Plankton: Scalable network configuration verification through model checking

Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P. Brighten Godfrey, and Matthew Caesar, *University of Illinois at Urbana–Champaign*

https://www.usenix.org/conference/nsdi20/presentation/prabhu

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Plankton: Scalable network configuration verification through model checking

Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P. Brighten Godfrey, Matthew Caesar
University of Illinois at Urbana-Champaign

Abstract

Network configuration verification enables operators to ensure that the network will behave as intended, prior to deployment of their configurations. Although techniques ranging from graph algorithms to SMT solvers have been proposed, scalable configuration verification with sufficient protocol support continues to be a challenge. In this paper, we show that by combining equivalence partitioning with explicit-state model checking, network configuration verification can be scaled significantly better than the state of the art, while still supporting a rich set of protocol features. We propose Plankton, which uses symbolic partitioning to manage large header spaces and efficient model checking to exhaustively explore protocol behavior. Thanks to a highly effective suite of optimizations including state hashing, partial order reduction, and policy-based pruning, Plankton successfully verifies policies in industrial-scale networks quickly and compactly, at times reaching a 10000× speedup compared to the state of the art.

1 Introduction

Ensuring correctness of networks is a difficult and critical task. A 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 [19] and HSA [13], the state of the art has evolved to support real-time data plane verification [15, 12], and more recently, analysis of configurations [6, 5, 7, 1, 25].

Configuration analysis tools such as Batfish [6], ERA [5], ARC [7] and Minesweeper [1] are designed to take as input a given network configuration, a correctness specification, and possibly an "environment" specification, such as the maximum expected number of failures, external route advertisements, etc. Their task is to determine whether, under the given environment specification, the network configuration will always meet the correctness specification. As with most formal verification domains, the biggest challenge in configuration analysis is scalability. Being able to analyze the behavior of multiple protocols executing together is a non-trivial task. Past verifiers have used a variety of techniques to try to surmount this scalability challenge. While many of them sacrifice their correctness or expressiveness in the process, Minesweeper maintains both by modeling the network using SMT constraints and performing the verification using an SMT solver. However, we observe that this approach scales poorly with increasing problem size (4+ hours to check a 245-device network for loops, in our experiments). So, this paper addresses the following question: *Can a configuration verification tool have broad protocol support, and also scale well?*

We begin our work by observing that scalability challenges in configuration verification stem from two factors — the large space of possible packet headers, and the possibly diverse outcomes of control plane execution, particularly in the presence of failures. We believe that general purpose SAT/SMT techniques are not as well equipped to tackle these challenges as domain-specific techniques specifically designed to address them. In fact, these challenges have been studied extensively, in the domains of data plane verification and software verification. Data plane verification tools analyze the large header space to determine all possible data plane behaviors and check their correctness. Software verification techniques explore the execution paths of software, including distributed software, and identify undesirable executions that often elude testing. Motivated by the success of the analysis algorithms in these domains, we attempted to combine the two into a scalable configuration verification platform. The result — Plankton — is a configuration verifier that uses equivalence partitioning to manage the large header space, and explicit-state model checking to explore protocol execution. Thanks to these efficient analysis techniques, and an extensive suite of domain-specific optimizations, Plankton delivers consistently high verification performance.

Feature	Batfish	BagPipe	ARC	ERA	Minesweeper	Plankton
All data planes, including failures	○	○	◐	○	●	●
Support beyond specific protocols	●	○	○	●	●	●
Soundness when assumptions hold	●	●	●	◐*	●	●

∗ *For segmentation policies only*

Figure 1: Comparison of configuration verification systems

Our key contributions are as follows:

- We define a configuration verification paradigm that combines packet equivalence computation and explicit-state model checking.
- We develop optimizations that make the design feasible for networks of practical scale, including optimizations to reduce the space of event exploration, and techniques to improve efficiency of exploration.
- We implement a Plankton prototype with support for OSPF, BGP and static routing, and show experimentally that it can verify policies at practical scale (less than a second for networks with over 300 devices). Plankton outperforms the state of the art in all our tests, in some cases by as much as 4 orders of magnitude.

2 Motivation and Design Principles

Configuration verifiers take as input the configuration files for network devices, and combine them with an abstraction of the lower layers — the distributed control plane, which executes to produce data plane state, and the forwarding hardware that will use that state to process traffic. They may additionally take an *environment specification*, which describes interactions of entities external to the network, such as possible link failures, route advertisements, etc. Their task is to verify that under executions enabled by the supplied configuration and environment, correctness requirements are never violated. Configuration verifiers thus help ensure correctness of proposed configurations prior to deployment.

Figure 1 illustrates the current state of the art in configuration verification. As the figure shows, only Minesweeper [1] can reason about multiple possible converged data plane states of the network (e.g., due to topology changes or control plane non-determinism), while also having the ability to support more than just a specific protocol, and maintaining soundness of analysis. All other tools compromise on one or more key features. ARC [7], for example, uses graph algorithms to compute the multiple converged states enabled by failures, but only for shortest-path routing. As a result it cannot handle common network configurations such as BGP configurations that use LocalPref, any form of recursive routing, etc. The reason for the mismatch in Minesweeper's functionality in contrast to others is that it makes a different compromise. By using an SMT-based formulation, Minesweeper is able to achieve good data plane coverage and feature coverage, but pays the price in performance. As experiments show [2], Minesweeper scales poorly with network size, and is unable to handle networks larger than a few hundred devices in reasonable time. Our motivation for Plankton is simple — can we design a configuration verification tool without compromising scale or functionality?

Achieving our goal requires tackling two challenges: packet diversity and data plane diversity. Packet diversity, which refers to the large space of packets that needs to be checked, is easier to handle. We leverage the notion of *Packet Equivalence Classes* (PECs), which are sets of packets that behave identically. Using a trie-based technique similar to VeriFlow [15], we compute PECs as a partitioning of the packet header space such that the behavior of all packets in a PEC remains the same throughout Plankton's exploration. A more interesting aspect of PECs is how to handle dependencies across PECs without compromising performance. In Plankton, this is done by a dependency-aware scheduler designed to maximize independent analysis (§ 3.2).

Data plane diversity refers to the complexity of checking every possible converged data plane that an input network configuration may produce. It is the task of the control plane model to ensure coverage of these possible outcomes. Simulation-based tools (the best example being Batfish [6]) execute the system only along a single non-deterministic path, and can hence miss violations in networks that have multiple stable convergences, such as certain BGP configurations. ARC's graph-based approach accounts for possible failures, but can support only shortest-path routing. In order to overcome these shortcomings, Minesweeper, the current state of the art in terms of functionality, uses an SMT solver to search through possible failures and non-deterministic protocol convergence, to find any converged state that represents a violation of network correctness.

A key intuition behind our approach is that the *generic search technique employed by SMT solvers makes the core of the configuration verification problem much more difficult than it has to be*. Network control planes operate using simple algorithms which can not only be easily modeled in software, but can also find a protocol's outcome much more quickly than general-purpose SMT solving. In fact, the common case is that the control plane computes some variant of shortest or least-cost paths. To illustrate this point, we implemented simple single-source shortest path solvers in SMT (Z3) and a model checker (SPIN). The SMT formulation is implemented as constraints on the solution, while the model checker explores execution paths of the Bellman-Ford algorithm; and in this simplistic case the software has deterministic execution. The result is that the model checker approach is similar to a normal execution of software, and is around 12,000× faster even in a moderate-sized fat tree network of $N = 180$ nodes (Figure 2).

Of course, this is intentionally a simplistic, fully-

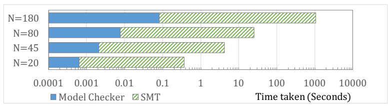


Figure 2: Comparison of two ways to compute shortest paths.

deterministic case with simple implementations. The point is that the model checking approach begins with a huge advantage — so huge that it could explore many non-deterministic execution paths and still outperform SMT. This leads to our next key intuition: *the effect of non-determinism is important, but the amount of "relevant" non-determinism is limited.* Networks can experience "relevant" non-determinism like the choice of what failures occur, and protocol execution; as well as "irrelevant" non-determinism like message timing that ultimately doesn't affect the outcome. Configurations and protocols are usually designed to keep the outcome mostly deterministic, with non-deterministic branch points ultimately leading to one or a small number of different converged states.

Motivated by this intuition, we create a control plane model that incorporates the possible non-deterministic behaviors, but we also implement optimizations so that when the *outcome* of the executions is actually deterministic, the "irrelevant" non-determinism is pruned enough that performance is comparable to simulation. This model is exhaustively explored by SPIN, a model checker designed for Promela programs. SPIN performs a depth-first search on the state space of the program, looking for states that violate the policy being checked. We further assist SPIN through optimizations that minimize the size of individual states, thus making the traversal process more efficient. Thanks to these two types of optimizations, Plankton achieves our goal of scalable, general-purpose configuration verification.

3 Plankton Design

We now present Plankton's design, illustrated in Figure 3.

3.1 Packet Equivalence Classes

The first phase in Plankton's analysis is the computation of Packet Equivalence Classes. As we discussed in § 2, Plankton uses a trie-based technique inspired by VeriFlow. The trie in Plankton holds prefixes obtained from the configuration, including any prefixes that are advertised (explicitly or automatically), any prefixes appearing in route maps, any static routes, etc. Each prefix in the trie is associated with a config object, that describes any configuration information that is specific to that prefix. For example, consider Figure 4, which illustrates a highly simplified example where the prefixes `128.0.0.0/1` and `192.0.0.0/2` are being advertised over OSPF in a topology with 3 devices. The trie holds three config objects — the default, and one for each advertised prefix.

Once the trie is populated, Plankton performs a recursive traversal, simultaneously keeping track of where the prefix boundaries define division of the header space. For each known partition, it stores the most up-to-date network-wide config known. When the end of any new prefix is reached, the config object that is associated with it is merged with the network-wide config for the partition that

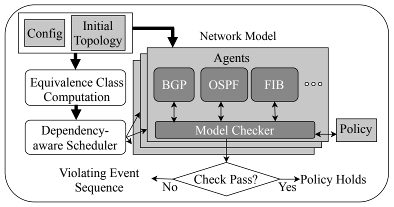


Figure 3: Plankton design

denotes the prefix. In our simple example, the traversal produces three classes defined by ranges — `[192.0.0.0, 255.255.255.255]` with two nodes originating prefixes, `[128.0.0.0, 191.255.255.255]` with only one origin, and `[0.0.0.0, 127.255.255.255]` without any node originating any prefix. As the example shows, each PEC-specific configuration computed this way will still include information about the original prefixes contributing to the PEC. Storing these prefixes may seem redundant. However, note that even within a PEC, the lengths of the prefixes that get advertised or get matched in route filters play a role in decision making.

3.2 Dependency-aware Scheduling

It is tempting to believe that Packet Equivalence Classes could be analyzed fully independently of each other, and that an embarrassingly parallel scheme could be used in the verification process. While this is indeed true sometimes, there can often be dependencies between various PECs. For example, consider a network that is running iBGP. For the various peering nodes to be able to communicate with each other, an underlying routing protocol such as OSPF should first establish a data plane state that forwards packets destined to the devices involved in BGP. In such a network, the manner in which OSPF determines the forwarding behavior for the device addresses will influence the routing decisions made in BGP. In other words, the PECs that are handled by BGP depend on the PECs handled by OSPF. In the past, configuration verification tools have either ignored such cases altogether, or, in the case of Minesweeper, modeled these classes simultaneously. Specifically, for a network of with n routers running iBGP, Minesweeper creates $n+1$ copies of the network, searching for the converged solution for the n loopback addresses and also BGP. Effectively, this turns the verification problem into one quadratically larger than the original. Given that configuration verification scales significantly worse than linearly in input size, such a quadratic increase in input size often makes the problem intractable.

Plankton goes for a more surgical approach. Once the PECs are calculated, Plankton identifies dependencies between the Packet Equivalence Classes, based on recursive routing entries, BGP sessions, etc. The dependencies are stored as a directed graph, whose nodes are the PECs, and directed edges indicate which PECs depend on which others. In order to maximize parallelism in verification runs across PECs, a dependency-aware scheduler first identifies strongly

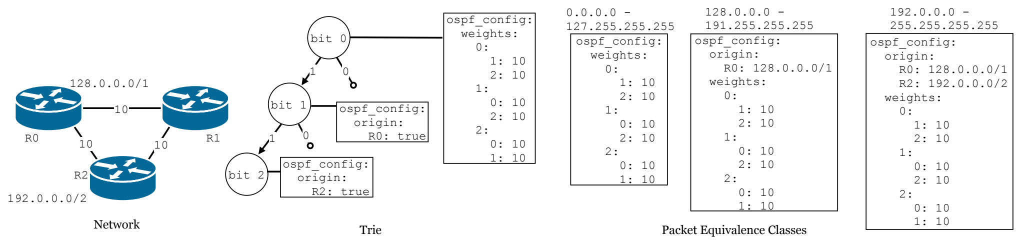


Figure 4: Packet Equivalence Class computation

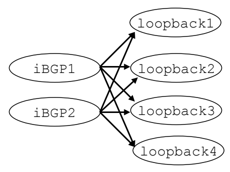


Figure 5: PEC Dependency Graph

connected components in the graph. These SCCs represent groups of PECs that are mutually dependent, and hence need to be analyzed simultaneously through a single verification run. In addition, if an SCC is reachable from another, it indicates that the upstream SCC can be scheduled for analysis only after the one downstream has finished. Each verification run is a separate process. For an SCC S, if there is another SCC S' that depends on it, Plankton forces all possible outcomes of S to be written to an in-memory filesystem (S will always gets scheduled first). Outcomes refer to every possible converged state for S, together with the non-deterministic choices made in the process of arriving at them. When the verification of S' gets scheduled, it reads these converged states, and uses them when necessary.

Minesweeper's technique of replicating the network roughly corresponds to the case where all PECs fall into a single strongly connected component. We expect this to almost never be the case. In fact, in typical cases, the SCCs are likely to be of size 1, meaning that every PEC can be analyzed in isolation, with ordering of the runs being the only constraint[*]. For example, Figure 5 illustrates the dependency graph for a typical case where two PECs are being handled in iBGP on a network with 4 different routers. The only constraint in this case is that the loopback PECs should be analyzed before the iBGP PECs can start. In such cases, Plankton's keeps the problem size significantly smaller, and maximizes the degree of parallelism that can be achieved.

When a PEC needs the relevant converged states of past PECs for its exploration, the non-deterministic choices may need to be coordinated across all these PECs. In particular, consider the choice of link failures: if we hypothetically executed one PEC assuming link L has failed and another PEC assuming L is working, the result represents an invalid execution of the overall network. Therefore, our current prototype matches topology changes across explorations. A second class of non-deterministic choices is protocol non-determinism. In our experiments, we have not seen cases of protocol non-determinism that requires matching across PECs. OSPF by its nature has deterministic outcomes, but on networks which have non-determinism in their internal routing (e.g., non-deterministically configured BGP for internal routing) and where message timing is correlated across PECs (e.g., via route aggregation), the system would need to coordinate this non-determinism to avoid false positives.

[*]An example where the SCCs are bigger than one PEC is the contrived case where there exists a static route for destination IP A whose next hop is IP B, but another static route for destination IP B whose next hop is IP A.

3.3 Explicit-state Model Checker

The explicit state model checker SPIN [9] provides Plankton its exhaustive exploration ability. SPIN verifies models written in the Promela modeling language, which has constructs to describe possible non-deterministic behavior. SPIN's exploration is essentially a depth-first search over the possible states of the supplied model. Points in the execution where non-deterministic choices are made represent branching in the state space graph.

Plankton's network model is essentially an implementation of the control plane in Promela. Our current implementation supports OSPF, BGP and static routing. Recall from § 3.1 that Plankton partitions the header space into Packet Equivalence Classes. For each SCC, Plankton uses SPIN to exhaustively explore control plane behavior. In order to improve scalability, Plankton also performs another round of partitioning by executing the control plane for each prefix in the PEC separately. This separation of prefixes is helpful in simplifying the protocol model. However, it does limit Plankton's support for route aggregation. While features such as a change in the routing metric can be supported, if there is a route map that performs an exact match on the aggregated prefix, it will not apply to the more specific routes, which Plankton models. Once the converged states of all relevant prefixes are computed, a model of the FIB combines the results from the various prefixes and protocols into a single network-wide data plane for the PEC.

In what follows, we present Plankton's network model that will be executed by SPIN. We will initially present a simple, unoptimized model, which is functionally correct but has significant non-determinism that is irrelevant to finding different converged states. Subsequently, in § 4, we discuss how Plankton attempts to minimize irrelevant non-determinism, making the execution of the deterministic fragments of the control plane comparable to simulation.

3.4 Abstract Protocol Model

To define a control plane suitable for modeling real world protocols such as BGP and OSPF, we look to the technique used by Minesweeper wherein the protocols were modeled as instances of the stable paths problem. Along similar lines, we consider the Simple Path Vector Protocol [8], which was originally proposed to solve the stable paths problem. We first extend SPVP to support some additional features that we wish to model. Based on this, we construct a protocol we call the *Reduced Path Vector Protocol*, which we show to be sufficient to correctly perform model checking, if we are interested only in the converged states of the network. We use RPVP as the common control plane protocol for Plankton. We begin with a brief overview of SPVP, highlighting our extensions to the protocol. Appendix A contains the full details of the protocol and our extensions.

3.4.1 SPVP and its extension

SPVP is an abstract model of real-world BGP, replacing the details of BGP configurations with abstract notions of import/export filters and ranking functions. For each node n and peer n', the import and export filters dictate which advertisements (i.e. the advertiser's best path to the *origin*) n can receive from and send to n', respectively. The ranking function for n dictates the preference among all received advertisements. These notions can be inferred from real-world configurations.

We slightly extend the original SPVP [8] to support more features of BGP. The extensions are as follows: we allow for multiple origins instead of a single one; the ranking function can be a *partial order* instead of a total one to allow for time based tie breaking; and to be able to model iBGP, we allow the ranking function of any node to change at any time during the execution of protocol.

It is well known that there are configurations which can make SPVP diverge in some or all execution paths. However, our goal is only to check the forwarding behavior in the converged states, through explicit-state model checking. So, we define a much simpler model that can be used, without compromising the soundness or completeness of the analysis (compared to SPVP).

3.4.2 Reduced Path Vector Protocol (RPVP)

We now describe RPVP, which is specifically designed for explicit-state model checking of the converged states of the extended SPVP protocol.

In RPVP, the message passing model of SPVP is replaced with a shared memory model. The network state only consists of the values of the best known path of each node at each moment (best-path). In the initial state, the best path of all nodes is $\perp$, except origins, whose best path is ε. At each step, the set of all enabled nodes (E) is determined (Algorithm 1, line 5). A node n is considered enabled if either i) the current best path p of n is invalid, meaning that the next hop in p has a best path that is not a prefix of p.

$$\mathrm{invalid}(n) \triangleq \mathrm{best\text{-}path}(\mathrm{best\text{-}path}(n).\mathrm{head}) \neq \mathrm{best\text{-}path}(n).\mathrm{rest}$$

Algorithm 1 RPVP

```
1:  procedure RPVP(:)
2:      Init : $\forall n \in N - Origins.\ \mathrm{best\text{-}path}(n) \leftarrow \perp$
3:      Init : $\forall o \in Origins.\ \mathrm{best\text{-}path}(o) = \varepsilon$
4:      while true do:
5:          $E \leftarrow \{n \in N \mid \mathrm{invalid}(n) \lor \exists n' \in \mathrm{peers}(n).\ \mathrm{can\text{-}update_n}(n')\}$
6:          if $E = \emptyset$ then:
7:              break
8:          end if
9:          $n \leftarrow \mathrm{nondet\text{-}pick}(E)$
10:         if $\mathrm{invalid}(n)$ then
11:             $\mathrm{best\text{-}path}(n) \leftarrow \perp$
12:         end if
13:         $U \leftarrow \mathrm{best}(\{n' \in \mathrm{peers}(n) \mid \mathrm{can\text{-}update_n}(n')\})$
14:         $n' \leftarrow \mathrm{nondet\text{-}pick}(U)$
15:         $p \leftarrow \mathrm{import}_{n,n'}(\mathrm{export}_{n',n}(\mathrm{best\text{-}path}(n')))$
16:         $\mathrm{best\text{-}path}(n) \leftarrow p$
17:     end while
18: end procedure
```

Or ii) there is a node n' among the peers of n that can produce an advertisement which will change the current best path of n. In other words, n' has a path better than the current best path of n, and the path is acceptable according to the export and import policies of n' and n respectively.

$$\mathrm{can\text{-}update_n}(n') \triangleq \mathrm{better}(\mathrm{import_{n,n'}}(\mathrm{export_{n',n}}(\mathrm{best}(n'))), \mathrm{best}(n))$$

Where $better_n(p, p')$ is true when path p is preferred over p' according to the ranking function of n.

At any step of the execution, if there is no enabled node, RPVP has reached a converged state. Otherwise a node n is non-deterministically picked among the enabled set (line 9). If the current best path of n is invalid, the best path is set to $\perp$. Among all peers of n that can produce advertisements that can update the best path of n, the neighbors that produce the highest ranking advertisements are selected (line 13). Note that in our model we allow multiple paths to have the same rank, so there may be more than one elements in the set U. Among the updates, one peer n' is non-deterministically selected and the best path of n is updated according to the advertisement of n' (lines 14-16). By the end of line 16, an iteration of RPVP is finished. Note that there are no explicit advertisements propagated; instead nodes are polled for the advertisement that they would generate based on their current best path when needed. The the protocol terminates once a converged state for the target equivalence class is reached. RPVP does not define the semantics of failure or any change to the ranking functions. Any topology changes to be verified are made before the protocol starts its execution and the latest version of the ranking functions are considered.

A natural question is whether or not performing analysis using RPVP is sound and complete with respect to SPVP. Soundness is trivial as each step of RPVP can be simulated using a few steps of SPVP. If we are only concerned about the converged states, RPVP is complete as well:

Theorem 1. *For any converged state reachable from the initial state of the network with a particular set of links L failing at certain steps during the execution of SPVP, there is an* *execution of RPVP with the same import/export filters and ranking functions equal to the latest version of ranking functions in the execution of SPVP, which starts from the initial state in which all links in L have failed before the protocol execution starts, and reaches the same converged state. Particularly, there is a such execution in which at each step, each node takes a best path that does not change during the rest of the execution.*

Proof. The proof can be found in the Appendix. □

Theorem 1 implies that performing model checking using the RPVP model is complete. Note that RPVP does not preserve all the transient states and the divergent behaviors of SPVP. This frees us from checking unnecessary states as we are only interested in the converged states. Yet, even the reduced state space has a significant amount of irrelevant non-determinism. Consequently, we rely on a suite of other domain-specific optimizations (§ 4) to eliminate much of this non-determinism and make model checking practical.

Note that our presentation of RPVP has assumed the that a single best path is picked by each node. This matches our current implementation in that we do not support multipath in all protocols. In a special-case deviation from RPVP, our implementation allows a node running OSPF to maintain multiple best paths, chosen based on multiple neighbors. While we could extend our protocol abstraction to allow multiple best paths at each node, it wouldn't reflect the real-world behavior of BGP which, even when multipath is configured, makes routing decisions based on a single best path. However, such an extension is valid under the constrained filtering and ranking techniques of shortest path routing. Our theorems can be extended to incorporate multipath in such protocols. We omit that to preserve clarity.

3.5 Policies

We primarily target verification of data plane policies over converged states of the network. Similar to VeriFlow [15], we don't implement a special declarative language for policies; a policy is simply an arbitrary function computed over a data plane state and returning a Boolean value. Plankton implements a Policy API where a policy supplies a callback which will be invoked each time the model checker generates a converged state. Plankton gives the callback the newly-computed converged data plane for a particular PEC, as well as the relevant converged states of any other PEC that the current PEC depends on. Plankton checks the callback's return value, and if the policy has failed, it writes a trail file describing the execution path taken to reach the particular converged state.

Our API allows a policy to give additional information to help optimize Plankton's search: *source nodes* and *interesting nodes*. We define two converged data plane states for a PEC to be *equivalent* if their paths from the source nodes have the same length and the same interesting nodes are in the same position on the path. Plankton may suppress checking a converged state if an equivalent one (from the perspective of that policy) has already been checked (§ 4.2 and § 4.3 describe how Plankton does this). If source and interesting nodes are not supplied, Plankton by default assumes that all nodes might be sources and might be interesting.

As an example, consider a waypoint policy: traffic from a set S of sources must pass through firewalls F. The policy specifies sources S, interesting nodes F, and the callback function steps through each path starting at S and fails when it finds a path that does not contain a member of F. As another example, a loop policy can't optimize as aggressively: it has to consider all sources.

In general, this API enables *any policy that is a function of a single PEC's converged data plane state*. We have found it simple to add new policies, currently including: Reachability, Waypointing, Loop Freedom, BlackHole Freedom, Bounded Path Length and Multipath Consistency [1]. We highlight several classes of policies that fall outside this API: (i) Policies that inspect the converged control plane state, as opposed to the data plane: while not yet strictly in the clean API, this information is easily available and we implemented a representative example, Path Consistency (§5), which asserts that the control plane state as well as the data plane paths for a set of devices should be identical in the converged state (similar to Local Equivalence in Minesweeper [1]). (ii) Policies that require multiple PECs, e.g., "packets to two destinations use the same firewall". This would be an easy extension, leveraging Plankton's PEC-dependency mechanism, but we have not performed a performance evaluation. (iii) Policies that inspect dynamic behavior, e.g., "no transient loops prior to convergence", are out of scope just as they are for all current configuration verification tools.

4 Optimizations

Although Plankton's RPVP-based control plane greatly reduces the state space, naive model checking is still not efficient enough to scale to large networks. We address this challenge through optimizations that fall into two major categories — reducing the search space of the model checker, and making the search more efficient.

4.1 Partial Order Reduction

A well-known optimization technique in explicit-state model checking, Partial Order Reduction (POR) involves exploring a set of events only in one order, if the various orderings will result in the same outcome. In general, precisely answering whether the order of execution affects the outcome can be as hard as model checking itself. Model checkers such as SPIN provide conservative heuristics to achieve some reduction. However, in our experiments, this feature did not yield any reduction. We believe this is because our model of the network has only a single process, and SPIN's heuristics are designed to apply only in a multiprocess environment. Even if we could restructure the model to use SPIN's heuristics, we do not expect significant reductions, as evidenced in past work [4]. Instead, we implement POR heuristics, based on our knowledge of the RPVP control plane[†].

4.1.1 Explore consistent executions only

To describe this optimization, we first introduce the notion of a *consistent* execution: For a converged state S and a partial execution of RPVP π , we say that π is consistent with S iff at each step of the execution, a node picks a path that is equal to it its best path in S and never changes it.

Readers may notice that Theorem 1 asserts the existence of a consistent execution leading to each converged state of the network, once any failures have happened. This implies that if the model checker was to explore only executions that are consistent with some converged state, completeness of the search would not be compromised (soundness is not violated since every such execution is a valid execution). Of course, when we start the exploration, we cannot know the exact executions that are consistent with some converged state, and hence need to be checked. So, we conservatively assume that every execution we explore is relevant, and if we get evidence to the contrary (like a device having to change a selected best path), we stop exploring that execution.

4.1.2 Deterministic nodes

Even when there is the possibility of non-deterministic convergence, the "relevant" non-determinism is typically localized. In other words, after each non-deterministic choice, there is a significant amount of *essentially* deterministic behavior before the opportunity for another non-deterministic choice, if any, arises. (We consider it analogous to endgames in chess, but applicable at any point in the protocol execution.) However, this is obscured by "irrelevant" non-determinism – particularly, ordering between node execution that doesn't impact the converged state. Our goal is to prune the irrelevant non-determinism to reduce the search space for Plankton's model checker.

For an enabled node n in state S with a single best update u, we say n is *deterministic* if in all possible converged states reachable from S, n will have the path selected after n processes u. Of course, with the model checker having only partially executed the protocol, it is highly non-obvious which nodes are deterministic! Nevertheless, suppose for a moment we have a way to identify at least *some* deterministic nodes. How could we use this information? At each step of RPVP, after finding the set of enabled nodes, if we can identify at least one deterministic enabled node, we choose *one* of these nodes and instruct SPIN to process its update.

[†]Since we wish to check all converged states of the network, it can be argued that any reduction in search space is essentially POR. But here, we are referring optimizations that have a localized scope.

(More specifically, we pick one arbitrarily.) This avoids the costly non-deterministic branching caused by having to explore the numerous execution paths where *each one* of the enabled nodes is the one executed next. The following theorem shows this is safe.

Theorem 2. *Any partial execution of RPVP consistent with a converged state can be extended to a complete execution consistent with that state.*

Proof. The proof can be found in the Appendix. □

By definition, choosing *any* deterministic node as the single node to execute next produces a new state that remains consistent with all possible converged states reachable from the prior state. Thus, Theorem 2 implies this deterministic choice does not eliminate any converged states from being reachable, preserving completeness. Note that this optimization does not require the entire network to have one converged state; it can apply at every step of the execution, possibly between non-deterministic choices.

What remains is the key: how can we identify deterministic nodes? Perfect identification is too much to hope for, and we allow our heuristics to return fewer deterministic nodes than actually exist. We build heuristics that are specific to each routing protocol, prioritizing speed and simplicity above handling atypical cases like circular route redistribution.

For OSPF, our detection algorithm runs a network-wide shortest path computation, and picks each node only after all nodes with shorter paths have executed. We cache this computation so it is only run once for a given topology, set of failures, and set of sources.

For BGP, the detection algorithm performs the following computation: For each node which is enabled to update its best path, it checks whether there exists a pending update that would never get replaced, because the update would definitely be preferred over other updates that are enabled now or may be in the future. To check this, we follow the node's BGP decision process, so if the update is tied for most-preferred in one step it moves to the next. For each step of the decision process, the preference calculation is quite conservative. For local pref, it marks an update as the winner if it matches an import filter that explicitly gives it the highest local pref among all import filters. For AS Path, the path length of the current update must be the minimum possible in the topology. For IGP cost, the update must be from the peer with minimum cost. If at any node, any update is found to be a clear winner, the node is picked as a deterministic node, and is allowed to process the update. If no node is found that has a clear winner but there is a node that has ≥ 2 updates tied for the most preferred, then we deterministically pick any one such node and have SPIN non-deterministically choose which of the multiple updates to process. Figure 6 illustrates these scenarios on a BGP network, highlighting one sequence of node selections (out of many possible).

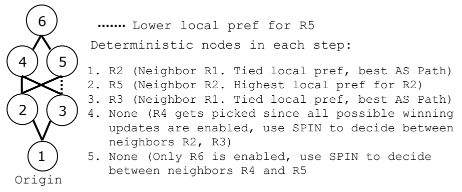

6
4
5
2
3
1
Origin
Lower local pref for R5
Deterministic nodes in each step:
1. R2 (Neighbor R1. Tied local pref, best AS Path)
2. R5 (Neighbor R2. Highest local pref for R2)
3. R3 (Neighbor R1. Tied local pref, best AS Path)
4. None (R4 gets picked since all possible winning updates are enabled, use SPIN to decide between neighbors R2, R3)
5. None (Only R6 is enabled, use SPIN to decide between neighbors R4 and R5

Figure 6: Step-by-step choice of deterministic nodes (Each node has a different AS number).

The detection algorithm may fail to detect some deterministic nodes. For instance, suppose node N is deterministic but its import filter from neighbor M sets the highest local pref for updates with a particular community attribute, and M can never assign that attribute. Then the detection algorithm will fail to mark N as deterministic. But successfully identifying *at least one* deterministic node in a step will avoid non-deterministic branching at that step. As long as this happens frequently, the optimization will be helpful.

Even if the decision on a node is ambiguous in a particular state, the system will often make progress to a state where ambiguities can be resolved. In the example above, once M selects a path (and therefore will never change its path as described in § 4.1.1), the detection algorithm no longer needs to account for a possible more-preferred update from it, and may then be able to conclude that N is deterministic.

4.1.3 Decision independence

If node A's actions are independent of any future decisions of node B and vice versa, then the execution ordering between A and B does not matter. We check a sufficient condition for independence: any route advertisements passed between these nodes, in either direction, must pass through a node that has already made its best path decision (and therefore will not transmit any further updates). In this case, we pick a single arbitrary execution order between A and B.

4.1.4 Failure ordering

As stated in § 3.4.2, the model checker performs all topology changes before the protocol starts execution. We also enforce a strict ordering of link failures, reducing branching even further.

4.2 Policy-based Pruning

Policy-based pruning limits protocol execution to those parts of the network that are relevant to the satisfaction/failure of the policy. When a policy defines a set of source nodes (§ 3.5), it indicates that the policy can be checked by analyzing the forwarding from those nodes only. The best example for this is reachability, which is checked from a particular set of starting nodes. When an execution reaches a state where all source nodes have made their best-path decision, Plankton considers the execution, which is assumed to be consistent, to have finished. In the cases where only a single prefix is defined in a PEC, Plankton performs a more aggressive pruning, based on the influence relation. Any device that cannot influence a source node is not allowed to execute. With some additional bookkeeping, the optimization can be extended to cases where multiple prefixes contribute to a PEC, but our current implementation does not support this. The optimization is also not sound when applied to PECs on which other PECs depend. A router that does not speak BGP may not directly influence a source node, but it may influence the routing for the router IP addresses, which in turn may affect the chosen path of the source node. So, the optimization is not applied in such cases.

4.3 Choice of Failures

In addition to the total ordering of failures described in § 4.1.4, Plankton also attempts to reduce the number of failures that are explored, using equivalence partitioning of devices as proposed by Bonsai [2]. Bonsai groups devices in the network into abstract nodes, creating a smaller topology overall for verification. Plankton computes Device Equivalence Classes (DECs) similarly, and defines a Link Equivalence Class (LEC) as the set of links between two DECs. For each link failure, Plankton then explores only one representative from each LEC. When exploring multiple failures, we refine the DECs and LECs after each selection. Note that this optimization limits the choice of failed links, but the verification happens on the original input network. In order to avoid remapping interesting nodes (§ 3.5), they are each assigned to a separate DEC. Since the computed DECs can be different for each PEC, this optimization is done only when there are no cross-PEC dependencies.

4.4 State Hashing

During the exhaustive exploration of the state space, the explicit state model checker needs to track a large number of states simultaneously. A single network state consists of a copy of all the protocol-specific state variables at all the devices. Maintaining millions of copies of these variables is naturally expensive, and in fact, unnecessary. A routing decision at one device doesn't immediately affect the variables at any of the other devices. Plankton leverages this property to the reduce memory footprint, storing routing table entries as 64-bit pointers to the actual entry, with each entry stored once and indexed in a hash table. We believe this optimization can be applied to other variables in the network state too, as long as they are not updated frequently. Picking the right variables to optimize this way, and developing more advanced hash-based schemes, can be explored in the future.

5 Evaluation

We prototyped Plankton including the equivalence class computation, control plane model, policy API and optimizations in 373 lines of Promela and 4870 lines of C++, excluding the SPIN model checker. We experimented with our prototype on Ubuntu 16.04 running on a 3.4 GHz Intel Xeon processor with 32 hardware threads and 188 GB RAM.

We begin our evaluation with simple hand-created topologies incorporating protocol characteristics such as shortest path routing, non-deterministic protocol convergence, redistribution, recursive routing, etc. Among these tests, we incorporated examples of non-deterministic protocol execution from [8], as well as BGP wedgies, which are policy violations which can occur only under some non-deterministic execution paths. In each of these cases, Plankton correctly finds any violations that are present.

Having tested basic correctness, we next evaluate performance and compare to past verifiers. What sets Plankton apart from tools other than Minesweeper is its higher data plane coverage and ability to handle multiple protocols (Figure 1). We therefore compare primarily with Minesweeper but also include ARC in some tests.

We also evaluated Bonsai [2], a preprocessing technique that helps improve the scalability of configuration verification, for specific policies. Bonsai could assist any configuration verifier. We integrated Bonsai with Plankton, and experimentally compare the performance of Bonsai+Minesweeper and Bonsai+Plankton. However, it is important to study the performance of these tools without Bonsai too: Bonsai's network compression cannot be applied if the correctness is to be evaluated under link failures, or if the policy being evaluated is not preserved by Bonsai.

Experiments with synthetic configurations
Our first set of performance tests uses fat trees. We construct fat trees of increasing sizes, with each edge switch originating a prefix into OSPF. Link weights are identical. We check these networks for routing loops. In order to cause violations, we install static routes at the core routers. In our first set of experiments, the static routes match the routes that OSPF would eventually compute, so there are no loops. Then, we change the static routes such that some of the traffic falls into a routing loop. Figure 7(a) illustrates the time and memory consumed, using Plankton running on various numbers of cores, and using Minesweeper. We observed that under default settings, Minesweeper's CPU utilization keeps changing, ranging from 100% to $1,600\%$. In this experiment and all others where we run both Minesweeper and Plankton, the two tools produced the same policy verification results. This serves as an additional correctness check for Plankton. Bonsai is not used, because its currently available implementation does not appear to support loop policies.

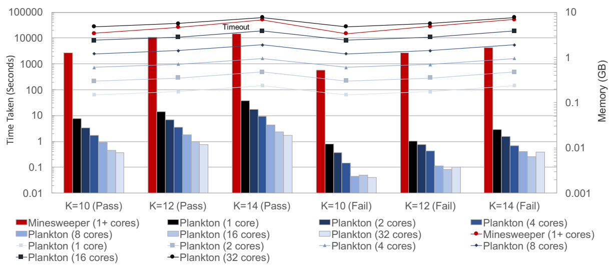
As the results show, Plankton scales well with input network size. The speed and memory consumption varies as expected with the degree of parallelism. Even on a single core, Plankton is quicker than Minesweeper for all topologies. For larger networks, Plankton is several orders of magnitude quicker. On the memory front, even on 16 cores, Plankton's footprint is smaller than Minesweeper's.

Encouraged by the good performance numbers, we scale up to very large fat trees (Figure 7(b)). Here, Minesweeper doesn't finish even in 4 hours, even with a 500-device network (in the case of passing loop check, even in a 245-device network). So, we did not run Minesweeper on the larger networks. We run Plankton with a single CPU core only, to illustrate its time-memory tradeoff: since the analyses of individual PECs are fully independent and of identical computational effort, running with n cores would reduce the time by $n\times$, and increase memory by $n\times$. For example, in the 2,205-device network, Plankton uses about 170 GB per process. Policies that check a single equivalence class are much cheaper: for example, single-IP reachability finishes in seconds or minutes even on the largest networks (Figure 7(b)).

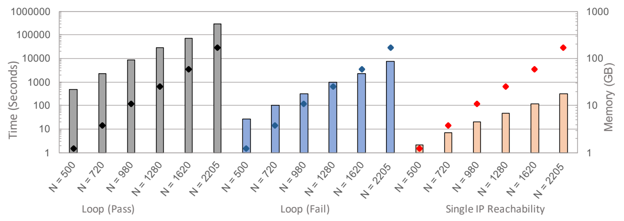
Next, we test Plankton with a very high degree of non-determinism. We evaluated a data center setting with BGP, which is often employed to provide layer 3 routing down to the rack level in modern data centers [17]. We configure BGP as described in RFC 7938 [17] on fat trees of various sizes. Furthermore, we suppose that the network operator intends traffic to pass through any of a set of acceptable *waypoints* on the aggregation layer switches (e.g., imagine the switches implement a certain monitoring function). We pick a random subset of aggregation switches as the waypoints in each experiment. However, we create a "misconfiguration" that prevents multipath and fails to steer routes through the waypoints[‡]. Thus, in this scenario, whether the selected path passes through a waypoint depends on the order in which updates are received at various nodes, due to age-based tie breaking [16]. We check waypoint policies which state that the path between two edge switches should pass through one of the waypoints. Plankton evaluates various non-deterministic convergence paths in the network, and determines a violating sequence of events. Time and memory both depend somewhat on the chosen set of aggregation switches, but even the worst-case times are less than 2 seconds (Figure 7(c)). We consider this a success of our policy-based pruning optimization: the network has too many converged states to be verified in reasonable time, but many have equivalent results in terms of the policy.

Experiments with semi-synthetic configurations
We use real-world AS topologies and associated OSPF link weights obtained from RocketFuel [24]. We pick a random ingress point that has more than one link incident on it. We verify that with any single link failure, all destination prefixes are reachable from that ingress. Here, Minesweeper's SMT-based search algorithm could be beneficial, due to the large search space created by failures. Nevertheless, Plankton performs consistently better in both time and memory (Figure 7(d)). Both tools find a violation in each case. The time taken by Plankton with 16 and 32 cores are often identical, since a violation is found in the first set of PECs. Note

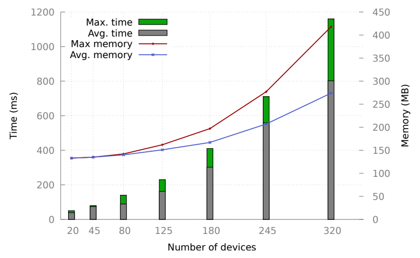
[‡]This setup is convenient for practical reasons, as our current Plankton prototype implementation does not support BGP multipath.



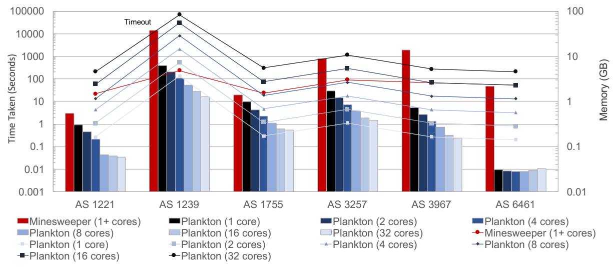
(a) Fat trees with OSPF, loop policy, multi-core



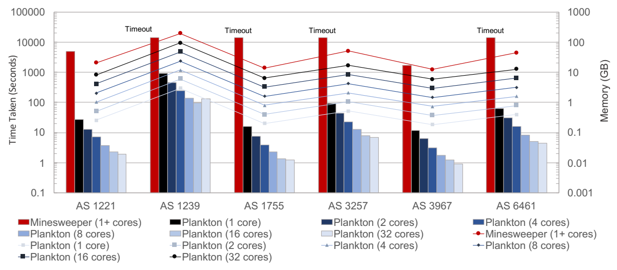
(b) Fat trees with OSPF, multiple policies, 1 core



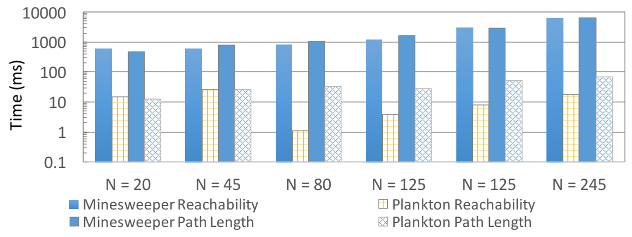
(c) Fat trees with BGP, waypoint policy, 1 core



(d) AS topologies with OSPF and failures, reachability policy, multi-core



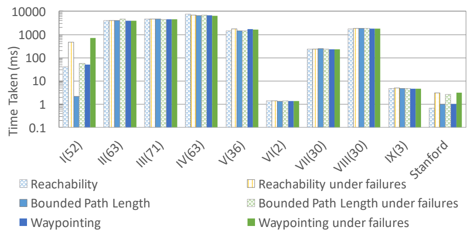
(e) AS topologies with iBGP over OSPF, reachability policy, multi-core



(f) Bonsai-compressed fat trees with OSPF, multiple policies, 8 cores

Network	Links Failed	ARC	Plankton
Fat tree (20 nodes)	0	1.08 s	0.19 s
	≤ 1	1.05 s	0.23 s
	≤ 2	1.00 s	0.31 s
Fat tree (45 nodes)	0	12.17 s	0.49 s
	≤ 1	12.40 s	0.55 s
	≤ 2	12.49 s	2.09 s
Fat tree (80 nodes)	0	300.50 s	0.93 s
	≤ 1	280.00 s	1.98 s
	≤ 2	294.12 s	18.30 s
Fat tree (125 nodes)	0	4847.57 s	1.90 s
	≤ 1	5096.96 s	9.34 s
	≤ 2	4955.95 s	159.37 s
AS 1221 (108 nodes)	0	41.62 s	1.12 s
	≤ 1	40.50 s	2.80 s
	≤ 2	38.83 s	106.57 s
AS 1775 (87 nodes)	0	49.32 s	1.02 s
	≤ 1	48.65 s	2.73 s
	≤ 2	46.52 s	155.28 s

(g) Networks with link failures, all-to-all reachability policy, 8 cores



(h) Real-world configs (number of devices in parentheses), multiple policies, 1 core

Network	Policy	Links Failed	Memory	Time
II	Loop	0	2.37 GB	8.79 s
		≤ 1	2.47 GB	13.62 s
	Multipath Consistency	0	2.37 GB	16.28 s
		≤ 1	2.37 GB	22.01 s
	Path Consistency	0	2.37 GB	15.43 s
		≤ 1	2.37 GB	23.55 s
III	Loop	0	2.36 GB	11.49 s
		≤ 1	2.88 GB	29.81 s
	Multipath Consistency	0	2.37 GB	16.33 s
		≤ 1	2.67 GB	24.86 s
	Path Consistency	0	2.36 GB	15.53 s
		≤ 1	2.88 GB	21.01 s
IV	Loop	0	2.31 GB	12.37 s
		≤ 1	2.40 GB	13.14 s
	Multipath Consistency	0	2.34 GB	16.36 s
		≤ 1	2.37 GB	17.04 s
	Path Consistency	0	2.33 GB	16.33 s
		≤ 1	2.37 GB	17.00 s

(i) Real-world configs, multiple policies, 32 cores

Figure 7: Plankton experiments. Bars and lines/points denote time and memory consumption, respectively.

that in this experiment and the next, we did not use Bonsai, because (i) it cannot be used for checks involving link failures, and (ii) the topology has hardly any symmetry that Bonsai could exploit.

To evaluate our handling of PEC dependencies, we configure iBGP over OSPF on the AS topologies. The iBGP prefixes rely on the underlying OSPF routing process to reach the next hop. We check that packets destined to the iBGP-announced prefixes are correctly delivered. It is worth noting that this test evaluates a feature that, to the best of our knowledge, is provided only by Plankton and Minesweeper. Thanks to the dependency-aware scheduler, Plankton performs multiple orders of magnitude better (Figure 7(e)). This is unsurprising: Minesweeper's approach of duplicating the network forces it to solve a *much* harder problem here, sometimes over $300\times$ larger.

Integration with Bonsai

We integrated Plankton with Bonsai to take advantage of control plane compression when permitted by the specific verification task at hand. We test this integration experimentally by checking Bounded Path Length and Reachability policies on fat trees running OSPF. The symmetric nature of fat trees is key for Bonsai to have a significant impact. We measure the time taken by Plankton and Minesweeper, *after* Bonsai preprocesses the network. Plankton still outperforms Minesweeper by multiple orders of magnitude (Figure 7(f)).

Comparison with ARC

Having evaluated Plankton's performance in comparison with Minesweeper, we move on to comparing the performance of Plankton and ARC. ARC is specifically designed to check shortest-path routing under failures, so we expected the performance to be much better than the more general-purpose Plankton, when checking networks compatible with ARC. We check all-to-all reachability in fat trees and AS topologies running OSPF, under a maximum of 0, 1 and 2 link failures. Similar to Minesweeper, ARC's CPU utilization ranges from 100% to 600% under default settings. We allocate 8 cores to Plankton. Plankton is multiple orders of magnitude faster in most cases (Figure 7(g)).[§] This is genuinely surprising; one reason that *may* explain the observation is that ARC always computes separate models for each source-destination pair, whereas Plankton computes them based only the destination, when verifying destination address routing. Nevertheless, we do not believe that there is a fundamental limitation in ARC's design that would prevent it from outperforming Plankton on the networks that can be checked by either tool. Interestingly, while ARC's resiliency-focused algorithm doesn't scale as easily as Plankton for larger networks, its performance actually sometimes slightly *improves* when the number of failures to be checked increases. Plankton on the other hand scales poorly when checking increasing levels of resiliency. We do not find this concerning, since most interesting checks in the real world involve only a small number of failures. When we performed these experiments with Minesweeper, no check involving 2 failures ran to completion except the smallest fat tree.

Testing with real configurations

We used Plankton to verify 10 different real-world configurations from 3 different organizations, including the publicly available Stanford dataset. We first check reachability, waypointing and bounded path length policies on these networks, with and without failures. All except one of these networks use some form of recursive routing, such as indirect static routes or iBGP. We feel that this highlights the significance of Plankton's and Minesweeper's support for such configurations. Moreover, the PEC dependency graph for these networks did not have any strongly connected components larger than a single PEC, which matches yet another of our expectations. Interestingly, we did find that the PEC dependency graph had *self loops*, with a static route pointing to a next hop IP within the prefix being matched. It is also noteworthy that in these experiments, the only non-determinism was in the choice of links that failed, which substantiates our argument that network configurations in the real world are largely deterministic. Figure 7(h) illustrates the results, which indicate that Plankton can handle the complexity of real-world configuration verification.

In our next experiment with real world configs, we identify three networks where Loop, Multipath Consistency and Path Consistency policies are meaningful and non-trivial to check. We check these policies with and without link failures. Figure 7(i) illustrates the results of this experiment. The results indicate that the breadth of Plankton's policies scale well on real world networks. The Batfish parser, which is used by Minesweeper, was incompatible with the configurations, so we could not check these configs on Minesweeper (checking the Stanford dataset failed *after* parsing). However, the numbers we observe are significantly better than those reported for Minesweeper on similar-sized networks, for similar policies.

[§]Our numbers for ARC are similar to those reported by its authors for similar sized networks, so we believe we have not misconfigured ARC.

Experiment	Optimizations	Time	Memory
Ring, OSPF, 4 nodes, 1 failure	All	343 μs	137.43 MB
	None	1.56 ms	137.39 MB
Ring, OSPF, 8 nodes, 1 Failure	All	623 μs	143.22 MB
	None	0.13 s	137.04 MB
Ring, OSPF, 16 nodes, 1 Failure	All	2.44 ms	137.89 MB
	None	266.48 s	7615.57 MB
Fat tree, OSPF, 20 nodes	All	464 μs	551.73 MB
	None	> 5 min	> 8983.55 MB
Fat tree, OSPF, 245 nodes	All	4.297 s	1908 MB
	All but link failure opt.	64.97 s	72862 MB
AS 1221 iBGP	All	27.54 s	254.22 MB
	All but deterministic node opt.	25.43 s	254.34 MB
Fat tree, BGP, 20 nodes	All	46 ms	137 MB
	All but deterministic node opt.	> 5 min	> 6144 MB
	All but policy-based pruning	> 5 min	> 6144 MB

Figure 8: Experiments with optimizations disabled/limited

Optimization Cost/Effectiveness

To determine the effectiveness of Plankton's optimizations, we perform experiments with some optimizations disabled or limited. Figure 8 illustrates the results from these experiments. When all optimizations are turned off, naive model checking fails to scale beyond the most trivial of networks. The optimizations reduce the state space by $4.95\times$ in smaller networks and by as much 24,968$\times$ in larger ones.

To evaluate device-equivalence based optimizations in picking failed links, we perform loop check on fat trees running OSPF under single link failure with the optimization turned off. We observed a $15\times$ reduction in speed, and a $38\times$ increase in memory overhead, indicating the effectiveness of the optimization in networks with high symmetry.

Experiment	No Bitstate Hashing	Bitstate Hashing
180 Node BGP DC Waypoint (Worst Case)	202 MB	67 MB
320 Node BGP DC Waypoint (Worst Case)	428 MB	215 MB
AS 1239 Fault Tolerance (2 cores)	7.33 GB	4.52 GB
AS 1221 Fault Tolerance (1 core)	163.53 MB	60 MB

Figure 9: The effect of bitstate hashing on memory usage

In the next set of experiments, we measure the impact of our partial order reduction technique of prioritizing deterministic nodes (§ 4.1.2). We first try the iBGP reachability experiment with the AS 1221 topology, with the detection of deterministic nodes in BGP disabled. We notice that in this case the decision independence partial order reduction produces reductions identical to the disabled optimization, keeping the overall performance unaffected. In fact, the the time improves by a small percentage, since there is no detection algorithm that runs at every step. We see similar results when we disable the optimization on the edge switches in our BGP data center example. However, this does not mean that the deterministic node detection can be discarded — in the BGP data center example, when the optimization is disabled altogether, the performance falls dramatically. The next optimization that we study is policy-based pruning. On the BGP data center example, we attempt to check a waypoint policy, with policy-based optimizations turned off. The check times out, since it is forced to generate every converged data plane, not just the ones relevant to the policy.

SPIN provides a built-in optimization called *bitstate hashing* that uses a Bloom filter to keep track of explored states, rather than storing them explicitly. This can cause some false negatives due to reduced coverage of execution paths. We find that bit state hashing provides significant reduction in memory in a variety of our test cases (Figure 9). According to SPIN's statistics our coverage would be over 99.9%. Nevertheless, we have not turned on bitstate hashing in our other experiments in favor of full correctness.

6 Limitations

Some of the limitations of Plankton, such as the lack of support for BGP multipath and limited support for route aggregation, have been mentioned in previous sections. As discussed in § 3.2, Plankton may also produce false positives when checking networks with cross-PEC dependencies, because it expects that every converged state of a PEC may coexist with every converged state of other PECs that depend on it. However, such false positives are unlikely to happen in practice, since real-world cases of cross-PEC dependencies (such as iBGP) usually involve only a single converged state for the recursive PECs. Our current implementation of Plankton assumes full visibility of the system to be verified, and that any dynamics will originate from inside the system. So, influences such as external advertisements need to be modeled using stubs that denote entities which originate them. Plankton's technique is not suited for detecting issues in vendor-specific protocol implementations, a limitation that all existing formal configuration analysis tools share. As with most formal verification tools, one needs to assume that Plankton itself is correct, both in terms of the theoretical foundations as well as the implementation. Correct-by-construction program synthesis could help in this regard.

7 Related Work

Data plane verification: The earlier offline network verification techniques [19, 13] have evolved into more efficient and real-time ones (e.g., [15, 12, 10, 26, 3, 11]), including richer data plane models (e.g., [21, 14]). These techniques however, cannot verify configurations prior to deployment.

Configuration verification: We discussed the state of the art of configuration verification in § 2, and how Plankton improves upon the various tools in existence. CrystalNet [18] emulates actual device VMs, and its results could be fed to a data plane verifier. However, this would not verify non-deterministic control plane dynamics. Simultaneously improving the fidelity of configuration verifiers in *both* dimensions (capturing dynamics as in Plankton and implementation-specific behavior as in CrystalNet) appears to be a difficult open problem.

Optimizing network verification: Libra [27] is a divide-and-conquer data plane verifier, which is related to our equivalence class-based partitioning of possible packets. The use of symmetry to scale verification has been studied in the data plane [22] and control plane (Bonsai [2]). We have discussed how Plankton uses ideas similar to Bonsai, as well as integrates with Bonsai itself.

Model checking in the networking domain: Past approaches that used model checking in the networking domain have focused almost exclusively on the network software itself, either as SDN controllers, or protocol implementations [4, 23, 20]. Plankton uses model checking not to verify software, but to verify configurations.

8 Conclusion and Future Work

We described Plankton, a formal network configuration verification tool that combines equivalence partitioning of the header space with explicit state model checking of protocol execution. Thanks to pragmatic optimizations such as partial order reduction and state hashing, Plankton produces significant performance gains over the state of the art in configuration verification. Improvements such as checking transient states, incorporating real software, partial order reduction heuristics that guarantee reduction, etc. are interesting avenues of future work.

Acknowledgements: We thank the anonymous reviewers and our shepherd Cole Schlesinger for their comments. We also thank Ratul Mahajan and Ryan Beckett for their help in integrating Bonsai with Plankton. This work was supported by NSF grants CNS-1513906 and CCF-1421575.

References

[1] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 155–168.

[2] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. Control plane compression. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 476–489.

[3] Bjørner, N., Juniwal, G., Mahajan, R., Seshia, S. A., and Varghese, G. ddnf: An efficient data structure for header spaces. In *Haifa Verification Conference* (2016), Springer, pp. 49–64.

[4] Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. A NICE way to test openflow applications. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 127–140.

[5] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., and Varghese, G. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), pp. 217–232.

[6] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. A general approach to network configuration analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 469–483.

[7] Gember-Jacobson, A., Viswanathan, R., Akella, A., and Mahajan, R. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), SIGCOMM '16.

[8] Griffin, T. G., Shepherd, F. B., and Wilfong, G. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw. 10*, 2 (Apr. 2002), 232–243.

[9] Holzmann, G. J. The model checker spin. *IEEE Trans. Softw. Eng. 23*, 5 (May 1997), 279–295.

[10] Horn, A., Kheradmand, A., and Prasad, M. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 735–749.

[11] Horn, A., Kheradmand, A., and Prasad, M. R. A precise and expressive lattice-theoretical framework for efficient network verification. In *2019 27st IEEE International Conference on Network Protocols (ICNP)* (2019), IEEE.

[12] Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S. 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, 2013).

[13] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 113–126.

[14] Kheradmand, A., and Rosu, G. P4K: a formal semantics of P4 and applications. *CoRR abs/1804.01468* (2018).

[15] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying Network-wide Invariants in Real Time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), NSDI'13, USENIX Association, pp. 15–28.

[16] Lapukhov, P. Equal-Cost Multipath Considerations for BGP. Internet-Draft draft-lapukhov-bgp-ecmp-considerations-00, Internet Engineering Task Force, Oct. 2016. Work in Progress.

[17] Lapukhov, P., Premji, A., and Mitchell, J. Use of BGP for Routing in Large-Scale Data Centers. RFC 7938 (Informational), Aug. 2016.

[18] Liu, H. H., Zhu, Y., Padhye, J., Cao, J., Tallapragada, S., Lopes, N. P., Rybalchenko, A., Lu, G., and Yuan, L. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, ACM, pp. 599–613.

[19] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference* (New York, NY, USA, 2011), SIGCOMM '11, pp. 290–301.

[20] Musuvathi, M., and Engler, D. R. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1* (Berkeley, CA, USA, 2004), NSDI'04, USENIX Association, pp. 12–12.

[21] Panda, A., Lahav, O., Argyraki, K., Sagiv, M., and Shenker, S. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 699–718.

[22] Plotkin, G. D., Bjørner, N., Lopes, N. P., Rybalchenko, A., and Varghese, G. Scaling network verification using symmetry and surgery. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2016), POPL '16, ACM, pp. 69–83.

[23] Sethi, D., Narayana, S., and Malik, S. Abstractions for model checking SDN controllers. In *2013 Formal Methods in Computer-Aided Design* (oct 2013), IEEE.

[24] Spring, N., Mahajan, R., Wetherall, D., and Anderson, T. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw. 12*, 1 (Feb. 2004), 2–16.

[25] Weitz, K., Woos, D., Torlak, E., Ernst, M. D., Krishnamurthy, A., and Tatlock, Z. 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* (2016), OOPSLA 2016, ACM.

[26] Yang, H., and Lam, S. S. Real-time verification of network properties using atomic predicates. In *2013 21st IEEE International Conference on Network Protocols (ICNP)* (2013), IEEE, pp. 1–11.

[27] Zeng, H., Zhang, S., Ye, F., Jeyakumar, V., Ju, M., Liu, J., McKeown, N., and Vahdat, A. Libra: Divide and conquer to verify forwarding tables in huge networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, 2014), USENIX Association, pp. 87–99.

A Extended SPVP

In extended SPVP, for each node n, and for each peer $n' \in \text{peers}(n)$, $\text{rib-in}_n(n')$ keeps track of the most recent advertisement of n' to n. In addition, $\text{best-path}(n)$ keeps the best path that n has to one of the *multiple* origins. Peers are connected using reliable FIFO message buffers to exchange advertisements. Each advertisement consists of a path from the advertising node to an origin. In each step (which we assume is performed atomically) a node n takes an advertisement

p from the buffer connected to peer n', and applies an import filter on it ($\text{import}_{n,n'}(p)$). n then updates $\text{rib-in}_n(n')$ with the new imported advertisement. In our extension, we assume that each node has a ranking function λ that provides a *partial order* over the paths acceptable by the node. n then proceeds by updating its best-path to the highest ranking path in rib-in_n. If the best path in rib-in_n have the same rank as the current best path and that path is still valid, $\text{best-path}(n)$ will not change. If the best path is updated, n advertises the path to its peers. For each peer n', n applies the export filter on the path ($\text{export}_{n,n'}(\text{best-path}(n))$) before sending the advertisement.

The import filter, the export filter, and the ranking functions are abstract notions that will be inferred from the configuration of the node. We make reasonable assumptions about these notions (Appendix B). Attributes such as local pref, IGP cost, etc. are accounted for in the the ranking function and the import/export filters.

If a session between two peers fails, the messages in the buffer are lost and the buffer cannot be used anymore. We assume that when this happens, each peer gets $\perp$ as the advertised path. Additionally, to be able to model iBGP, in extended SPVP we allow the ranking function of any node n to change at any time during the execution of the protocol. This is to model cases in which for example a link failure causes IGP costs to change. In such cases we assume that n receives a special message to recompute its best-path according to the new ranking function.

The state of network at each point in time consists of the values of best-path, rib-in, and the contents of the message buffers. In the initial state S_0, the best path of the origins is ε and the best path of the rest of the nodes are $\perp$ which indicates that the node has no path. Also for any $n, n' \in V$, $\text{rib-in}_n(n')$ is $\perp$. We assume that initially the origins put their advertisements in the message buffer to their peers, but the rest of the buffers are empty.

An (partial) execution of SPVP is a sequence of states π which starts from S_0 and each state is reachable by a single atomic step of SPVP on the state before it. A converged state in SPVP, is a state in which all buffers are empty. A complete execution is an execution that ends in a converged state. It is well known that there are configurations which can make SPVP diverge in some or all execution paths. However, our goal is to only to check the forwarding behavior in the converged states, through explicit-state model checking. So, we define a much simpler model that can be used, without compromising the soundness or completeness of the analysis (compared to SPVP).

B Assumptions

We make the following assumptions in in our theoretical model.

- Both import/export filters return $\perp$ if the filter rejects the advertisement according to the configuration.
- All import filters reject paths that cause forwarding loops. They also do not alter the path (unless the path is rejected).
- All export filters for each node n not rejecting an advertisement, will append n at the end of the advertised path. No other modification is made to the path.
- Path $\perp$ has the lowest ranking in all ranking functions.
- The import/export filters never change during the execution of the protocol. Note that we do not make such assumption for the ranking functions.

Note that these are reasonable assumptions with respect to how real world protocols (especially BGP) work.

C Proof of Theorems

Proof of Theorem 1. For a complete execution $\pi = S_0, S_1, ..., S_c$ of SPVP and a state S_i in that execution, for any node n, we say that n is converged in S_i for execution π iff n has already picked the path it has in the converged state (S_c) and does not change it:

$$\text{converged}_\pi(n, S_i) \triangleq \\ \forall j. i \le j \le c : \text{best-path}_{S_j}(n) = \text{best-path}_{S_c}(n)$$

It it clear that when a node converges in an state, it remains converged (according to the definition above).

Lemma 1. *In any complete execution $\pi = S_0, S_1, ..., S_c$ of SPVP, for any sate S_i, for any two nodes n and n', if $\exists l : n' = \text{best-path}_{S_i}(n)[l]$* [¶] *, and $\text{best-path}_{S_i}(n') \neq \text{best-path}_{S_i}(n)[l :]$, there is a j ($i < j \le c$) such that $\text{best-path}_{S_j}(n) \neq \text{best-path}_{S_i}(n)$.*

Proof. This can be shown by a simple induction on the length of the prefix of the best path of n from n to up to n' (l). If $l = 1$ (i.e the two nodes are directly connected) then either n' will advertise its path to n and n and will change its path or the link between n and n' fails in which case n will receive an advertisement with $\perp$ as the path (Section 3.4.1), which causes n to change its path. Note that the argument holds even if the ranking function of n or n' changes. If $l > 1$, assuming the claim holds for lengths less than l, for $n'' = \text{best-path}_{S_i}(n)[0]$, either $\text{best-path}_{S_i}(n'') \neq \text{best-path}_{S_i}(n)[1 :]$ in which case due to induction hypothesis the claim holds, or $\text{best-path}_{S_i}(n'') = \text{best-path}_{S_i}(n)[1 :]$ in which case we note that $n' = \text{best-path}_{S_i}(n'')[l-1]$, and since the length of the path n'' to n' is less than l, by induction hypothesis we know that eventually (i.e for a $j > i$) $\text{best-path}_{S_j}(n'') \neq \text{best-path}_{S_j}(n)[1 :]$ and by induction hypothesis this will lead to a change in the best path of n.

[¶]For a path $P = p_0, p_1, ..., p_n$, we denote p_i by $P[i]$ and $p_i, p_{i+1}, ..., p_n$ by $P[i :]$.

□

Corollary 1. *For any complete execution π of SPVP , for any node n, any node along the best path of n in the converged state converges before n.*

Now consider a complete execution $\pi = S_0, S_1, ..., S_c$ of SPVP. We will construct a complete execution of RPVP with $|N|$ steps (where N is set of all nodes) resulting in the same converged state as S_c. We start with a topology in which all the links that have failed during the execution of SPVP are already failed. For any node n, we define $C_\pi(n) = min\{i|\text{converged}_\pi(n, S_i)\}$. Consider the sequence $n_1, n_2, ..., n_{|N|}$ of all nodes sorted in the increasing order of C_π. Now consider the execution of RPVP $\pi' = S'_0, S'_1, ..., S'_{|N|}$ which starts from the initial state of RPVP and in each state S'_i, (a) either node n_{i+1} is the picked enabled node and the node $p_i = \text{best-path}_{S_c}(n_{i+1})[0]$ is the picked best peer, or (b) in case $p_i = \perp$, nothing happens and $S'_{i+1} = S'_i$.

First, note that (modulo the repeated states in case *b*), π' is a valid execution of of RPVP: at each state S'_i (in case *a*), n_{i+1} is indeed an enabled node since its best path at that state is $\perp$ and according to corollary 1, p_i has already picked its path: Also p_i will be in the set of best peers of n_{i+1} (line 13 in RPVP). Assume this is not the case, i.e there exists another peer p' that can advertise a better path. This means that in S_c of SPVP, p' can send an advertisement that is better (according to the version of ranking functions in S_c) than the converged path of n_{i+1}. This contradicts the fact that S_c is a converged path.

Second, note that $S'_{|N|}$ is a converged state for RPVP, because otherwise, using similar reasoning as above, S_c can not be converged. Also it is easy to see that $\text{best-path}_{S'_{|N|}} = \text{best-path}_{S_c}$

Finally, note that in π', once a node changes its best path from $\perp$, it does not change its best path again.

□

Proof of Theorem 2. We begin by making two observations about RPVP that are key to the proof:

- RPVP for a prefix can never converge to a state having looping paths.
- If a node u adopts the best path of a neighbor v, v will be next hop of u.

Consider any converged state S. The theorem states that any partial execution that is consistent with S can be extended to a full execution that leads to S. We prove the theorem by induction on the length of the longest best path in S.

Base case: If in a network a converged state exists where the best path at each node is of length 0, that means that each node is either an origin or doesn't have a best path for the prefix. Since any execution apart from the empty execution (where no protocol event happens) is not consistent with this state, the theorem holds.

Induction hypothesis: If a converged state exists in a network such that all best paths are of length k or less, then any partial execution that is consistent with the converged state can be extended to a full execution that reaches the converged state.

Induction step: Consider a network with a converged state S such that the longest best path is of length $k + 1$. We first divide the nodes in the network into two classes — N, which are the nodes with best paths of length k or less, and N', which are nodes with best paths of length $k + 1$. Consider a partial execution π that is consistent with S. We identify two possibilities for π:

Case 1: Every node that has executed in π falls into N. In this case, we define a smaller network which is the subgraph of the original network, induced by the nodes in N. In this network, the path selections made in S will constitute a converged state. This is because in the original network, in the state S, the nodes in N are not enabled to make state changes. So, we can extend π such that we get an execution π' where nodes in N match the path selections in S. Now, we further extend π' with steps where each node in N' reads the best path from the node that is its nexthop in S and updates its best path. When every node in N has done this, the overall system state will reach S.

Case 2: At least one node in N' has executed in π. In this case, we observe that since π is consistent with S, by the definition of a consistent execution, no node in the network has read the state of any node in N'. So, we can construct an execution π' which has the same steps as π, except that any step taken by a node in N' is skipped. As in the previous case, π' can be extended to reach a converged state in the subgraph induced by N. We extend π, first by using the steps that extend π', and if necessary, taking additional steps at nodes from N' to reach S. □