Predicting Network Futures with Plankton

Santhosh Prabhu
University of Illinois at Urbana-Champaign
prabhum2@illinois.edu

Ali Kheradmand
University of Illinois at Urbana-Champaign
kheradm2@illinois.edu

Brighten Godfrey
University of Illinois at Urbana-Champaign
pbg@illinois.edu

Matthew Caesar
University of Illinois at Urbana-Champaign
caesar@illinois.edu

ABSTRACT

Recent years have seen significant advancement in the field of formal network verification. Tools have been proposed for offline data plane verification, real-time data plane verification and configuration verification under arbitrary, but static sets of failures. However, due to the fundamental limitation of not treating the network as an evolving system, current verification platforms have significant constraints in terms of scope. In real-world networks, correctness policies may be violated only through a particular combination of environment events and protocol actions, possibly in a non-deterministic sequence. Moreover, correctness specifications themselves may often correlate multiple data plane states, particularly when dynamic data plane elements are present. Tools in existence today are not capable of reasoning about all the possible network events, and all the subsequent execution paths that are enabled by those events. We propose Plankton, a verification platform for identifying undesirable evolutions of networks. By combining symbolic modeling of data plane and control plane with explicit state exploration, Plankton performs a goal-directed search on a finite-state transition system that captures the behavior of the network as well as the various events that can influence it. In this way, Plankton can automatically find policy violations that can occur due to a sequence of network events, starting from the current state. Initial experiments have successfully predicted scenarios like BGP Wedgies.

CCS CONCEPTS

• **Networks** → *Network management*; *Network monitoring*;

APNet'17, Hong Kong, China

DOI: 10.1145/3106989.3106991

KEYWORDS

Network Troubleshooting, Correctness

ACM Reference format:
Santhosh Prabhu, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2017. Predicting Network Futures with Plankton. In *Proceedings of APNet'17, Hong Kong, China, August 03-04, 2017,* 7 pages.
DOI: 10.1145/3106989.3106991

1 INTRODUCTION

Ensuring correctness of networks is a difficult task, but given the critical nature of today's networks, an important one too. The growing number of network verification tools are targeted towards automating this process as much as possible, thereby reducing the burden on the network operator. Verification platforms have improved steadily in the recent years, both in terms of scope and scale. Starting from offline data plane verification tools like Anteater [18] and HSA [11], the state of the art has evolved to support real-time data plane verification [10, 12], and more recently, analysis of the configuration [3–5]. However, all existing verification techniques have a fundamental limitation — they do not treat the network as the evolving system that it is.

This puts significant constraints on our ability to verify networks. Correctness of the network may often not be about a single data plane state, but a *temporal property* describing the possible evolution of that state. For example, the network operator may wish to verify the policy: *Each data flow may pass through any intrusion detection system, but **always** the same one so connections are tracked*. Furthermore, even simpler reachability policies may often be satisfied by the current data plane state, but violated when the state changes due to events such as link failures, reconnections or arrival of packets. To make things worse, there is significant non-determinism in how the network state evolves, both within protocols as well as outside. This creates problems that are particularly hard to catch. BGP wedgies [6], where the converged network state depends on non-deterministic events, are perhaps the best known example, but there can also be other connectivity problems caused by race conditions in interaction between different protocols or multiple instances of the same protocol [2, 13, 21].

Network verification techniques that exist today are not capable of checking such policies. Data plane analysis tools [10–12, 18] can accurately analyze the present data plane state, but cannot reason about future states that may potentially violate the policy. There also exist platforms such as Batfish [4] and ERA [3], which are designed to do "what-if" testing. With these tools, the operator can try out various failure scenarios (e.g. link failures) to see if the correctness policy is compromised in any of them. ARC [5] is a tool designed to function without operator involvement, but verifies shortest-path routing protocols under potential link failures, and hence is not capable of modeling tricky BGP configurations. Neither ARC nor any of the other techniques in existence are capable of reasoning about arbitrary environment events, and the many possible non-deterministic evolution paths that those events can trigger. They may declare the network to be safe, while in reality, a non-deterministic path of protocol execution may cause correctness goals to be violated. Ideally, a network verification tool should examine all possible network evolutions (subject to constraints of reason), and report violations that occur in any possible future state.

In this paper, we propose Plankton, the first network verification platform that can reason about non-deterministic evolutions of the network, in response to possible external events. In addition to doing so, Plankton is capable of verifying not only single-snapshot policies, but also temporal properties including protocol convergence. Plankton performs this analysis by using an exhaustive state space search on a symbolic model that includes the data plane, control plane, and the environment. By combining ideas of data plane/control plane equivalences with scalable state exploration techniques such as Partial Order Reduction, Plankton can detect possible policy violations that have thus far been undetectable. Microbenchmarks of a prototype of Plankton show that it scales well to real-world networks.

2 MOTIVATION

The network verification platforms of today can be categorized into a few major categories. We briefly discuss each of these next.

Data plane verification: These tools verify the current incarnation of the data plane for reachability violations. Anteater[18] and HSA [11] perform this analysis offline, whereas VeriFlow [12] and NetPlumber [10] are capable of real-time verification, and thus, in SDNs, prevent incorrect updates from ever reaching the data plane. Despite their usefulness, these tools are limited in the sense that they cannot reason about violations before they actually happen. In other words, policy violations need to actually exist in the network, for them to be detected. This severely reduces the time available to the administrators to fix the problem.

Configuration verification under particular topology scenarios: These tools directly analyze the configuration rather than a particular data plane incarnation. Hence, they are capable of predicting correctness violations even before they occur in the data plane. Batfish [4] performs simulation of the control plane to first generate a data plane, whereas ERA [3] performs symbolic exploration of the control plane, by computing equivalence classes of routing updates. BagPipe [22] uses a theorem prover to check for policy violations in BGP configurations. While these tools are an improvement over data plane analysis platforms, they are not capable of automatically detecting correctness issues which manifest as a result of external influences, such as link failures. Detecting such issues using these tools requires the operator to iterate the various environmental conditions, and the number of iterations required can escalate very quickly.

Configuration verification under topology changes: This class of techniques are capable of verifying policies under potential failures that cause changes to the topology. The only known tool in this category, ARC, uses a model that is not expressive enough to capture tricky behaviors in protocols such as BGP. More generally, this class represents approaches that only look at the final topology, and misses out on any violations caused by the manner in which the protocol executes. A good example that illustrates the limitations of these tools are BGP Wedgies, which cannot be detected automatically by these tools, since the topology is the same in both the ideal as well as the non-ideal state of the network.

All the methods described above are designed only to check reachability policies in a single data plane incarnation. They are not capable of checking conditions that require correlation of states, or about the convergence of the protocol. This kind of ability is important in many practical settings, such as verifying consistency of forwarding across the evolution of the data plane state, verifying policies in presence of dynamic components, verifying correctness in transient states etc. Plankton stands out as being capable of supporting these goals. In summary, Plankton is the first formal verification technique that can:

- detect policy violations due to various sources of non-determinism.
- support liveness checks, including protocol convergence.
- verify temporal properties over data plane states, when the data plane evolves due to stateful elements.
- check policies over transient states.

Greater expressibility in Plankton is enabled by its unique combination of symbolic modeling with scalable explicit state exploration. This combination is realized by first defining

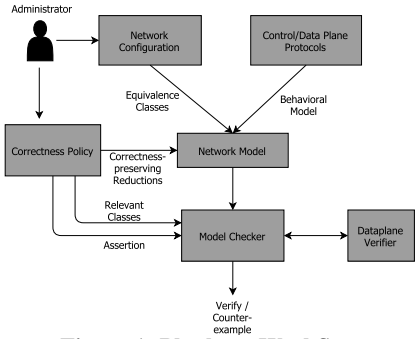

Figure 1: Plankton Workflow

equivalences in control/data planes, and then defining network models in terms of these equivalence classes in an explicit state model checker. The network models capture the steps in protocol execution as well as other events such as failures, reconnections, packet send etc. The model checker provides fast exploration of this state space, with efficient state keeping and other scalability techniques such as Partial Order Reduction. We describe the overall design of Plankton next.

3 PLANKTON WORKFLOW

Fig. 1 illustrates the verification workflow using Plankton.

The key component of Plankton that drives its temporal reasoning ability is an **explicit-state model checker**. These kind of model checkers verify transition systems by exhaustively searching through each state of the system. Their advantage over the other class of model checkers (known as symbolic model checkers) is that they do not require a precomputed transition relation for the states of the system. So, they are capable of executing even real code as part of the model checking process. To achieve scalability, they rely on efficient hashing-based state management[9] and state space reduction techniques such as *Partial Order Reduction*.

More than the known benefits of explicit-state model checkers over symbolic ones, for Plankton, we are motivated by the fact that we can look at individual data plane states produced in the various execution paths of the network, and verify them individually using a data plane verification tool such as VeriFlow [12]. By defining predicates over individual data plane states, we allow the model checker to verify properties that correlate multiple states. These properties are expressed in *Linear Temporal Logic* [19], a logic system designed to express logical formulae interpreted over execution paths.

The model checker also allows us to verify liveness properties (properties of the form *something good eventually happens*), which may be violated by loops in the state transition graph. These kind of policies cover interesting correctness requirements, such as *The protocol eventually converges*.

Although the model checker does an enumerative exploration of the network model, Plankton protects itself from an impossibly large scalability challenge by defining the model itself in terms of equivalence classes. Past work has defined equivalence classes over varying scope - the data plane only[12], control plane message paths[3] etc. In Plankton, equivalence classes are defined over control plane messages such that any two messages that belong to the same equivalence class will be handled identically by all devices in the network, even after applying any hypothetical changes that are explored by the model checker. These equivalence classes are computed by examining the protocol configuration at each device in the network, and computing the coarsest partition of the packet space such that each class has a distinct configuration for each protocol throughout the network. While this may appear to be a large number at first glance, Plankton does not need to proactively reason about the fate of each of these equivalence classes. When the policy to be verified is known, the actual reasoning and analysis can be done just on the equivalence classes that are needed for doing the verification correctly.

The protocol models that we use in Plankton are defined based on standardized interpretations of the protocols. This is a limitation of Plankton, as it prevents implementation-specific problems from being detected. In theory, it is possible for Plankton to use vendor-specific models for each device, but we believe our current approach is more pragmatic. Indeed, past work on configuration verification has also relied on similar models for protocols [3–5]. When exploring these models, the model checker interprets the processing of one update in any protocol by a device in the network as a state transition. Depending on the update, this may or may not result in changes to the data plane. This level of granularity allows us to detect a relatively large set of issues while maintaining good scalability.

The component of Plankton that is responsible for actually detecting violations is the data plane verifier. The verifier is invoked by the model checker every time a change is made to the data plane state associated with an equivalence class. The verifier is essentially responsible for evaluating predicates over a given data plane state. This functionality serves two purposes - first, a predicate can define data plane properties that are expected to hold on all the data plane states that are generated by the model checker. Second, these predicates can be combined with LTL operators to express interesting temporal properties about the data plane evolution. Thanks to recent advances in real-time verification of data plane states [10, 12], we can afford to use data plane verification as an oracle for end-to-end network verification, and still verify interesting properties on acceptable timescales.

4 PROOF OF CONCEPT

We implemented proof-of-concept mappings for OSPF and BGP in Promela, a modeling language that is interpreted by a well-known explicit model checker known as SPIN[8]. These

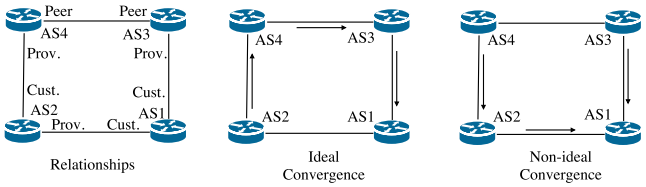

Figure 2: BGP Wedgie

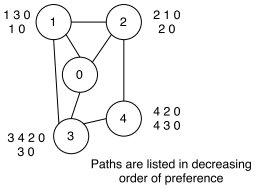

Figure 3: A BGP configuration that doesn't converge (Reproduced from [7])

models are currently designed to explore one equivalence class at a time. This allows us to capture a large fraction of interesting properties, while significantly simplifying the model. However, we can choose to model multiple classes, if a motivating case is found. We used this implementation to test for known violations in networks. We modeled the network illustrated in Figure 2, including the AS relationships and primary/backup preferences. We checked for the following condition: *can AS1 be the next-hop for AS2, while the link between AS3 and AS1 is up?* In our first experiment, we started from an unconverged network as the initial state. Plankton's search algorithm found a violation, and reported an execution path where AS4 picks AS2 as its successor even before it receives the advertisement from AS3. This possibility is in fact known, and to avoid it, network administrators often bring up backups only after the primary path is established. In our second experiment, we started from an "ideally converged" initial state, with AS2 having picked AS4 as its successor. In this experiment, Plankton finds the execution where the primary link fails and then reconnects, causing the network to converge to non-ideal state. These results match the expectation, and illustrates the effectiveness of Plankton's exhaustive search. We also checked for convergence of BGP in networks such as the one illustrated in Figure 3, where violations were correctly identified by Plankton.

To microbenchmark the scalability of the approach, we ran the following experiment: On datacenter topologies of various sizes, we configure BGP as described in RFC 7983[15]. We simulate a case where BGP attributes are configured to allow multipath, but due to a misconfiguration, only one path is chosen between any source-destination pair. With such a misconfiguration, the paths selected can often depend on the order in which updates are received at various nodes [14]. On such a network, we check policies which state that the path between two edge switches should pass through one among a particular set of aggregation switches. Plankton evaluates various nondeterministic convergence paths in the network,

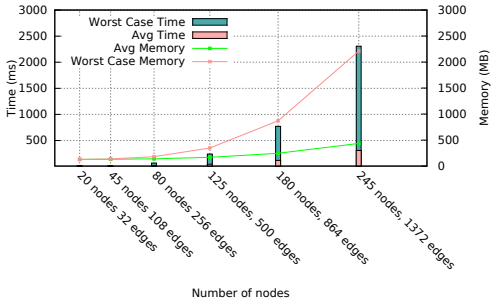

Figure 4: Time and memory taken by Plankton to find an execution in BGP datacenters

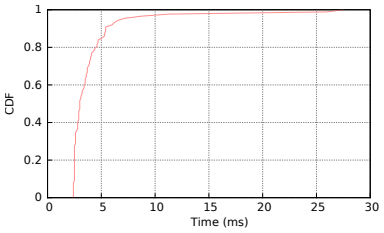

Figure 5: CDF of time taken to check for wedgies for specific origins in the CAIDA AS topology

and determines a violating sequence of protocol steps. Figure 4 illustrates the average and worst-case time taken by Plankton to find the violations. These numbers were obtained on a 4 GHz Intel Core i7 CPU, running single threaded. The numbers are promising — the checks ran in acceptable time for real-world mid-scale enterprise datacenter networks.

We also ran an experiment using the CAIDA AS topology [17], by randomly picking ASes with two providers and marking one as primary and the other as backup. Then, we check this configuration for potential Wedgies. Figure 5 illustrates the CDF of time taken to finish verification (with either a positive or negative result). As can be seen, most checks finish within milliseconds. We note here that this is possible after many rounds of optimizing our BGP model, without which the check never terminates within a timeout of 5 minutes. We describe these optimizations in the next section.

5 DISCUSSION

In this section, we discuss some of the design choices and optimizations in Plankton. Some are implemented in our prototype, but may be extended; some are entirely new, to be tackled in future work.

5.1 Why Model Checkers?

The first question one may ask is how a model checker is better than naive exhaustive exploration using simulation. This is an important question particularly because we use an explicit state model checker rather than a symbolic one, which means that it generates each individual state of the system

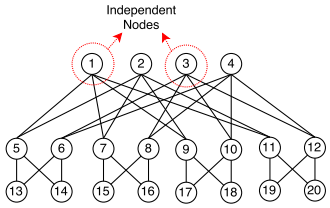

Figure 6: Partial Order Reduction in Plankton

separately, and performs the verification. The benefits that we obtain from using the model checker are primarily as follows:

- **Efficient branching:** In cases where the protocol execution runs into a large number of steps, having a state-machine representation with the ability to traverse states both forward as well as backward avoids the need to repeat steps over and over.
- **Ability to verify temporal properties:** The model checker is equipped with techniques for on-the-fly model checking, which enables Plankton to verify temporal properties specified in LTL, and also detect issues such as non-convergence.
- **Efficient state keeping:** The model checker employs techniques such as bitstate hashing to efficiently carry out the bookkeeping associated with the exploration. (In principle this can be implemented in any simulation-based exploration). We discuss bitstate hashing in detail later.

5.2 Optimizations in Plankton

Partial Order Reduction: Partial Order Reduction (POR) is an optimization used by model checkers when exploring large state spaces. It leverages the fact that certain actions can be applied in any order, without altering the outcome. So, the model checker needs to verify only one sequence of actions, rather than all possible sequences. This optimization is particularly useful in analysis of asynchronous systems, and hence should make a significant difference in analyzing network protocols, which have significant asynchronous behavior (For example, in a network running OSPF, only a single order of message delivery needs to be explored, if transient states are not relevant to the correctness property).

To ensure that we achieve good POR, Plankton performs a reduction separate from the automatic reduction done by the model checker. This reduction may eliminate certain transient states from being checked, while not compromising the correctness of verification. If the policy being verified concerns only the converged states of the network, Plankton tries to eliminate as many transient states as possible, while retaining all possible converged states. If, on the other hand, transient states are indeed relevant to the check, and there exist one or more transient states where the policy could be violated, Plankton ensures that at least one such state will be checked. This is done by defining the reduction strictly based on the policy being verified. Once the type of policy and the exact parameters for it are known, the POR mechanism generates a conservative dependency model that captures which protocol steps can be executed independently of which others without affecting the property. The network program that is passed to the model checker allows non-deterministic choice of actions only in cases where the order of the actions can potentially affect the satisfaction or violation of the property. Thus, only a reduced extent of non-determinism needs to be exhaustively explored by the model checker before concluding whether the policy is satisfied or not. For example, consider the fat-tree network illustrated in Figure 6. Assuming that the network runs BGP as described in RFC 7983, when verifying a path sequencing policy in converged states, the dependency model determines that core nodes 1 and 3 are independent, so, in states where both nodes are enabled only one order of execution is explored. Our ongoing work aims to create a more general POR mechanism with reduced dependency on the policy being verified.

Parameterized Modeling: Our current experiments have used models that are parameterized by equivalence class. In other words, we model only one equivalence class at any given time. As we discussed in §3, these equivalence classes are defined such that they do not need to be split further even when the network reconverges in response to events. So, verifying one equivalence class in isolation is feasible. However, there do exist correctness policies that cannot be evaluated by looking at a single equivalence class alone. For example, the policy that two different packets should always have the same fate starting from a particular device may not be verifiable with this model, if the packets happen to be in different equivalence classes (if not, the property is vacuously satisfied).It is indeed possible to use Plankton's approach to look at more than one equivalence class simultaneously. We do not do that yet, because we believe the single-EC approach is already capable of capturing a rich set of policies.

Cone of Influence Reduction: Often, only a small fraction of the overall set of applicable events would be relevant to the verification. For instance, when verifying a policy about a particular OSPF path under failures, we can restrict the set of potential failures to the links that are on the shortest path. Cone of influence reduction limits the set of applicable events to only those that can make a difference to whether the correctness requirement is satisfied or not. This reduction can be made at varying granularity, which determines its effectiveness. In our current experiments, the reduction is done very conservatively while a protocol is being executed. So, we cannot eliminate many steps that are relevant in some phase of protocol execution, but not in others. By increasing granularity to a level where every step is executed only if it can affect the end-result, we can avoid exploring unnecessary executions altogether. We are working on heuristics to

Experiment	Without bitstate hashing	With bitstate hashing
125 Node DC (Worst Case)	347.5 MB	35.4 MB
180 Node DC (Worst Case)	870.3 MB	69 MB
245 Node DC (Worst Case)	2211.2 MB	121.1 MB
CAIDA Wedgie (Avg Case)	135.6 MB	23.6 MB

Figure 7: The effect of bitstate hashing on memory use

identify the relevant steps accurately.

Bitstate Hashing: Bitstate hashing is an optimization that is provided to Plankton by the model checker. It refers to the use of a bloom filter to keep track of explored states, rather than storing them explicitly. Naturally, this can cause some false negatives, but model checkers such as SPIN provide probabilistic guarantees of correctness when using bitstate hashing. Using this technique can provide significant reduction in memory overhead, as illustrated in Figure 7. Coverage is measured using a *hash factor*, which is recommended to be greater than 100. The observed hash factor in our experiments was greater than 1 million, which makes bitstate hashing a viable option. Nevertheless, we have not had to use bitstate hashing thus far, having seen good memory scaling.

5.3 Possible Future Optimizations

Iterative Deepening: Even with optimizations like partial order reduction, there can still be a large number of non-deterministic paths that are possible, only few of which lead to an actual violation. In some cases, picking the right non-deterministic path to explore can make the difference between a quick result and pointless exploration. With Iterative Deepening, all paths are explored up to a certain depth before starting again and exploring up to a greater depth. Naturally, this approach has benefits only when there is a real violation to be found, and the path from the initial state to the violation is relatively short. When the policy is actually satisfied, this method of search is more expensive than Depth First Search. However, since any possibility of a quick violation is eliminated first, a long-running IDS indicates that there is no *reasonable* violation to be found, which constitutes a best-effort result, that may often be acceptable.

Incremental Modeling: Plankton is required to repeatedly compute data plane state under various environmental scenarios. Often, only the result, which is a deterministic function of the current environment input, is necessary. For example, when verifying properties over a deterministic shortest-path routing protocol, if the transient states are not relevant to the property, Plankton can choose the most efficient way to generate the data plane state. Given the manner in which network protocols operate, this would mean retaining the previously generated data plane state and then simulating events that would turn the previous environment into the new environment. In the shortest-path routing example, if verifying under single link failure, Plankton only needs to reconnect the previously failed link and then disconnect the current failed link. We expect that depending on the protocol and the correctness invariant being verified, such incremental computation will provide significant improvements to scalability.

Heuristic Search: The optimizations discussed so far are designed to prune the large state space to practical proportions. There can be significant benefit in also directing the model checker's search process, so that it first explores paths that are more likely to produce a violation to the policy. There already exist model checkers with heuristic support, so the biggest challenge will be in defining effective heuristics for the policies that Plankton verifies.

Symbolic Model Checking: The current architecture of Plankton already has some elements of symbolic analysis, since models are created with respect to equivalence classes. To push this further, the explicit state model checker can be replaced by a symbolic model checker, which would explore multiple non-deterministic execution paths simultaneously. Such symbolic exploration can bring benefit in networks like datacenters where significant symmetry exists. The challenge in using symbolic model checker is that we rely on a data plane verifier to do the actual data plane check, which only looks at a single data plane state. The first step towards making the transition will be to equip data plane verification techniques to handle multiple data plane states simultaneously. Switching to symbolic model checkers would also make it difficult to build models for complicated protocols, since the entire transition system needs to be appropriately encoded before starting the verification process.

6 RELATED WORK

Similarities and differences of Plankton to existing network verification techniques have been discussed in previous sections. We have shown that Plankton is the first system capable of analyzing network evolution over time, and in presence of non-determinism. Other efforts for formal methods in networking, such as correct-by-construction network management techniques[1, 16, 20], share Plankton's goal of eliminating errors in networks, but they follow a more radical approach that requires re-architecting the network. The use of model checking as a tool for verification is well studied, but its use in combination with network equivalence classes to create a compact network model, and the use of this model to verify network properties is a new research direction.

7 CONCLUSION

We described Plankton, a formal verification tool for networks that performs explicit-state verification of a network model defined over equivalence classes. Plankton is capable of automatically detecting violations triggered along only particular non-deterministic paths, including for properties that are themselves temporal in nature. These abilities of Plankton represent a significant improvement in the scope of network verification itself, bringing us one step closer to error-free network design.

REFERENCES

[1] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 328–341. https://doi.org/10.1145/2934872.2934909

[2] Rodney Dunn. 2014. Preferring MPLS VPN BGP Path with IGP Backup. https://supportforums.cisco.com/document/29361/preferring-mpls-vpn-bgp-path-igp-backup. (2014). Accessed: 2017-06-20.

[3] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA, 217–232. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/fayaz

[4] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 469–483. http://dl.acm.org/citation.cfm?id=2789770.2789803

[5] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. 14. https://doi.org/10.1145/2934872.2934876

[6] T. Griffin and G. Huston. 2005. BGP Wedgies. RFC 4264 (Informational). (Nov. 2005). http://www.ietf.org/rfc/rfc4264.txt

[7] Timothy G. Griffin, F. Bruce Shepherd, and Gordon Wilfong. 2002. The Stable Paths Problem and Interdomain Routing. *IEEE/ACM Trans. Netw.* 10, 2 (April 2002), 232–243. http://dl.acm.org/citation.cfm?id=508325.508332

[8] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Softw. Eng.* 23, 5 (May 1997), 279–295. https://doi.org/10.1109/32.588521

[9] Gerard J. Holzmann. 1998. An Analysis of Bitstate Hashing. *Form. Methods Syst. Des.* 13, 3 (Nov. 1998), 289–307. https://doi.org/10.1023/A:1008696026254

[10] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian

[11] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 113–126. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/kazemian

[12] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-wide Invariants in Real Time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, Berkeley, CA, USA, 15–28. http://dl.acm.org/citation.cfm?id=2482626.2482630

[13] Petr Lapukhov. 2010. Understanding EIGRP SoO and BGP Cost Community. http://blog.ine.com/wp-content/uploads/2010/04/understanding-eigrp-soo-bgp-cost-community.pdf. (2010). Accessed: 2017-06-20.

[14] Petr Lapukhov. 2016. *Equal-Cost Multipath Considerations for BGP*. Internet-Draft draft-lapukhov-bgp-ecmp-considerations-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-lapukhov-bgp-ecmp-considerations-00 Work in Progress.

[15] P. Lapukhov, A. Premji, and J. Mitchell. 2016. Use of BGP for Routing in Large-Scale Data Centers. RFC 7938 (Informational). (Aug. 2016), 35 pages. https://doi.org/10.17487/RFC7938

[16] Franck Le, Geoffrey G. Xie, and Hui Zhang. [n. d.]. Theory and New Primitives for Safely Connecting Routing Protocol Instances. In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. 219–230. https://doi.org/10.1145/1851182.1851210

[17] Matthew Luckie, Bradley Huffaker, Amogh Dhamdhere, Vasileios Giotsas, and kc claffy. 2013. AS Relationships, Customer Cones, and Validation. In *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC '13)*. 14. https://doi.org/10.1145/2504730.2504735

[18] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference (SIGCOMM '11)*. New York, NY, USA, 290–301. https://doi.org/10.1145/2018436.2018470

[19] Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77)*. 12. https://doi.org/10.1109/SFCS.1977.32

[20] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. 2017. Correct by Construction Networks Using Stepwise Refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 683–698. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ryzhyk

[21] Cisco Support. 2016. OSPF Redistribution Among Different OSPF Processes. http://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/4170-ospfprocesses.html. (2016). Accessed: 2017-06-20.

[22] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, 16. https://doi.org/10.1145/2983990.2984012