High-coverage testing of softwarized networks

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

Gohar Irfan Chaudhry
University of Illinois at Urbana-Champaign
gic2@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

Network operators face a challenge of ensuring correctness as networks grow more complex, in terms of scale and increasingly in terms of diversity of software components. Network-wide verification approaches can spot errors, but assume a simplified abstraction of the functionality of individual network devices, which may deviate from the real implementation. In this paper, we propose a technique for high-coverage testing of end-to-end network correctness using the real software that is deployed in these networks. Our design is effectively a hybrid, using an explicit-state model checker to explore all network-wide execution paths and event orderings, but executing real software as subroutines for each device. We show that this approach can detect correctness issues that would be missed both by existing verification and testing approaches, and a prototype implementation suggests the technique can scale to larger networks with reasonable performance.

CCS CONCEPTS

• **Networks** → *Network properties*;

KEYWORDS

Network Verification, Correctness

ACM Reference Format:
Santhosh Prabhu, Gohar Irfan Chaudhry, Brighten Godfrey, and Matthew Caesar. 2018. High-coverage testing of softwarized networks. In *SecSoN '18: Workshop on Security in Softwarized Networks: Prospects and Challenges, August 24, 2018, Budapest, Hungary.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3229616.3229617

1 INTRODUCTION

Networks have long been complex entities, with dozens of protocols, thousands of lines of configurations, and the need to satisfy many services and users. In service providers and increasingly in enterprises, these networks are changing, incorporating software components on commodity hardware. Software network functions allow sophisticated functionality to be deployed (like security services or transcoding video streams) and improve the ease of deploying new services. However, the diversity of such software can increase complexity of the network even further, and this is compounded by the desire to make rapid changes. We also speculate that the lack of industry standards for middlebox software may increase the chance of an accidental misconfiguration in some environments. The result is that ensuring correctness and security of modern networks with software components is critical and challenging.

Network verification has brought mathematical rigor into policy enforcement, generally checking either data plane state [9, 12, 14] or configurations [2, 5, 6, 16]. In either case, the verifier assumes an abstraction or model of the network. For example, a verifier may model IP longest prefix match forwarding hardware, whether packet filters are applied on ingress or egress ports, and how BGP route selection breaks ties. If the model does not match reality, then the verifier may produce incorrect results. In some cases, like data plane verification of traditional switches and routers, these assumptions might typically be realistic. But for a network incorporating extensive software elements, assuming a network model becomes a serious limitation, for several reasons:

- Implementation bugs: Given the highly specialized nature of middleboxes, there is both a high likelihood of bugs occurring, and also the risk of them being uncaught for a significant length of time. These bugs (or simply implementation quirks) may cause network policy violations even if the network operator has configured the middlebox fully correctly. Writing a bug-for-bug faithful model of the software would be close to impossible.
- Lack of a model: Part of the point of building a softwarized network is to be able to code custom features, behaviors, and even whole distributed systems. As such, we may lack a starting point for a model, unlike data plane and control plane elements that typically operate with standardized protocols (BGP, spanning tree, etc.).

An alternate approach is to emulate the network. For example, instead of simulating BGP, CrystalNet [13] runs an emulation of the network using real router VMs, and this could apply to Virtualized Network Functions (VNFs) as well. The disadvantage is that emulation has low coverage of the network's execution paths; even an approach like ATPG [18] that injects many packets would be unable to guarantee exploration of the network's dynamics, which are perhaps more important in a softwarized environment built to automatically respond to network events.

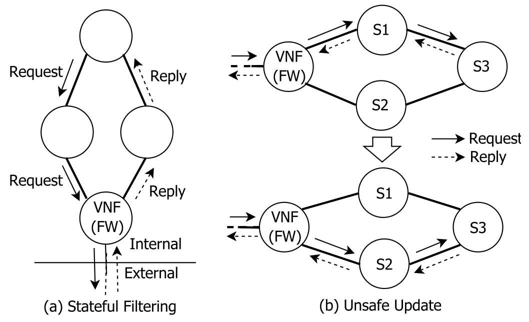


Figure 1: Policy violations in VNFs

In this paper, we ask: How close can we come to getting the best of both worlds, with the faithfulness of real software and the high coverage of verification?

We initiate an exploration of that question with a hybrid approach. Intuitively, to retain high accuracy, we need to execute the actual running software with its actual configuration for each *individual* component. But we use those components, and partial executions of them, as building blocks to assemble and reassemble in an exhaustive exploration of network-wide execution paths driven by an explicit-state model checker. Our key contributions are as follows:

- We propose a hybrid approach combining emulation-based testing and model-based verification, and show through examples that it can find intent violations that would be missed by each approach individually (§2).
- We design such a hybrid system, Plankton-neo (network emulation option), which builds on the Plankton network verification platform [16] but inserts invocation of real software network elements in its execution loop, inserting packets and interpreting the results. Our design explores how to invoke these devices efficiently by both running parallel instances and state restoration.
- We implement and evaluate a prototype of Plankton-neo, showing that it can catch actual problems that arise with the `iptables` software firewall on Linux, and that it can validate a multi-tenant data center with 196 routers and 64 tenants under 80 minutes.

While more remains to be done (e.g., improving performance and experimenting with more diverse software elements), we believe these results show that hybrid approaches are both valuable and potentially feasible for checking security and correctness properties of softwarized networks. We expect that verification-emulation hybrids may offer a rich design space to explore in the future.

2 MOTIVATION

In this section, we review automated debugging of VNFs, and illustrate using examples how Plankton-neo can advance the state of the art. Past work in debugging softwarized networks falls into two broad categories, as we discuss below.

2.1 Model-based verification

Model-based verification techniques such as VMN [15] use manually created models for the network components, including middleboxes, and use formal techniques such as model checking to analyze the behavior of the system under various possible execution paths. While these techniques perform an exhaustive analysis, their accuracy depends on the correctness of the models they use. Consider the network fragment illustrated in Figure 1(a). It shows a virtualized firewall running on a server, configured to perform standard stateful filtering — block connection attempts from outside the organization, but allow replies to past requests. Additionally, routing has computed forwarding paths such that requests and replies pass through different switches (but the same firewall). In this setup, one may expect that if a request originates from inside, it should not be blocked, nor should be its reply. An intuitive model-based verification platform may even declare that this condition holds. However, when running a virtualized firewall using `iptables` on Linux, the behavior of this setup depends on specific kernel configuration variables. For example, when the `rp_filter` variable is enabled, the kernel performs reverse-path filtering (RPF) — for each packet it forwards, it constructs a hypothetical reply and tries injecting it into the outgoing port for the current packet. If this hypothetical reply does not go out the incoming interface of the current packet, the current packet is dropped. This would cause the reachability condition above to be violated.

2.2 Emulation-based testing

CrystalNet [13] executes real software implementations of network components in emulated environments, and the results can be checked for correctness. However, this technique executes only one (potentially non-deterministic) run of the network at a time, and may hence miss issues that may occur only under certain specific conditions of protocol execution, packet delivery, failures, etc.

In particular, softwarized networks may have issues that neither model-based verification nor emulation-based testing would find. Consider Figure 1(b). The network is being updated from using the path through S1 (top figure) to using the path through S2 (bottom figure). Suppose the devices' forwarding entries are updated in the order S2, FW, S3, S1. In this network, we may require that for any request was sent previously, the reply is not dropped. If the firewall was implemented on Linux with `rp_filter` enabled, there exists a potential violation of the correctness policy, as follows. The request reaches the firewall, and is sent to S3 through S1. S2 and FW get updated to the new configuration. The reply reaches S3, and gets forwarded to FW through S1. At this point, RPF at FW would drop the reply, because according to FW's new configuration, the packet should have come from S2 and not S1.

The above issue is unlikely to be caught by emulation, since it manifests only when events are interleaved in a particular order. While a fully accurate model of the VNF firewall *could* detect this, creating such a model is near-impossible in practice. Consider the `rp_filter` variable. The default value in the Linux kernel for this variable is 0, which disables RPF. However, some Linux distributions set it to a different value (most commonly 1), enabling RPF. But even using the same distribution is not sufficient to guarantee consistency: RedHat 5 and 6 both use the same value 1 for `rp_filter`, but the

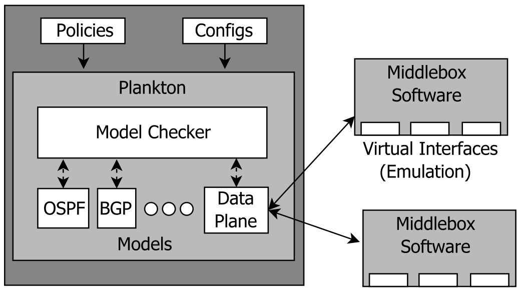


Figure 2: System organization

semantics for this configuration value is different in RedHat 5 and RedHat 6 [1]. In other words, bugs and the lack of standardization pose a formidable challenge to writing (and keeping up-to-date) high-fidelity models for middleboxes.

This issue is one example among countless that might arise. Our goal is to be able to discover such issues with high-coverage testing, particularly with respect to the dynamic events that software network elements will increasingly create (compared to static data planes). The hybrid technique we describe next is the first form of automated analysis that can reliably diagnose the example above. Of course, we should expect that we will give up *some* coverage compared with full formal verification; we will discuss this limitation later.

3 DESIGN

Figure 2 illustrates the organization of Plankton-neo. It builds upon Plankton [16], which uses model checking to formally verify network models. We shall now briefly describe the architecture of Plankton.

3.1 Plankton Design

Plankton models the control plane protocols and the network's environment as agents, which are written manually to simulate the behavior of their real-world counterparts. The key difference between Plankton's model and a real network is that the agents in Plankton are designed to operate on *equivalence classes* of packets rather than a single packet at a time (For example, the OSPF routing agent performs route computation for each equivalence class separately.) For each equivalence class, the collection of agents may execute in various non-deterministic ways, causing the overall network state to evolve differently. Plankton uses an explicit-state model checker to exhaustively explore all the possible executions, verifying that correctness policies are not violated in any of these executions. In order to do so, the model checker invokes a dataplane verifier as an oracle that determines the truth of Boolean predicates over individual dataplane states, and uses the results to reason about temporal policies defined over these predicates.

Plankton-neo implements its hybrid testing technique by augmenting Plankton's design in certain specific ways, which we now discuss.

3.2 The Dataplane Agent

Plankton's use of a dataplane verifier to check individual dataplane states means that in terms of network state transitions, a single dataplane state is atomic. In other words, Plankton cannot reason about fine-grained state changes caused by packets moving through the dataplane. This makes Plankton unusable for any form of policy verification involving dynamic dataplane elements, whose behavior is closely tied to these fine-grained changes. In order to overcome this limitation, Plankton-neo replaces the dataplane verifier in Plankton with a *dataplane agent*. The dataplane agent mimics the dataplane of the network, by forwarding a single *symbolic* packet, as dictated by the forwarding rules computed by the control plane agents. In other words, the dataplane agent is a manually written model for the forwarding behavior of the devices in the network. However, the dataplane agent makes one important exception. While simple forwarding devices such as switches and routers are represented using the manually-created model, middlebox software elements execute in their original form. For each middlebox in the network, a "virtual device" is created on the host that runs Plankton-neo, consisting of virtual interfaces in one-to-one correspondence with the actual interfaces on the middlebox, and running the same software. The configuration supplied to the middlebox is updated to operate over the virtual interfaces rather than the original physical interfaces. Since middlebox software can work only with concrete packets and not equivalence classes, a fully instantiated *representative* is picked for each equivalence class. The difference between a symbolic packet and a representative packet is subtle, but important. A symbolic packet is a logical entity, that merely denotes an equivalence class, whereas a representative packet is a real packet that can be processed by network devices, chosen from the many packets that constitutes the equivalence class. We elaborate on the computation of equivalence classes and the selection of the representatives later.

The model checker in Plankton-neo exhaustively explores the various execution paths of system, for each equivalence class. In addition to protocol execution, topology changes etc, in Plankton-neo, this includes the hop-by-hop forwarding of the symbolic packet by the dataplane agent. When the symbolic packet reaches a middlebox, the representative packet for the equivalence class is injected into the appropriate interface of the emulated copy of the middlebox. Then, the fate of the injected packet is observed, and the dataplane agent interprets the observation as an action performed on the symbolic packet. In essence, each middlebox defines an "API" that allows the verification algorithm to query for the outcome of packets reaching one of the interfaces. Using such a system, we can verify a variety of policies about end-to-end correctness of the network. Perhaps the most relevant are temporal policies that pertain to how the network changes its behavior in response to traffic. For example, a network with a web cache may state that *No HTTP requests should be sent to the server more than once*.

3.3 Saving and restoring middlebox state

As the model checking algorithm exhaustively searches through possible executions of the network looking for policy violations, it will be required to perform packet injection into various middleboxes many times, under various hypothetical scenarios. Each time such an injection happens, the intention of the algorithm is to observe the

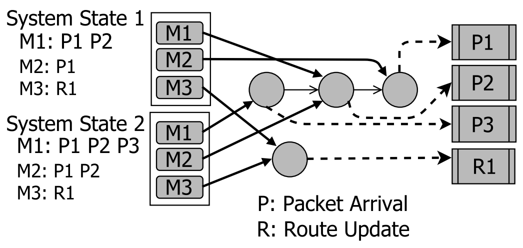


Figure 3: Duplicate eliminated update history storage

fate of the packet if it was to reach the middlebox under the specific circumstances that the algorithm has contrived. So, we require the state of the virtual middlebox to match the state intended by the algorithm before the packet can be injected. In order to do so, we implement the network model in the following way: In addition to the middlebox software running on the emulated interfaces, for each middlebox, Plankton-neo stores the initial state, plus a list of all updates that have been made to the middlebox. We define this update trace to hold two types of updates: any control plane messages updating the middlebox state, such as routing table changes, and any packets that were previously injected, along with the interfaces where the injection happened. When a new packet needs to be injected into a middlebox, we first match the emulator state with the one described by the trace — by first restarting middlebox software with the initial configuration, and then replaying all the historical updates that have supposedly happened in the past. This approach of putting a middlebox into a desired state is attractive, because it does not require any knowledge of the internal workings of the software. It is also more practical than snapshotting the virtual memory of the middlebox in order to store its state. However, it does assume that the software is deterministic, and has no dependency on timing. In other words, starting from an initial state, replaying the same sequence of updates is assumed to put the middlebox in the same final state.

We rely on two observations to make our update history storage scalable:

- The same update may be part of multiple update histories.
- Many update histories may differ from each other only in terms of a few updates.

Our optimization reduces memory overhead by leveraging these redundancies in update histories. Updates that have historically been made to any middlebox in the network are kept in a hash table, and reused. No update is created twice, at any point during the exploration of the system. Furthermore, every sequence of updates is also duplicate-eliminated through hashing. This happens not only for full sequences, but also subsequences. Figure 3 illustrates this layout. With this technique, each middlebox needs to store only a pointer as its entire history (In addition to the amortized cost of storing the updates).

3.4 Computing equivalence classes and representatives

Just like Plankton, the network model in Plankton-neo is defined in terms of equivalence classes. However, while Plankton allows the equivalence classes to be abstract (*Request Class*, *Reply Class*, *Suspicious Class* etc), they cannot be when working with real software because for each class, we need to inject a representative concrete packet into the emulated middlebox. Plankton-neo can use any set of equivalence classes that has the property that all members of the class have identical behavior throughout the network, in the data and control planes. A good starting point for equivalence classes is to compute each equivalence class as a maximal set of packets that such that the network-wide configurations specify that they should be handled in the same way; e.g., if the network had a single firewall, the collection of all packets that it is configured to drop could be one equivalence class; this could be subdivided as other devices with different configurations are added. (This assumes all devices have configurations that specify the relevant packet handling, which is generally true for industry-standard switches, routers, and middleboxes.)

The representative for each packet class can be picked arbitrarily. However, once it is injected, the state within the middlebox may change in ways that affect future packets, effectively creating a finer set of equivalence classes, each of which must be checked if we want full coverage. But these finer classes are not externally visible. Plankton-neo currently takes an educated guess: it considers packets that would be a reply to the injected packet (i.e., P' with source and destination IP swapped) as a new class, and will therefore later try injecting such a packet into the network.

3.5 The coverage tradeoff

As we discussed in § 1, Plankton-neo's hybrid approach attempts to combine the high coverage of model checking with the fidelity of real software. The gap between the hybrid approach and a fully formal approach is defined by two major assumptions — the accuracy of equivalence class selection, and the validity of the determinism assumption. Plankton-neo's guess of the relevant set of classes is naturally not guaranteed to be perfect (so that Plankton-neo's execution of the middlebox with a single representative packet per equivalence class is insufficient to cover all behavior). The closer these classes are to the actual equivalence classes defined by hidden states in the middlebox software, the higher the coverage. The assumption of deterministic software execution also has a direct impact on the coverage achieved by Plankton-neo. Any internal non-determininism may cause missed execution paths, since we execute each middlebox just once per injected packet. Non-determinism may also interfere with replay, though deterministic replay techniques may apply.

Since either of the two assumptions may be violated in practice, Plankton-neo may not always achieve full coverage of the state space. However, Plankton-neo does achieve much better coverage of execution paths as compared to simple testing, while avoiding the need for a full behavioral model and the lack of fidelity to real software that is inherent in model-based verification.

4 PROOF OF CONCEPT

We implemented a simple version of our technique over Plankton, with support for Linux-based middleboxes. For emulating VNF devices, we create `tap` interfaces, and create separate routing tables in the kernel for the emulated middleboxes. When the verification algorithm invokes packet injection, the representative is sent into

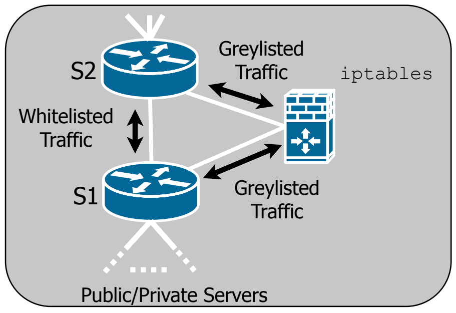


Figure 4: Policy enforcement in a multi-tenant DC

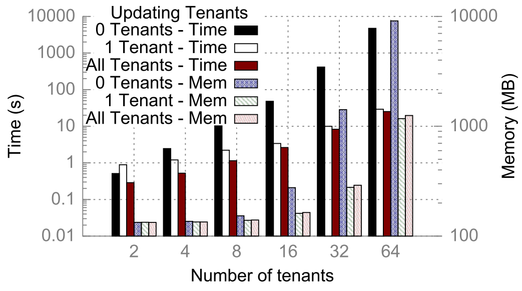
the appropriate `tap` interface. A special data payload is used to distinguish the injected packet from any other packets that may originate from the kernel (we assume that payloads are not modified by the middleboxes). If the packet does not make it to any of the `tap` interfaces within a configurable timeout (we use 50ms in our experiments), it is assumed to be dropped.

Using this implementation, we attempted the problem described in § 1. We verify a network with 4 switches and a virtualized `iptables` firewall, as illustrated in Figure 1 a. The policy we verify is that the request and reply traffic reach their respective destinations. When reverse path filtering is enabled, the firewall blocks the request traffic, causing the policy to fail. However, when we disable reverse path filtering in the kernel, the policy passed.

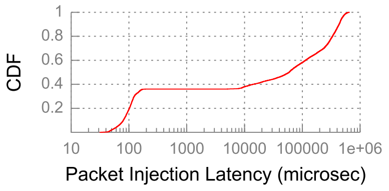
To test our approach at scale, we perform an experiment inspired by [7]. We consider a multi-tenant datacenter with varying number of tenants, with two types of traffic — whitelisted or greylisted. Each tenant also has two types of servers — public or private. Initially, each tenant allows all traffic to public servers, but for private servers, whitelisted traffic is always allowed, and for greylisted traffic, only replies to past requests are allowed. We use `iptables` running on Ubuntu 14.04 to enforce the policies, as shown in Figure 4. This use of a real VNF implementation is what distinguishes our experiment from similar experiments in past work [7, 15]. In our test, we assume that some of the tenants now wish to reclassify HTTP from greylisted to whitelisted . This change is implemented by updating two routers. Naturally, at any point during the transition, the correctness spec for the network states that no legitimate reply to a past request be dropped. In our experiments, Plankton-neo finds the following violation. Access switch $S1$ gets updated before the request packet reaches it. So, the request is forwarded directly to $S2$, without passing through the firewall. The reply packet reaches $S2$, before $S2$ is updated with the new policy. Abiding by the old policy, $S2$ forwards the reply to the `iptables` firewall. The firewall drops the reply, since it has not seen the request.

We perform the above experiment with different number of tenants, with varying number of tenants performing the whitelisting change. Figure 5 a illustrates the time and memory used by a single run of the test (Time and number of tenants axes are logarithmic). When the fraction is 0, there are no changes being made, and hence, the policy passes. When at least one tenant updates its policy, a violation of the policy exists, and we correctly find it.

Figure 5 a provides some interesting insights into the nature of the problem. Intuitively, the verification problem is hardest when



(a) Time and Memory Overhead



(b) Latency CDF

Figure 5: Measurements from DC experiment

Experiment	Models	Real Software
64 Tenants, no update	36.66 s	4732.13 s
32 Tenants, no update	5.41 s	413.84 s
64 Tenants, all tenants update	22.47 s	27.73 s

Figure 6: Comparison with model-based verification

there is no update happening in the network. This is because the testing procedure has to exhaust every possible execution and finally declare that the policy holds. This is reflected in the time and memory consumption.

Figure 5 b illustrates the CDF of the latency incurred in performing packet injection and observing the outcome. It is clear that there exist two different groups of latency measurements. The faster one indicates packet processing at line rate, without having to restore middlebox state or wait for timeouts. This is the case where the emulated middlebox is in the same state as the one the verifier requires it to be, and the packet that is newly injected does not get dropped. The slower measurements may be due to timeout or state restoration.

One aspect we wish to evaluate is the overhead of verifying middleboxes in their true form as opposed to using models. To this end, we repeat the multi-tenant datacenter experiment with models rather than actual software for the virtualized firewalls. The models we use are similar to the ones used by VMN [15]. They perform stateful firewalling, without any support for advanced features such as reverse-path filtering. Figure 6 compares the time taken by the two approaches. It can be seen that incorporating real software is at times as much as two orders of magnitude slower. This is, however, to be expected. Determining the behavior of a packet at a middlebox may take as long as 1 second when done using actual software, as illustrated in Figure 5 b. With our simplistic firewall model this operation requires only a few machine instructions. We believe the increased time overhead is justified by the greater fidelity of the verification process. In fact, the results obtained using models are

Experiment	Single emulation	Multiple Emulation
64 Tenants, no update	5835.52 s	4732.13 s
32 Tenants, no update	680.28 s	413.84 s
64 Tenants, all tenants update	29.22 s	27.73 s

Figure 7: Single emulated device vs. one per middlebox

correct only if we configure the middleboxes to behave exactly the way the models expect them to behave. Nevertheless, we are working on techniques to optimize the process further.

5 DISCUSSION

We now discuss some specific elements of our design.

5.1 Number of emulated devices

The design we described in § 3 uses a separate set of virtual interfaces and uses a separate emulation of each middlebox in the network. Alternatively, we can also have multiple middleboxes emulated on a single (or smaller number of) middlebox instances. This is possible because the model checking algorithm only checks the behavior of one middlebox at any given time. When the packet needs to be injected into a particular middlebox, we run that middlebox over the virtual interfaces, and use update replay to put the middlebox in the required state. But doing so has an impact on performance, due to the inherently high latency incurred in update replay (See Figure 5(b)). When using a separate emulation for each middlebox, there is a higher likelihood that the emulated middlebox is in the same state as expected by the verifier, and hence, does not require additional state restoration. We empirically evaluate this design choice by repeating the multi-tenant DC experiment using a single emulation setup for all middleboxes. Figure 7 shows that for our test case, having separate emulation for each middlebox reduces execution time by as much as 39%. This can be further improved by having *multiple* emulated devices to run the same middlebox, so we can save different states. In general, the relationship between the middleboxes in the network and the emulation setups in Plankton-neo can be many-to-many. A full exploration remains to be done.

5.2 Matching the production environment

As stated in § 1, the primary goal of Plankton-neo is to test middleboxes as close as possible to their production settings. This includes executing the actual middlebox software, matching the operating system, the system configuration etc. In our presentation so far, we have made two implicit assumptions — all the VNFs in the network run on the same operating system with identical configuration, and the verification is performed on a host that is identical. These are not fundamental assumptions. The design of the system also allows for the packet injection to be performed on multiple machines, as long as the outcome is conveyed to the model checking algorithm.

5.3 Beyond middleboxes

Although our discussion in this paper has focused on testing middlebox software, the technique itself is not strictly limited to middleboxes. Any network component that allows state restoration through update replay may be incorporated into Plankton-neo in a similar manner. This may include control plane software or even hardware. We intend to study the various challenges and opportunities in this space in greater detail.

6 RELATED WORK

Data plane verification: This class includes offline network verification techniques such as Anteater [14] and HSA [11], and more recent real-time tools such as VeriFlow [12], NetPlumber [10] and DeltaNet [9]. They are designed for verifying single dataplane states, without taking into account the possible dynamics that are introduced by middleboxes. Moreover, they are perform the verification using an internal model of the dataplane, rather than the actual hardware/software.

Configuration verification: Configuration verification tools such as Minesweeper [2] and ARC [6] verify the network configuration rather than a single dataplane state. However, they too are not capable of handling the dynamics of softwarized networks.

Model-based verification: VMN [15] uses manually created models of dynamic dataplane devices and an SMT-based problem formulation to verify temporal policies over softwarized networks. In § 2, we discussed the limitations of model based techniques, and illustrated that Plankton-neo's hybrid approach can detect real issues that would be missed by techniques such as VMN. NICE [3] is architecturally similar to Plankton-neo in that it executes real software for some parts of the network, along with models for the others. However, Plankton-neo targets verification of dataplanes with middleboxes, rather than Openflow, as NICE does.

Emulation: CrystalNet [13] performs an emulation of actual device virtual machines, and its results could be fed to a data plane verifier. However, as we illustrated in § 2, Plankton-neo can uncover correctness issues that may occur only under specific conditions that cannot be reliably produced by emulation.

Software dataplane verification: Past work on software dataplane verification has focused on correctness of the individual software components, such as software implementations of single routers or NAT [4, 17]. In contrast, Plankton-neo performs end-to-end correctness testing of networks that deploy such software components.

Concolic execution: The hybrid approach proposed by Plankton-neo closely resembles concolic execution [8] used for software testing. We expect to leverage past work in this space to improve Plankton-neo further.

7 CONCLUSION

We proposed Plankton-neo, a high coverage testing technique for softwarized networks. By combining real implementations of middleboxes with models for standardized network components, Plankton-neo aims to combine the advantages of both emulation as well as model-based verification. Our experiments have shown that Plankton-neo can be a powerful and scalable tool that can assist in faster adoption of software in networks.

The authors thank the anonymous reviewers for their insightful comments. This work was supported by NSF CNS Award #1513906.

REFERENCES

[1] [n. d.]. Red Hat Customer Portal. https://access.redhat.com/solutions/53031. ([n. d.]). Accessed: 2018-03-20.

[2] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *Proceedings of SIGCOMM '17*. ACM, New York, NY, USA, 155–168.

[3] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Presented as part of NSDI 12*. USENIX, San Jose, CA, 127–140.

[4] Mihai Dobrescu and Katerina Argyraki. 2014. Software Dataplane Verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 101–114.

[5] 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 *OSDI 16*. GA, 217–232.

[6] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *Proceedings of the SIGCOMM 2016*.

[7] Soudeh Ghorbani and Philip Brighten Godfrey. 2017. COCONUT: Seamless Scale-out of Network Elements.. In *EuroSys*. 32–47.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223. https://doi.org/10.1145/1064978.1065036

[9] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *NSDI 17*. USENIX Association, Boston, MA, 735–749.

[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 *NSDI 13*. Lombard, IL.

[11] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *NSDI 12*. USENIX, San Jose, CA, 113–126.

[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 NSDI '13*. USENIX Association, Berkeley, CA, USA, 15–28.

[13] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. 2017. CrystalNet: Faithfully Emulating Large Production Networks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 599–613.

[14] 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 SIGCOMM '11*. New York, NY, USA, 290–301. https://doi.org/10.1145/2018436.2018470

[15] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths. In *NSDI 17*. USENIX Association, Boston, MA, 699–718.

[16] Santhosh Prabhu, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2017. Predicting Network Futures with Plankton. In *Proceedings of the First Asia-Pacific Workshop on Networking (APNet'17)*. ACM, New York, NY, USA, 92–98. https://doi.org/10.1145/3106989.3106991

[17] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. In *Proceedings of the SIGCOMM '17*. ACM, New York, NY, USA, 141–154.

[18] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic Test Packet Generation. In *Proceedings of CoNEXT '12*. ACM, New York, NY, USA, 241–252.