1994)
27. Reger, G., Suda, M., Voronkov, A.: Finding finite models in multi-sorted first-order logic. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 323–341. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40970-2_20
28. Subotić, P.: Logic defined static analysis. Ph.D. thesis, University College London (2018)