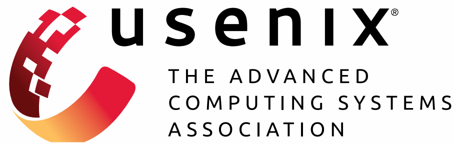


Config2Spec: Mining Network Specifications from Network Configurations

Rüdiger Birkner, *ETH Zürich;* Dana Drachsler-Cohen, *Technion;*
Laurent Vanbever and Martin Vechev, *ETH Zürich*

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

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

Open access to the Proceedings of the
17th USENIX Symposium on Networked
Systems Design and Implementation
(NSDI '20) is sponsored by

Config2Spec: Mining Network Specifications from Network Configurations

Rüdiger Birkner[1] Dana Drachsler-Cohen[2*] Laurent Vanbever[1] Martin Vechev[1]

[1]*ETH Zürich* [2]*Technion*

Abstract

Network verification and configuration synthesis are promising approaches to make networks more reliable and secure by enforcing a set of policies. However, these approaches require a formal and precise description of the intended network behavior, imposing a major barrier to their adoption: network operators are not only reluctant to write formal specifications, but often do not even know what these specifications are.

We present *Config2Spec*, a system that automatically synthesizes a formal specification (a set of policies) of a network given its configuration and a failure model (e.g., up to two link failures). A key technical challenge is to design a synthesis algorithm which can efficiently explore the large space of possible policies. To address this challenge, *Config2Spec* relies on a careful combination of two well-known methods: data plane analysis and control plane verification.

Experimental results show that *Config2Spec* scales to mining specifications of large networks (>150 routers).

1 Introduction

Consider the task of a network operator who—tired of human-induced network downtimes—decides to rely on formal methods to verify her network-wide configurations [4,14,22,30] or to synthesize them automatically [5,9,10,28,29]. The operator quickly realizes that both verifiers and synthesizers require a specification of the correct intended network-wide behavior. A few generic requirements quickly come to mind: surely she wants her network to ensure reachability. At the same time, she realizes that her network does *way* more than just ensuring reachability. Among others, it needs to enforce load balancing for popular destinations, provide isolation between customers, drop traffic for suspicious prefixes, and reroute business traffic via predefined waypoints—all these under failures and over hundreds of devices. Writing the precise specification seems daunting, especially as most of it has been homegrown over years, by a team of network engineers (some of which do not even work there anymore).

This situation illustrates the difficulty of writing network specifications. Akin to software specifications, formal specifications are hard to write (as hard as writing the program in the first place [20]), debug, and modify [2,21]. Yet, without easier ways to provide network specifications, network verification and synthesis are unlikely to get widely deployed.

Config2Spec We introduce *Config2Spec*, a system that automatically mines a network's specification from its configurations and a failure model (e.g., up to k failures). *Config2Spec* is precise: it returns *all* policies that hold under the failure model (no false negatives) and *only* those (no false positives).

Challenges Mining precise network specifications is challenging as it involves exploring two exponential search spaces: *(i)* the space of all possible policies, and *(ii)* the space of all possible network-wide forwarding states. The challenge stems from the fact that individually exploring each of the search spaces can be prohibitive: a search for the true policies is hard since they are a small fraction of the policy space, while a search for the violated policies is hard since these require witnesses (data planes), which are often sparse.

Insights *Config2Spec* addresses the above challenges by combining the strengths of data plane analysis and control plane verification. Data plane analysis enables us to compute the set of policies that hold for a single data plane, thereby providing an efficient way of *pruning* policies. On the other hand, control plane verification is an efficient way of *validating* that a single policy holds for all the data planes. *Config2Spec* combines the two approaches to prune the large space of policies through sampling and data plane analysis and then, to avoid the need of exploring all data planes, validating the remaining policies with control plane verification. The key insight is to dynamically identify the approach providing for better progress. We design predictors which rely on past iterations and the failure model to switch between the two approaches.

*Work done while at ETH Zürich.

Scalability While this approach scales, we identify three domain-specific techniques to improve it even further. First, to better utilize the pruning through data plane analysis, we design a *policy-aware* sampler of data planes. We experimentally show that our approach outperforms a random sampler: with typically fewer samples, it leads to pruning substantially more policies. Second, to reduce the number of queries posed to the verifier, we group queries to the control plane verifier. Third, we analyze the network topology to prune policies that are physically not feasible due to poor connectivity of the routers. For large networks and permissive failure models, this technique makes the difference between *Config2Spec* completing in few hours instead of days.

System We implemented *Config2Spec*, which leverages two state-of-the-art data plane analysis and control plane verification tools, Batfish [13] and Minesweeper [4]. As the implementation relies on these two tools, it is tied to configurations and features supported by them. The approach itself, is not limited to any specific type of configuration.

Config2Spec provides a substantial improvement over baselines that use each of the above tools in isolation (up to 8.3x against the best baseline). Further, *Config2Spec* often mines a precise network specification within an hour, and for large networks (> 150 routers) within 2.7 hours (for OSPF configurations) or 13.7 hours (for BGP configurations). We also illustrate that *Config2Spec* can handle real network configurations by running it successfully on Internet2's configurations.

Contributions Our main contributions are:

- A novel approach to automatically mine the specification of a network by leveraging both data plane analysis and control plane verification (§3).
- A dynamic predictor to decide which approach provides for better progress (§4).
- A policy-aware sampler to find data planes that are likely to prune more policies (§5).
- Policy grouping and topology-based trimming to reduce the number of queries posed to the verifier (§6, §7).
- An end-to-end implementation and an extensive evaluation across different topologies and baselines, showing that *Config2Spec* scales to large networks and significantly outperforms possible baselines (§8).

Novelty Several previous works [6, 7, 31] have looked into mining a network's specification by observing the content of the data plane. All of these works are limited to reachability policies and unlike *Config2Spec*, they either approximate the specification or do not consider the impact of failures on the specification. Concretely, they only produce the network's policies which hold when all links and routers are up. In contrast, *Config2Spec* is able to mine precise network specifications for a given failure model.

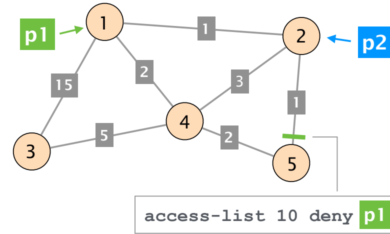


Figure 1: An OSPF network with five routers and two destinations. An ACL at router `5` blocks traffic destined to prefix `p1`, attached to router `1`.

Usefulness In general, the network's specification can be used for many different applications, such as configuration synthesis/verification and network management (e.g., analyzing the effects of configuration changes). Further, having the specification at hand allows network operators to check whether the policies they intend to enforce are indeed enforced.

In multiple discussions, network operators confirmed that a tool like *Config2Spec* is indeed useful. One operator mentioned that the adoption of a new monitoring tool fell through because it required the network's specification to detect flawed configuration changes. Another operator mentioned that having the network's specification at hand would greatly help them better understand their configurations that accumulated over years. Especially, since short-term fixes to problems that need immediate attention (e.g., congestion and hardware problems) are often forgotten and persist even long after the responsible engineer left the company. In addition, the specification can be used to streamline network's configuration by refactoring it, while keeping the same specification.

2 Motivation and Problem Definition

Obtaining a specification for how a network behaves can be useful in a variety of scenarios beyond network verification and synthesis, including helping the operator identify unexpected behaviors and inconsistencies, as well as enabling a smoother transition to (updated) configurations upon new requirements. To define the problem of mining specifications, we rely on two concepts: a *network specification*, composed of a set of *policies*, and a *failure model*, specifying under which failures the network specification should hold. We next define these concepts and illustrate them on a running example. Then, we introduce the network specification mining problem and discuss several baseline approaches together with their shortcomings, thus motivating our solution.

Running example Throughout the paper, we refer to the example shown in Fig. 1. Here, we have a network that consists of five routers and seven links. There are two host networks, `p1` and `p2`, attached to routers `1` and `2`. All routers

Policy	Meaning
`reachability(r, p)`	Traffic from `r` can reach `p`.
`isolation(r, p)`	Traffic from `r` is isolated from `p`.
`waypoint(r, w, p)`	Traffic from `r` to `p` passes through `w`.
`loadbalancing(r, p)`	Traffic from `r` to `p` is load balanced on at least two paths.

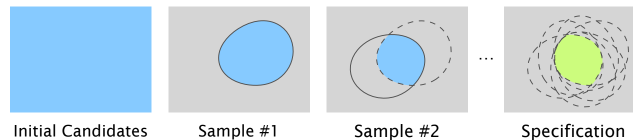
Table 1: Network policies (`r` and `w` are routers, `p` is a prefix).

are in the same OSPF area and the OSPF weights are depicted on the links. An IP access control list (ACL) on the interface from router `5` to `2` drops all packets destined to prefix `p1`.

Failure models A failure model consists of a *symbolic environment* and a number k. The symbolic environment defines which links are up or down, and which links may fail. Technically, a symbolic environment is a partition of the network links L into three subsets L_{up}, L_{down}, and $L_{symbolic}$ (i.e., given L_{up} and L_{down}, we can derive $L_{symbolic} = L \setminus (L_{up} \cup L_{down})$). The number k is a bound on the total number of links which can be simultaneously down. A *concrete environment* is a partition of the network links L into two subsets L_{up} and L_{down}. Namely, all links are fixed to a concrete state: up or down. We say that a failure model with a symbolic environment L^{SE}_{up}, L^{SE}_{down}, $L^{SE}_{symbolic}$ and a bound k, captures a concrete environment with L^{CE}_{up} and L^{CE}_{down} if $L^{SE}_{up} \subseteq L^{CE}_{up}$, $L^{SE}_{down} \subseteq L^{CE}_{down}$, and $|L^{CE}_{down}| \leq k$. Intuitively, a failure model captures all concrete environments for which the links in L^{SE}_{up} are up, the links in L^{SE}_{down} are down, and there are at most k links which are down.

For example, a failure model for our running example is $L_{symbolic} = L$ (i.e., L_{up} and L_{down} are the empty sets) and $k = 1$. This model describes any concrete environment with at most one link failure. There are eight concrete environments which meet this failure model: one where no link is down, and seven in which each of the links fails once. Another failure model is $L_{up} = \{$`2-4`$\}$, $L_{down} = \{$`2-5`$\}$, $L_{symbolic} = L \setminus (L_{up} \cup L_{down})$, and $k = 2$. This model describes any concrete environment whose link between routers `2` and `4` is up, the link between `2` and `5` is down, and the rest may be up or down. Since $k = 2$, another failed link is allowed in addition to `2-5`. There are six concrete environments that meet this failure model.

Network specification and policies A *network specification* consists of a set of policies. A *policy* captures a specific behavior in the network (e.g., reachability of two routers). It is modeled with a predicate (a constraint) which, given a concrete environment, evaluates to true if the policy holds for that concrete environment, and false otherwise. For our running example, the `reachability(5, p2)` policy evaluates to true for the concrete environment in which all links are up, and to false for the concrete environment where all links are down. We say a policy holds for a failure model if it holds for all concrete environments captured by the failure model.



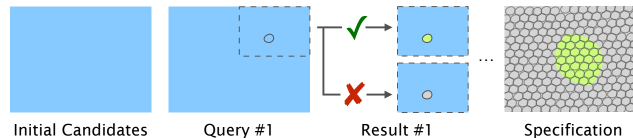


Figure 2: Illustration of the baseline approaches.

For example, the policy `reachability(5, p2)` holds for the failure model $L_{symbolic} = L$ and $k = 1$, but not for $k = 3$.

In our work, we focus on reachability, isolation, waypoint, and load balancing policies (summarized in Table 1). The reachability, isolation, and load balancing policies are defined as predicates over a router `r` and a subnet in the network `p`. These evaluate to true if, for the given concrete environment, traffic from router `r` can reach the prefix `p`, is isolated from `p`, or load balanced on at least two paths to `p`, respectively. The waypoint policy is defined over two routers `r` and `w`, and evaluates to true if, for the given concrete environment, traffic from `r` destined to prefix `p` passes through `w`. We note that our approach is extensible to any policy that is defined over the forwarding state (e.g., equal length paths).

Problem definition We now define the problem of mining a network specification:

Given a network configuration and a failure model, mine the network specification, i.e., the set of all policies which hold under the failure model.

For our running example and the failure model $L_{symbolic} = L$ and $k = 1$ (modeling up to one link failure), the network specification consists of the following policies:

```
reachability(1, p1), reachability(1, p2),
reachability(2, p1), reachability(2, p2),
reachability(3, p1), reachability(3, p2),
reachability(4, p1), reachability(4, p2),
reachability(5, p2), loadbalancing(4, p2).
```

Baseline solutions To address the above problem, one may consider two baseline approaches: *(i)* data plane analysis and *(ii)* control plane verification.

Data plane analysis Data plane analysis tools (e.g., [13, 17, 18]) enable reasoning of policies that hold for a certain concrete environment. Today, such tools are scalable enough to reason about all of our considered policies within seconds or minutes (mostly depending on the size of the network). Thus, one could use such tools to mine a specification by iterating over all concrete environments captured by the failure model, computing a data plane for each (from the configuration), and

analyzing them to infer the set of policies which hold for each concrete environment. The solution is then the intersection of all obtained policy sets. Fig. 2 (top) visualizes this approach. Initially, every policy is a candidate which can be part of the network specification (blue area). With every sampled data plane, the set of policies that hold for it are computed (shown in circle). These are then intersected with the policies of the previous samples (dashed circles). At the end, the remaining candidate policies are those that hold for all samples, and thus form the network specification (green area). Unfortunately, for large topologies or failure models with many concrete environments, this approach does not scale (see §8.2).

Control plane verification Control plane verification tools (e.g., [4]) enable checking individual policies for a given failure model. Technically, this can be accomplished by symbolically encoding the network, its configuration, the failure model and a policy into a formula, and then checking the satisfiability of this formula. Fig. 2 (bottom) visualizes this approach. Initially, all policies are part of the set of candidates of the specification. At every step, one policy (circle) is picked and posed as a query to the verifier. The verifier either returns that the policy holds (green) or shows a counterexample to disprove it (gray). In the end, every policy has either been verified or disproved. As in data plane analysis, while control plane verification tools scale to the policies that we consider, enumerating all possible policies and checking them one by one in the above manner is prohibitive (see §8.2).

3 Our Approach: *Config2Spec*

In this section, we first present our key insight of combining the two baseline approaches from §2 and explain the reasoning behind it. Then, we provide an overview of the system (details are provided in the following sections).

3.1 Key Insight

We address the problem of mining a network specification by combining the baseline approaches and leveraging their respective strengths: data plane analysis is efficient at pruning policies, while control plane verification is efficient at validating policies. The key idea of our combination is to reduce the space of policies by sampling forwarding states and pruning policies using data plane analysis, and then running control plane verification to verify a small set of remaining policies.

This combination works well because many policies which do not hold are *dense violations*. That is, they are violated for many of the concrete environments captured by the failure model. For example, in our running example and the failure model $L_{symbolic} = L$ with $k = 1$ (up to one failure), the policy `waypoint(3, 1, p2)` only holds for the concrete environment in which all links are up, but the one from router `3` to `4`. Thus, by sampling any other concrete environment (e.g., $L_{down} = \{$`2-5`$\}, L_{up} = L \setminus L_{down}$), and computing all policies that hold for it, we can prune `waypoint(3, 1, p2)`.

On the other hand, there are *sparse violations*, which are policies that do not hold for the failure model, but are violated only by very few concrete environments. For example, in our running example and the same failure model, the policy `isolation(5, p1)` is violated only by two concrete environments: *(i)* $L_{down} = \{$`2-5`$\}, L_{up} = L \setminus L_{down}$ and *(ii)* $L_{down} = \{$`1-2`$\}, L_{up} = L \setminus L_{down}$. Unless we check these particular environments, this policy cannot be pruned by data plane analysis. Thus, we prune sparse violations during the step of control plane verification. Since the overall number of true policies and sparse violations is often significantly smaller than the number of concrete environments, control plane verification is an efficient solution for this.

3.2 The *Config2Spec* System

We build on this insight to design *Config2Spec* (Fig. 3), which takes as input the network configuration (of all devices) and a failure model and outputs the network specification.

Config2Spec runs in a loop which dynamically switches between the two approaches until the specification is mined. To achieve this, *Config2Spec* relies on three main components: *(i)* predictors, *(ii)* data plane analysis, and *(iii)* control plane verification. In addition, *Config2Spec* maintains two sets of policies, `cands` which overapproximates the specification, and `verified` which underapproximates it. We next explain these sets, the algorithm flow and the three components. We provide the full algorithm of *Config2Spec* in Appendix A.

Cands and verified *Config2Spec* keeps two sets: *(i)* `cands`, containing the current candidate policies, i.e., the policies that are known to hold or have not been pruned yet, and *(ii)* `verified`, containing the policies that are known to hold. `cands` initially contains all possible policies (blue area in Fig. 3), while `verified` is initially empty (green area in Fig. 3). We note that in practice, to avoid storing all policies in `cands`, only to prune many of them upon the first iteration of data plane analysis, *Config2Spec* directly initializes `cands` to the set of policies that holds for some concrete environment.

An invariant of the execution is that `cands` is a superset of the network specification, i.e., it contains at least all the policies that hold, while `verified` is a subset of it, i.e., it contains only policies that hold. *Config2Spec* terminates when these sets are equal – implying both equal the network specification – and then returns `verified`. Precision is ensured as *Config2Spec* does not miss any policy thanks to the invariant that `verified` contains only true policies (no false positives), while `cands` cannot miss a true policy (no false negatives).

Flow At each iteration, *Config2Spec* checks if `cands` equals `verified`. If so, it terminates. Otherwise, it checks two predictors to decide which approach is the more promising one to pursue: data plane analysis or control plane verification.

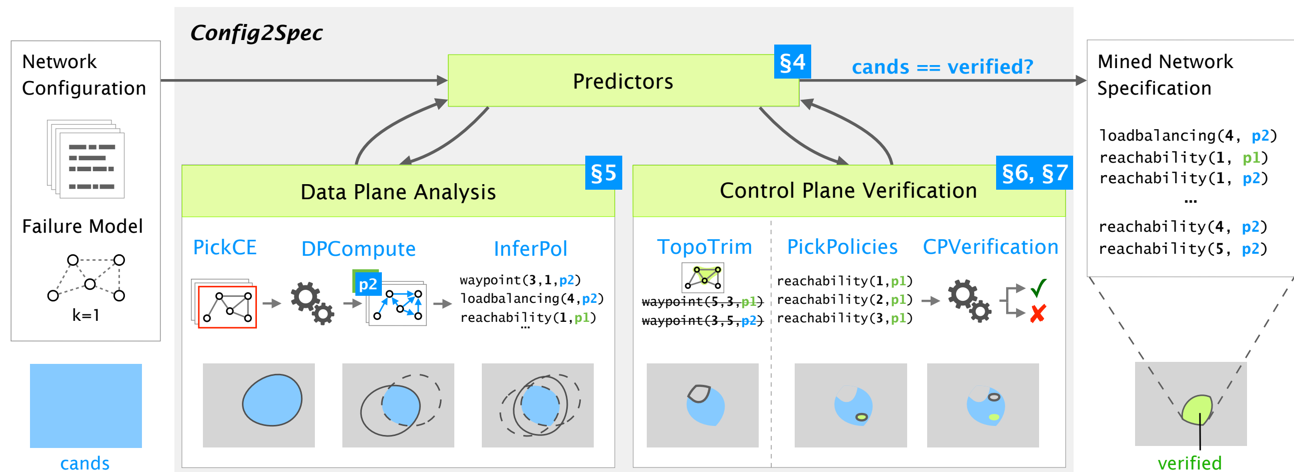


Figure 3: *Config2Spec* mines the specification from the network configuration and the failure model. It relies on three components: predictors, data plane analysis, and control plane verification. It maintains two sets: `cands`, consisting of the current candidate policies, and `verified`, consisting of the verified policies. During the execution, policies are removed from `cands` or added to `verified`. When `cands` equals `verified`, both equal the network specification, and then `verified` is returned.

Predictors (§4) We design two predictors to heuristically estimate which approach is likely to be more effective and dynamically transition between them. The predictors consider the execution times and the number of pruned and verified policies. The first predictor checks the effectiveness of each approach in classifying policies by measuring the time it needs to classify a single policy. The second predictor estimates the remaining time to mine the full specification.

Data plane analysis (§5) In every iteration of data plane analysis, *Config2Spec* samples a concrete environment, computes the policies that hold for it, and removes from `cands` any other policy. To sample a concrete environment, it executes `PickCE`, which employs a novel policy-aware sampler to find a concrete environment likely to prune more policies. Then, *Config2Spec* computes the data plane of that sample via `DPCompute`, which relies on prior tools (e.g., [13]). Next, it executes `InferPol` to compute all policies which hold for this data plane, and updates `cands` accordingly. Finally, *Config2Spec* checks whether all data planes have been analyzed. If so, it sets `verified` to `cands`, as the entire failure model has been covered and the full specification has been mined.

Control plane verification (§6) In each iteration of control plane verification, *Config2Spec* verifies a set of policies. For this, *Config2Spec* first executes `PickPolicies` to pick the next set of policies to verify. It then calls `CPVerification`, which relies on prior tools (e.g., [4]). The verifier either determines that all policies hold or returns a counterexample. In the former case, *Config2Spec* adds all the policies to `verified`, while in the latter case *Config2Spec* removes the ones violated by the counterexample from `cands`. Before the first iteration of control plane verification, *Config2Spec* invokes `TopoTrim` to reduce the verification overhead.

Topology-based trimming (§7) `TopoTrim` analyzes the topology and the failure model to trim (i.e., prune) policies which cannot hold regardless of the configuration (e.g., due to a lack of connectivity). It relies on graph algorithms to prune `reachability`, `waypoint`, and `loadbalancing` policies.

4 *Config2Spec*'s Predictors

In this section, we describe how *Config2Spec* dynamically decides whether to run the data plane analyzer or the control plane verifier. This decision relies on two predictors that capture the effectiveness of the approaches and the expected time remaining. Accordingly, *Config2Spec* infers which approach is more likely to make better progress. The predictors are: *(i)* the *Time-per-policy (TP) predictor*, favoring the approach more likely to classify more policies in a single execution, and *(ii)* the *Remaining-time (RT) predictor*, favoring the approach more likely to complete faster. If the predictors disagree on the approach, *Config2Spec* runs the data plane analyzer, we explain the reason for this choice shortly.

High-level behavior The predictors dynamically identify the different stages of the algorithm. In the beginning, sampling concrete environments is likely to provide the fastest progress, as at this stage the dense policies have not been pruned yet. Therefore, the TP predictor prefers data plane analysis initially. After most of the dense policies have been pruned, sampling environments may not significantly decrease the number of candidate policies anymore. At this point, the TP predictor starts to prefer control plane verification. Thus, the choice is then up to the RT predictor. It determines whether *Config2Spec* switches to control plane verification. If running data plane analysis for the remaining concrete environments

is likely to be faster than running control plane verification on the remaining unclassified policies, the RT predictor prefers data plane analysis. Otherwise, it prefers control plane verification. This choice depends on the failure model: if it captures a small number of concrete environments, enumerating all of them can be faster than verifying the remaining set of candidate policies. In our running example and the failure model $L_{symbolic} = L$ and $k = 1$, this is the case. To conclude, the joint behavior of the predictors is to prefer control plane verification whenever *(i)* there is a large number of concrete environments and *(ii)* most remaining policies are true policies (i.e., part of the specification) or sparse violations.

Computation The predictors rely on statistics of the previous runs. The TP predictor is implemented by tracking two times: $T^{TP}_{analysis}$ and T^{TP}_{verify}, which record the average time to classify a single policy through analysis or verification (respectively). For $T^{TP}_{analysis}$, this time is computed by taking the ratio of the execution time of the last run of the data plane analysis and the number of policies which were pruned as a result of this analysis. For T^{TP}_{verify}, this time is computed similarly by taking the ratio of the execution time of the last run of the verifier and the number of policies which were classified by the verifier. The latter number is one of the following. If the verifier proved all policies hold, it equals the number of policies. Otherwise, if the verifier returned a counterexample, this number equals to the number of policies which were discovered as violations (i.e., the counterexample violated them). The TP predictor prefers the data plane analyzer if $T^{TP}_{analysis} < T^{TP}_{verify}$.

The RT predictor is implemented by tracking two (different) times: $T^{RT}_{analysis}$ and T^{RT}_{verify}, which record the execution time of a single run of the analyzer and verifier (respectively). The RT predictor prefers the data plane analyzer if the remaining time of the analyzer, obtained by multiplying $T^{RT}_{analysis}$ with the number of non-analyzed concrete environments is smaller than the remaining time of the verifier, given by multiplying $T^{RT}_{verifier}$ with the remaining number of unclassified policies.

Initialization To initialize T^{TP}_{verify} and T^{RT}_{verify}, *Config2Spec* executes the verifier on M policy sets (in our implementation, $M = 10$). It then sets T^{RT}_{verify} to the average execution time of the verifier, and T^{TP}_{verify} to the average ratio of execution time and policies verified or pruned. The estimates $T^{TP}_{analysis}, T^{RT}_{analysis}$ are initially 0, to guide *Config2Spec* to begin by data plane analysis. This captures our premise that initially data plane analysis is likely to classify more policies (the dense violations, which are the vast majority of the policies).

Windows To smoothen the behavior of the predictors, the times are averaged over the last N runs of the analyzer or verifier (in our implementation, $N = 10$).

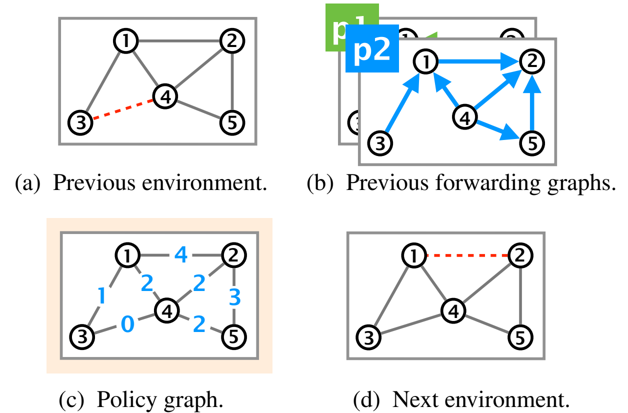


Figure 4: The policy graph is computed from the forwarding graphs of a previously analyzed concrete environment and guides us to an environment likely to prune more policies.

5 Data Plane Analysis

In this section, we present the key ingredients of running the data plane analysis in *Config2Spec*: the selection of the next concrete environment to analyze (`PickCE`), the computation of the data plane for that environment (`DPCompute`) and the inference of the policies from the data plane (`InferPol`).

5.1 Selection of Concrete Environments

At every iteration, one concrete environment is analyzed. The choice of this environment has a great impact on the overall runtime of the system. Thus, we design a sampling technique to pick the next concrete environment to prune a large number of policies from the set of candidates (`cands`). We call this technique *policy-aware sampling* as the next environment is picked based on the *policy graph*, a concept reflecting the current set of candidate policies, which we describe next.

Policy graph The *policy graph* for a given concrete environment is a copy of the network topology, augmenting the links with the number of policies that forward traffic along them. We say a `reachability(r,p)` policy forwards traffic along a link, if that link is part of a path in the forwarding graph of `p` from `r` to `p`. We define it similarly for the other policies. The policy graph allows us to identify the links on which large numbers of policies depend. Thus, we can pick a concrete environment in which these links are down. If the policies indeed hold only thanks to these links, they will be discovered as violations when analyzing this concrete environment.

We next define the policy graph. Given a network topology, a configuration, and a concrete environment, the policy graph extends the network topology with a mapping of links to weights (integers). The weight of a link represents the number of unclassified policies whose traffic is forwarded along that link. The weight is computed from the *forward-*

ing graphs of the concrete environment. Fig. 4 illustrates the concept of the policy graph using our running example (Fig. 1). Here, we are given an (already analyzed) concrete environment where all links are up, but the one between routers `3` and `4` (Fig. 4a). In this example, there are two destinations (`p1` and `p2`) and hence two forwarding graphs (Fig. 4b). For simplicity's sake, consider the following unclassified policies for destination `p2`: `reachability(i, p2)`, where `i` ranges over all five routers, and `loadbalancing(4, p2)`, which holds since router `4` has three paths to router `2` in the forwarding graph of `p2`. In this setting, the policy graph (Fig. 4c) maps, for example, link `1-3` to 1 (as only `reachability(3, p2)` depends on this link), link `2-5` to 3 (for `reachability(4, p2)`,`reachability(5, p2)` and `loadbalancing(4, p2)`), `1-2` to 4 (for `reachability(1, p2)`, `reachability(3, p2)`, `reachability(4, p2)` and `loadbalancing(4, p2)`), and `1-2` (which is down) to 0.

Policy-aware sampling Based on the idea of the policy graph, we design a policy-aware sampler for `PickCE`. The policy-aware sampler picks the next concrete environment to analyze based on the policy graph of the previously analyzed concrete environment and the current set of unclassified policies (`cands\verified`). This is done by selecting the links to add to L_{down} based on a probability distribution proportioned to the links' weights in the policy graph. The links' weights are computed by iterating over all unclassified policies (`cands\verified`) and counting, for each link, the number of policies that are forwarded along it. The probability distribution is needed to avoid getting stuck: a deterministic approach which adds the heaviest links to L_{down} can result in an oscillation between two concrete environments which already have been analyzed (we observed this phenomenon in practice). Adding non-determinism mitigates this issue, and in case it cannot, `PickCE` resorts to returning a random concrete environment which has not yet been analyzed. In the beginning, *Config2Spec* starts by analyzing the concrete environment in which all symbolic links are up.

For our running example and the policy graph in Fig. 4c, it assigns the link `1-3` to the probability $\frac{1}{14}$, `2-5` to $\frac{3}{14}$, and `1-2` to $\frac{4}{14}$. Assuming the usual failure model ($L_{symbolic} = L$ and $k = 1$), it then picks the next concrete environment by choosing one link that is down based on the distribution. For example, it picks the link `1-2` (Fig. 4d).

5.2 Analysis of a Concrete Environment

We now explain `DPCompute` and `InferPol`, which together compute all policies that hold for a given concrete environment and configuration.

The `DPCompute` algorithm executes two steps. First, for each router in the network, it computes the router's forwarding state. The forwarding state of a router is a list of destination prefix and next hop pairs. A pair `(p,w)` in the forwarding state of router `r` indicates that traffic reaching `r` for destination `p` is sent to router `w`. Computing the forwarding state of the routers is not trivial, however, there are solutions to efficiently compute them (e.g., [13]).

In the second step, `DPCompute` builds from the routers' forwarding states the forwarding graphs. It builds one forwarding graph for each equivalence class of destination prefixes (i.e., prefixes which can be captured via some prefix and have the same forwarding graph). The forwarding graph of a prefix `p` is a directed graph in which we have a link from router `r` to `w` if, according to `r`'s forwarding state, traffic for `p` is sent to `w`.

From the forwarding graphs, `InferPol` computes the policies by leveraging graph algorithms. For `reachability` and `waypoint` policies, it builds the *dominator tree* of all forwarding graphs. A dominator tree is a tree rooted at the destination of the forwarding graph. Its nodes are all routers that have at least one path to the destination. A router `a` is a child of a router `b` if *(i)* traffic from router `a` to the destination must pass through router `b` and *(ii)* for any other router `c` such that traffic from `a` must pass through it, traffic from `b` must also pass through it. `InferPol` infers a `reachability(r, p)` policy for every node `r` in the dominator tree of `p`. It further infers `waypoint(r,w,p)` for all routers `r` which are dominated by a waypoint `w` in the dominator tree of `p`. For `loadbalancing`, it computes the shortest paths in the network and infers `loadbalancing(r,p)` for routers `r` with multiple paths of the same cost available to reach destination `p`. For `isolation`, it infers `isolation(r,p)` for every router `r` and prefix `p` for which it has not inferred `reachability(r,p)`.

6 Control Plane Verification

Here, we present the two ingredients of the control plane verification in *Config2Spec*: the selection of policies to verify next (`PickPolicies`) and their verification (`CPVerification`).

CPVerification We begin with `CPVerification`, which takes as input a set of policies, the network configuration and the failure model. It checks whether all policies hold for any concrete environment meeting the failure model (for the given network configuration), or returns a counterexample.

Technically, the verifier symbolically encodes the configuration and the failure model as logical constraints: φ_{net} and φ_{fmodel}. The set of policies is encoded as a conjunction over formulas encoding the policies: $\varphi_{pols} = \bigwedge_{pl \in pols} \varphi_{pl}$. The verifier checks the satisfiability of $\varphi_{net} \wedge \varphi_{fmodel} \wedge \neg\varphi_{pols}$. If it is unsatisfiable, then all policies in *pols* hold. If the formula is satisfiable, then there is a counterexample, i.e., a concrete environment captured by the failure model, which under the given configuration violates φ_{pols} (i.e., at least one policy is violated). While the challenge of verifying network policies is not trivial, there are effective solutions (e.g., [4]).

PickPolicies This procedure takes the set of candidate policies (`cands`) and verified policies (`verified`) and returns the next set of policies to verify (from `cands` $\setminus$ `verified`). Since verifying is computationally expensive, the goal is to minimize the overall execution time of the verifier. By choosing a set of policies which have a dependency, the overall execution time of verifying them can be smaller than if they were verified one by one. Towards this goal, `PickPolicies` returns a maximal set of policies with the same destination prefix `p`.

We pick `p` arbitrarily, as once *Config2Spec* chooses to run the verifier, usually most policies are true policies.

Our grouping approach is always at least as good as verifying the policies one by one. The reason is that at each query to the verifier, at least one policy is classified. In the worst case, only one policy is classified as violation (if the verifier returned a counterexample which satisfies all policies but one). In a better case, several policies are classified as violation. In either of these cases, the violated policies are removed from `cands`, while the other policies in the set remain in `cands` (and will be verified in a later execution of `CPVerfication`). In the best case, all policies are classified as true policies. Namely, we can only gain from verifying multiple policies in the same execution of the verifier. Further, our grouping is maximal – grouping of policies with different prefixes is not helpful, as each prefix has a different forwarding graph, and so the verifier does not gain from grouping such policies.

7 Topology-based Trimming

In this section, we describe `TopoTrim`, a technique which reduces the load on the control plane verification by analyzing the failure model and the network topology. `TopoTrim` classifies policies as violations if their minimal connectivity requirements are not met under the given failure model.

`TopoTrim` is executed the first time *Config2Spec* chooses to run the verifier. It relies on the insight that some policies can be classified as violations directly from the network topology and failure model. For example, consider the network in Fig. 1 and the failure model with $L_{symbolic} = L$ and $k = 2$ (i.e., up to two link failures). We can infer that `reachability(3, p1)` cannot hold as `3` can become disconnected from the rest of the network if both links connected to it fail. For the same reason, any `waypoint` or `loadbalancing` policy where `3` is involved can be classified as violation.

To prune such policies, `TopoTrim` computes the $(k+1)$-edge-connected components of the topology for a failure model with k permitted failures. A $(k+1)$-edge-connected component is a set of nodes which remain connected even after removing any k edges. For example, for the network in Fig. 1 and the same failure model (where $k = 2$), the following routers are in a 3-edge-connected component: $\{1,2,4\}$.

There are efficient algorithms to compute $(k+1)$-edge-connected components, however they do not support links that must be up or down (L_{up} or L_{down}). To take these into account, `TopoTrim` first removes from the topology all links in L_{down}, updates k to $k - |L_{down}|$, and then, for each link in L_{up}, it adds k additional links between the routers to simulate that these routers are $(k+1)$-edge-connected. For example, for $L_{up} = \{(1,3)\}$, $L_{down} = \emptyset$ and $k = 2$, it adds two more edges between `1` and `3`, so they are considered 3-edge-connected.

Based on this, `TopoTrim` classifies the following policies as violations (which are thus removed from `cands`). The policies `reachability(r,p)` and `loadbalancing(r,p)`, for any router `r` and prefix `p` such that (r, r_p) is not in a $(k+1)$-edge-component, where r_p is the router attached to `p`. The policy `waypoint(r,w,p)` is classified as violation for any routers `r` and `w` and a prefix `p` such that *(i)* (r, w) is not in a $(k+1)$-edge-component or *(ii)* (w, r_p) is not in a $(k+1)$-edge-component, where r_p is the router attached to `p`.

8 Experimental Evaluation

In this section, we evaluate *Config2Spec* on multiple topologies to address the following research questions:

RQ1 How does *Config2Spec* scale to realistic topologies? We show that even for large networks with 158 routers and 189 links, it completes within 2.7 hours for OSPF configurations and 13.7 hours for BGP configurations.

RQ2 How does *Config2Spec* compare to the baselines? We show it improves the best one by up to a factor of 8.3.

RQ3 How do the domain-specific techniques contribute to *Config2Spec*? We show that *(i)* the policy-aware sampler leads to smaller candidate sets by up to a factor of 2 compared to random, and obtains them with fewer samples, and *(ii)* topology-based trimming and policy grouping reduce the queries by up to a factor of 2'500.

RQ4 Can *Config2Spec* be run on a real network configuration? We illustrate this on the Internet2 configuration.

Implementation *Config2Spec* is implemented in 5k lines of Python and Java code.[1] It computes the routers' forwarding states (§5.2) using Batfish [13], and verifies policies using Minesweeper [4]. We extended Minesweeper with the `waypoint` and `loadbalancing` policies. We note that while our implementation supports only configurations and features supported by these two third-party tools, our approach is not limited to specific configuration types or features.

Config2Spec takes as input the routers' configurations and a failure model. It outputs all policies that hold for the provided input. For large networks, we assume the network operator provides a list of devices that act as waypoints (e.g., middleboxes). In our experiments, we simulate it by randomly picking 20% of the routers to serve as waypoints.

Experiment setup To study how *Config2Spec* scales as a function of the topology size, we picked three topologies (small, medium, and large) from the Topology Zoo collec-

[1] Code is available at https://github.com/nsg-ethz/config2spec.

Topology	k	Config	Overall	DPA	CPV
BICS	1	OSPF	38.8 s	100%	0%
		BGP	68.3 s	100%	0%
	2	OSPF	228.8 s	30%	70%
		BGP	1’341.2 s	85%	15%
	3	OSPF	117.4 s	27%	73%
		BGP	319.7 s	14%	86%
Columbus	1	OSPF	398.0 s	100%	0%
		BGP	457.2 s	100%	0%
	2	OSPF	1’328.1 s	18%	82%
		BGP	6’772.0 s	17%	83%
	3	OSPF	907.0 s	27%	73%
		BGP	2074.1 s	18%	82%
US Carrier	1	OSPF	6’386.2 s	100%	0%
		BGP	6’813.4 s	100%	0%
	2	OSPF	10’528.4 s	15%	85%
		BGP	49’151.0 s	6%	94%
	3	OSPF	2’542.5 s	59%	41%
		BGP	5’873.3 s	34%	66%

Table 2: Execution time of *Config2Spec* as a function of the network topology, number of failures and configuration type.

tion [19]: BICS with 33 routers connected by 48 links, Columbus with 70 routers and 85 links, and US Carrier with 158 routers and 189 links. We used NetComplete [10] to synthesize OSPF and BGP configurations using its path-ordering specifications for 2, 4, 8 and 16 prefixes. For each configuration type and topology, we generated 5 configuration sets.

For each set of router configurations, *Config2Spec* computes all policies which hold, for all four policy types in Table 1. We consider three failure models, where k is 1, 2, or 3, and we fix $L_{up} = L_{down} = \emptyset$ and $L_{symbolic} = L$ (i.e., any link can be up or down). The reported results are averaged over these runs and the two configuration types (i.e., OSPF and BGP). We ran all experiments in virtual machines with 32 GB of RAM and 12 virtual cores running at 2.3 GHz.

8.1 Scalability of *Config2Spec*

We begin by studying how *Config2Spec* scales to realistic topologies. To this end, we ran experiments on all three topologies and three failure models, and measured the time *Config2Spec* spent on the data plane analysis part – including `PickCE`, `DPCompute` (which invoked Batfish), and `InferPol` – and the control plane verification part – including `PickPolicies` and `CPVerification` (which invoked Minesweeper). The other parts completed in negligible times and were thus ignored (e.g., `TopoTrim` completed within five seconds for US Carrier and less than a second for BICS).

Table 2 shows the overall execution time (*Overall*) and how it is split between data plane analysis (*DPA*) and control plane verification (*CPV*) as a function of the topology, the number of failures (k), and the configuration type (*Config*). For example, for the US Carrier topology with $k = 3$ and OSPF configurations, *Config2Spec* completed within 43 minutes, where 59% of that time was spent on data plane analysis.

The results show that even for the US Carrier topology with its 158 routers and 189 links, *Config2Spec* mined the specification in a reasonable time (within 2.7 hours, for OSPF, and 13.7 hours, for BGP). The results also demonstrate that the runtime mainly depends on the network size, secondly on the failure model, and lastly on the configuration type. This is expected: the larger the network, the larger the set of candidate policies and the set of concrete environments (whose size also depends on the failure model). In contrast to the effect of the network size on the execution times, the permissiveness of the failure model shows a different trend: execution times increase from $k = 1$ and $k = 2$, but drop for $k = 3$. This is thanks to the topology-based trimming (§7), which becomes very significant for $k = 3$ (or higher values of k). For the evaluated topologies, most router pairs are not 4-edge-connected, thus many policies are pruned. We provide more details on trimming in §8.3. The results show also that for $k = 1$, *Config2Spec* only performs data plane analysis. This is because the number of concrete environments is significantly smaller than the number of candidate policies throughout execution, leading the RT predictor to favor data plane analysis. Lastly, results show that for BGP configurations, the execution time is higher than for OSPF configurations. This is mainly due to Minesweeper, for which we observe a five to ten times increase in the verification time for BGP compared to OSPF.

Topology	k	Candidates	Specification	Percent
BICS	1	2’526.9	1’008.1	40%
	2	2’504.4	304.0	12%
	3	2’482.1	57.6	2%
Columbus	1	13’290.2	4517.1	34%
	2	13’150.4	350.4	3%
	3	13’271.0	27.2	0.2%
US Carrier	1	93’416.2	17’908.3	18%
	2	85’021.0	702.8	0.8%
	3	98’837.6	6.8	0.01%

Table 3: The number of candidate policies and the number of policies in the specification *Config2Spec* returns. Percent shows the fraction of the policies of all candidate policies.

Table 3 reports the number of candidate policies and the number of policies in the specification, for each topology and failure model, averaged across the different configuration sets and the configuration types. The reported number of candidate policies is the number of policies that hold for the first concrete environment picked by *Config2Spec* (*Config2Spec* always begins with data plane analysis). We consider this set as the initial set of candidates, rather than all instantiations

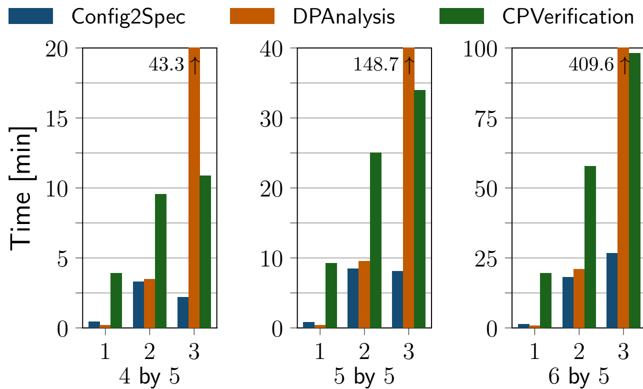


Figure 5: *Config2Spec* compared to the baselines of data plane analysis and control plane verification on grid topologies and different failure models. The bars of DPAnalysis and $k = 3$ are cut, and their maximum value is denoted next to them.

of the four policy types (Table 1), as the latter contains many policies which no concrete environment satisfies.

The results indicate that as the network size increases, the number of candidate policies increases, while the specification size (i.e., the number of policies that hold for all concrete environments) significantly drops. This demonstrates the challenge of *Config2Spec* to search in the large space of candidate policies for the small set of policies that hold.

8.2 Comparison to Baselines

We compare *Config2Spec* to the two baselines in §2: *(i)* a data plane analysis approach, which enumerates all data planes to infer the specification, and *(ii)* a control plane verification approach, which verifies the candidate policies one by one. As neither of the baselines scales to the larger networks considered in the last section, in this experiment, we use three grid topologies of sizes: 4 by 5, 5 by 5 and 6 by 5. We generated five sets of OSPF configurations per topology and used the failure model $L_{symbolic} = L$ with k ranging from 1 to 3.

Fig. 5 shows the execution time of each approach as a function of the topology and failure model. For $k = 2$ and $k = 3$, *Config2Spec* outperforms the baselines: the data plane analysis by 10.2x on average and up to 41.0x, and the control plane verification by 3.8x on average and up to 8.3x. For $k = 1$, data plane analysis is faster than *Config2Spec* because of *Config2Spec*'s setup time (i.e., the verification of few policies when initializing the predictors' times, see §4). Still, the overhead of *Config2Spec* is small (data plane analysis was faster on average by 24 seconds and by up to 37 seconds).

The results also show that both baselines have benefits. For less permissive failure models, data plane analysis performs better than control plane verification, whereas for permissive failure models it is the other way around. This demonstrates the advantage of the dynamic combination of *Config2Spec*.

8.3 Domain-specific Techniques

We next study how the domain-specific techniques improve *Config2Spec*'s performance. We study the following aspects: *(i)* how the policy-aware sampler (§5.1) helps reducing the number of concrete environments *Config2Spec* analyzes, and *(ii)* how topology-based trimming (§7) and policy grouping (§6) decrease the number of queries posed to the verifier.

Policy-aware sampler We compare the policy-aware sampler (called Policy-Aware) to a baseline which randomly picks a new concrete environment (called Random). We compare them by instantiating `PickCE` with each approach and running *Config2Spec* on the Topology Zoo topologies with the failure model $L_{symbolic} = L$ and $k = 3$, and with five sets of OSPF configurations and five sets of BGP configurations.

Table 4 shows the results. The first four columns show, for each approach, how many concrete environments were analyzed before *Config2Spec* transitioned to the verifier, and how many policies remained to verify (i.e., the percentage of remaining policies out of the policies that hold for the first sample). For example, for BICS, Policy-Aware required on average 36.4 samples before *Config2Spec* switched to verification, and at this point the size of the candidate policy set was reduced to 36.5% of the initial policy set (i.e., the set of policies which hold for the first sample).

Generally, the smaller the set of remaining policies (i.e., the closer the candidate set to the network specification is), the better. As a secondary goal, the number of analyzed concrete environments should be relatively small. Results indicate that Policy-Aware always obtains a better reduction in the size of the candidate set compared to Random. They also show that on average Policy-Aware typically required fewer samples than Random. However, we note that in 6 out of the 30 experiments, Random switched to verification before Policy-Aware did. This is not because Random made better progress. In contrary, the TP predictor decided to switch, as it observed that the concrete environments picked by Random were not effectively pruning policies anymore.

The next two columns of Table 4 provide more statistics. We checked, for each experiment, the relative size of the candidate sets for both approaches when *Config2Spec* with Policy-Aware transitioned to verification. For example, in one experiment using BICS, Policy-Aware transitioned to verification after 32 samples, and at that point the number of candidate policies was 970, while for Random, after 32 samples, there were 1'124 candidate policies, making the ratio 86.3%. In Table 4, Cands Ratio shows the average over the ten runs. We also checked how many additional samples Random required to reduce the candidate policies to (at most) the size obtained with Policy-Aware. For example, in that experiment for BICS, Policy-Aware required 32 samples to reduce the candidates to 970 policies, while Random required 43. Hence, Random needed 11 additional samples. In Table 4, Added Samples shows the average of this number. The re-

Topology	Policy-Aware Samples	Policy-Aware Candidates	Random Samples	Random Candidates	Cands Ratio	Added Samples	Policy-Aware PickCE	Policy-Aware DPAnalysis	Random PickCE	Random DPAnalysis
BICS	36.4	36.5%	42.1	39.5%	89.7%	45.7	22.1 ms	1.4 s	0.5 ms	1.3 s
Columbus	71.0	16.6%	79.0	26.4%	60.5%	109.0	63.1 ms	8.3 s	0.7 ms	7.7 s
US Carrier	113.8	9.6%	122.1	18.6%	51.6%	>500	358.2 ms	57.8 s	1.4 ms	51.6 s

Table 4: Comparison of the policy-aware sampler in `PickCE` (§5.1) and a random baseline. Samples is the number of samples before *Config2Spec* switched to verifier. Candidates is the percentage of remaining candidate policies at that point. Cands Ratio is the ratio of the candidate set sizes for Policy-Aware and Random when *Config2Spec* with Policy-Aware transitioned to verification. Added Samples is the number of samples Random needed to reduce the candidate set to the size of the candidate size with Policy-Aware. PickCE is the time to pick the next environment. DPAnalysis is the overall time to analyze a data plane.

sults indicate that Policy-Aware not only obtains a smaller candidate set, but reaches it significantly faster.

The last four columns of Table 4 show execution times: PickCE shows the execution time of the sampler, while DP-Analysis shows the overall execution time of a single data plane analysis (i.e., `DPCompute` and `InferPol`). Results show that while Policy-Aware takes more time than Random (as expected), the overhead is negligible compared to the overall execution time of the data plane analysis.

Topology-based trimming and policy grouping We next evaluate the topology-based trimming and policy grouping in reducing the number of queries to the verifier. We ran the experiments for the three topologies and the failure model with $k = 2$ and $k = 3$ (for $k = 1$, *Config2Spec* only performs data plane analysis §8.1). We measured how many queries to the verifier each technique saved. In every experiment, we recorded the number of policies *Config2Spec* had the first time it transitioned to the verification. This number, denoted B (for baseline), provides the number of queries to the verifier if we did not use either technique. We also recorded how many policies were pruned thanks to topology-based trimming. We count each policy that has been pruned as one saved query for the verifier, and denote the overall saved queries by T (for trimming). Also, we recorded how many queries were posed to the verifier (when employing policy grouping), and denote the number of queries by G (for grouping).

Fig. 6 shows the percentage of remaining queries after each optimization: $\frac{B-T}{B}$% for trimming and $\frac{G}{B}$% for policy grouping. For example, for BICS and $k = 2$, trimming pruned 51.1% of the policies. Policy grouping saved 41.5% and reduced the overall queries to the verifier to 9.6%. Overall, the reduction was 90.3%. The results show that the combination of trimming and policy grouping can reduce the number of queries to as little as 0.04%. Trimming is especially powerful for the larger topologies and for more permissive failure models ($k = 3$). The policy grouping also significantly reduces the number of queries to the verifier. The best case is for the largest network, where trimming reduced the number of queries to 1.15% and then policy grouping reduced it to 0.04%, compared to the baseline.

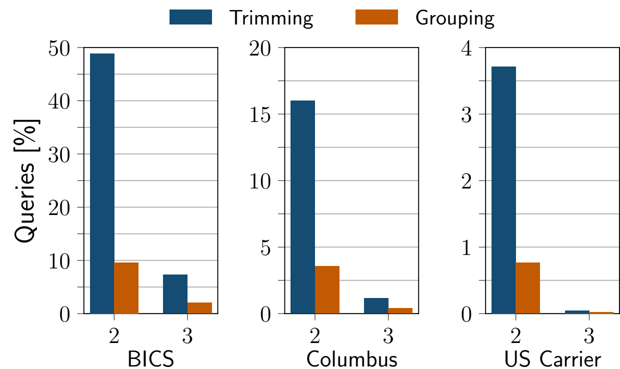


Figure 6: Reduction in the number of queries to the verifier thanks to topology-based trimming and policy grouping.

8.4 Running *Config2Spec* on Internet2

Finally, we demonstrate that *Config2Spec* can handle real configurations. For this, we took a publicly available configuration of the Internet2 network from May 2015 [8]. For Batfish to be able to parse this configuration, we had to remove multiple lines from it. Mostly, these parts concerned logging (e.g., `system dump-on-panic;`), anonymization leftovers (e.g., `Firewall Stanza Removed`) and other (for our purposes) irrelevant parts (e.g., `bfd-liveness-detection no-adaptation;`). For Minesweeper to be able to verify our queries, we had to remove parts of the BGP route-maps (community-matches and empty prefix-list matches). This does not affect the output, as we only mine the specification for internal prefixes, since no external peers are connected. In total, we had more than 90k lines of configuration. The topology consisted of 10 routers and 18 links. For a failure model with $L_{symbolic} = L$ and k from 1 to 3, *Config2Spec* required 32, 314, and 1'805 seconds to infer the network specification. It consisted of 3'962, 3'405, and 3'339 policies. The high number of policies, even for $k = 3$, stems from the fact that the five routers on the east-coast almost form a clique.

9 Related Work

In this section, we survey related work across five dimensions: specification mining, data plane analysis, control plane verification, network specification languages, and counterexample-guided inductive synthesis.

Specification mining Our work is inspired by works on specification mining [1], where high-level specifications are automatically inferred from low-level execution of programs. One example is Daikon [11], which dynamically detects program invariants (e.g., $x \neq 0$) by running the program and observing the values the program computes. For computer networks, Xie et al. [31] show how to compute the reachability specification for a given failure model based on the network's configuration. They compute the reachability upper bound – all policies that hold for at least one concrete environment – and lower bound – all policies that hold for all concrete environments. To scale, only an approximation of the bounds is computed. In contrast, *Config2Spec* computes the exact lower bound of reachability, as well as other policies, and thereby obtains a precise specification. Benson et al. [6] show how to mine reachability *policy units*, a high-level abstraction of pair-wise reachability, from network configurations for a single concrete environment. Like *Config2Spec*, it relies on data plane analysis. Unlike *Config2Spec*, failure models are not supported. Other works [7, 15] assess the complexity of managing the network and its overall health, i.e., the frequency of performance and availability problems, by analyzing its configurations.

Data plane analysis *Config2Spec* relies on a data plane analyzer. Several works exist and they differ mostly in their input. There are tools that require the forwarding state as input [16–18, 23], and others that compute the forwarding state from the network configuration [13]. These tools enable to check various properties, such as reachability and isolation, for the single forwarding state being analyzed.

Network verification *Config2Spec* also relies on a control plane verifier. Several works offer solutions for network verification, supporting different kinds of queries. Minesweeper [4] relies on an SMT-solver, and is currently the most general solution: it supports various properties (e.g., reachability, loop-freedom, router equivalence) and multiple (interacting) routing protocols. ERA [12] creates a unified control plane model that mainly allows to reason about reachability properties under multiple routing protocols. ARC [14] constructs an abstract graph representation of the data plane computation and supports various properties: reachability, isolation, waypointing and control plane equivalence. Many other tools focus on a single protocol such as Bagpipe [30].

Network specifications Many works introduce different network specification languages, varying in their expressiveness. Some allow to capture traffic classes at the path-level [3,5,27], while others use a higher-level abstraction describing traffic classes and high-level policies such as reachability and waypointing [24]. Despite the differences, *Config2Spec*'s output can be used by other tools, such as NetKAT [3], whose language can accommodate the policies we consider.

Counterexample-guided inductive synthesis (CEGIS) CEGIS is a technique in program synthesis in which examples guide the search for the target program [25, 26]. Technically, from an initial set of examples (which may be empty), the synthesizer proposes a candidate program consistent with the examples, and introduces it to a validator. The validator either confirms the candidate is the target program or returns a counterexample. The counterexample is added to the set of examples, guiding the synthesizer to look for a different candidate. *Config2Spec* can be seen as a synthesizer looking for (all) policies that hold for a given network configuration and failure model. Like CEGIS, it is guided by examples (the data planes) and a validator (the verifier). Unlike CEGIS, *Config2Spec* looks for *all* valid policies (and not a single one). This poses a greater challenge, both in terms of the search space and the burden on the validator. To cope, *Config2Spec* cleverly samples examples to prune the search space (without the help of the validator), trims and groups policies to save queries to the validator, and dynamically switches between sampling and verifying to expedite the search.

10 Conclusion

We introduced *Config2Spec*, a scalable approach for mining a network's specification from its configuration and a failure model. The key insight is to dynamically switch between data plane analysis and control plane verification. To scale further, we integrated three domain-specific techniques: *(i)* policy-aware sampling to pick concrete environments which are more promising for policy pruning, *(ii)* policy grouping to group queries and thereby reduce verification overhead, and *(iii)* topology-based trimming to prune policies infeasible for the given topology and failure model. We evaluated *Config2Spec* on different topologies and against two baselines. The results show that *Config2Spec* scales to large networks, unlike the baselines, and that our domain-specific techniques significantly contribute to the scalability.

Acknowledgements

We thank our shepherd Aditya Akella and the anonymous reviewers for the constructive feedback and comments. We are especially grateful to Ahmed El Hassany for his feedback and support with NetComplete.

References

[1] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining Specifications. In *ACM POPL*, Portland, OR, USA, 2002.

[2] Glenn Ammons, David Mandelin, Rastislav Bodík, and James R Larus. Debugging Temporal Specifications with Concept Analysis. In *ACM PLDI*, San Diego, CA, USA, 2003.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic Foundations for Networks. In *ACM POPL*, San Diego, CA, USA, 2014.

[4] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A General Approach to Network Configuration Verification. In *ACM SIGCOMM*, Los Angeles, CA, USA, 2017.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *ACM SIGCOMM*, Florianopolis, Brazil, 2016.

[6] Theophilus Benson, Aditya Akella, and David A. Maltz. Mining Policies from Enterprise Network Configuration. In *ACM IMC*, Chicago, IL, USA, 2009.

[7] Theophilus Benson, Aditya Akella, and David A Maltz. Unraveling the Complexity of Network Management. In *USENIX NSDI*, Boston, MA, USA, 2009.

[8] Min Cheng. Small. `https://github.com/jayvischeng/Small/tree/master/ServerData2`, 2015.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Network-wide Configuration Synthesis. In *CAV*, Heidelberg, Germany, 2017.

[10] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *USENIX NSDI*, Renton, WA, USA, 2018.

[11] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Elsevier Science of Computer Programming*, 69(1-3):35–45, 2007.

[12] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *USENIX OSDI*, Savannah, GA, USA, 2016.

[13] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. A General Approach to Network Configuration Analysis. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[14] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast Control Plane Analysis using an Abstract Representation. In *ACM SIGCOMM*, Florianopolis, Brazil, 2016.

[15] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. Management Plane Analytics. In *ACM IMC*, Tokyo, Japan, 2015.

[16] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real Time Network Policy Checking using Header Space Analysis. In *USENIX NSDI*, Lombard, IL, USA, 2013.

[17] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *USENIX NSDI*, San Jose, CA, USA, 2012.

[18] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying Network-Wide Invariants in Real Time. In *USENIX NSDI*, Lombard, IL, USA, 2013.

[19] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The Internet Topology Zoo. *IEEE JSAC*, 29(9):1765–1775, 2011.

[20] Axel van Lamsweerde. Formal Specification: a Roadmap. In *ACM FOSE*, Limerick, Ireland, 2000.

[21] Claire Le Goues and Westley Weimer. Specification Mining With Few False Positives. In *TACAS*, York, United Kingdom, 2009.

[22] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking Beliefs in Dynamic Networks. In *USENIX NSDI*, Oakland, CA, USA, 2015.

[23] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the Data Plane with Anteater. In *ACM SIGCOMM*, Toronto, ON, Canada, 2011.

[24] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *ACM SIGCOMM*, London, United Kingdom, 2015.

[25] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *ACM PLDI*, Tucson, AZ, USA, 2008.

[26] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ACM ASPLOS*, San Jose, CA, USA, 2006.

[27] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A Language for Provisioning Network Resources. In *ACM CoNEXT*, Sydney, Australia, 2014.

[28] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks. In *ACM POPL*, Paris, France, 2017.

[29] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Synthesis of Fault-Tolerant Distributed Router Configurations. In *ACM SIGMETRICS*, Irvine, CA, USA, 2018.

[30] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *ACM OOPSLA*, Amsterdam, Netherlands, 2016.

[31] Geoffrey G Xie, Jibin Zhan, David A Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks. In *IEEE INFOCOM*, Miami, FL, USA, 2005.

Algorithm 1: *Config2Spec*(conf, $\mathcal{F}$)

Input : conf: The network configuration.
$\mathcal{F}$: the failure model (i.e., L_{up}, L_{down}, $L_{symbolic}$, k).

Output : A specification: the set of all policies that hold for the given configuration and failure model.

```
1  cands ← allPolicies()
2  verified, prevEnvs, lastFwds ← ∅,∅,∅
3  T^TP_verify, T^RT_verify ← initVerificationTimes()
4  T^TP_analysis, T^RT_analysis ← 0,0
5  totalEnvs ← Σ_{j=0}^{k} (|L_symbolic| choose j)
6  while cands ≠ verified do
7      DP-RT ← T^RT_analysis · (totalEnvs − |prevEnvs|)
8      CP-RT ← T^RT_verify · |cands \ verified|
9      if T^TP_analysis < T^TP_verify or DP-RT < CP-RT then
10         env ← PickCE(F, cands\verified, prevEnvs,
             lastFwds)
11         lastFwds, T^RT_analysis ← DPCompute(env, conf)
12         pols = InferPol(lastFwds)
13         T^TP_analysis ← (cands \ pols = ∅) ? ∞ : T^RT_analysis / |cands\pols|
14         cands ← cands ∩ pols
15         prevEnvs ← prevEnvs ∪ {env}
16         if |prevEnvs| = totalEnvs then verified ← cands
17     else
18         pols ← PickPolicies(cands, verified)
19         cex, T^RT_verify ← CPVerification(pols, conf, F)
20         if cex = ⊥ then
21             verified ← verified ∪ pols
22             T^TP_verify ← T^RT_verify / |pols|
23         else
24             cands ← cands \ {p ∈ pols| cex violating p}
25             T^TP_verify ← T^RT_verify / {p∈ pols| cex violating p}
26 return verified
```

A Main Algorithm of *Config2Spec*

Here, we present the main algorithm of *Config2Spec* (Algorithm 1). Our algorithm takes as input the configuration `conf` and a failure model $\mathcal{F}$ consisting of $L_{up}, L_{down}, L_{symbolic}$ and k. It outputs all policies which hold for this setting. The algorithm maintains the time estimates presented in §4 as well as a few sets. We next present these sets and the initialization of the time estimates, and afterwards the algorithm flow.

Main data structures The algorithm maintains four sets: `cands`, `verified`, `prevEnvs`, and `lastFwds`.

The `cands` set contains all policies that are still candidates for the network specification (i.e., unclassified policies and verified policies). That is, it is a superset of the network specification. When the algorithm terminates, `cands` is exactly the set of policies which hold. During the execution, policies which are discovered as violations are removed from `cands`. Initially, this set consists of all reachability, isolation, load balancing, and waypoint policies. Although we focus on these policies, our algorithm easily extends to any policy supported by the data plane analyzer and control plane verifier.

The `verified` set is the set of all policies that the verifier proved to be part of the network specification. That is, it is a subset of the set of policies which hold. When the algorithm terminates, `verified` is exactly the network specification. During the execution, policies which are discovered as true policies are added to it.

The `prevEnvs` set contains all previously analyzed concrete environments, while the `lastFwds` set contains the forwarding graphs of the last analyzed concrete environment, which is used to pick the next concrete environment.

Initialization of predictors' times As discussed in §4, our predictors rely on four time estimates: $T^{TP}_{analysis}$, $T^{TP}_{verifier}$, $T^{RT}_{analysis}$, and T^{RT}_{verify}. These are initialized as discussed in §4, where to initialize T^{TP}_{verify} and T^{RT}_{verify}, *Config2Spec* executes the verifier on M policy sets by running M times Line 18–Line 25, which are shortly explained.

Flow After initialization, *Config2Spec* runs in a loop which terminates when `verified` equals `cands`, indicating that both are equal to the network specification. At each iteration of the loop, *Config2Spec* first computes the predictors to pick between the data plane analyzer and the control plane verifier. The TP predictor checks $T^{TP}_{analysis} < T^{TP}_{verify}$. The RT predictor checks $DP\text{-}RT < CP\text{-}RT$, where $DP\text{-}RT$ and $CP\text{-}RT$ are the remaining times of the analyzer and verifier.

If the data plane analysis is chosen to be executed (Line 10–Line 16), *Config2Spec* invokes `PickCE` to pick the next concrete environment. It then calls `DPCompute`, to compute the forwarding graphs, and `InferPol`, to compute all policies from `cands` that hold for this environment. Afterwards, it updates the time estimates $T^{RT}_{analysis}$ (to the execution time of `DPCompute`) and $T^{TP}_{analysis}$ (to the execution time per policy which was pruned in this iteration). Then, it retains in `cands` only the policies that hold for the given environment and updates `prevEnvs` with the new environment. Finally, it checks whether there are still more concrete environments to analyze. If not, then `cands` contains only true policies, and so it sets `verified` to `cands`.

If the control plane verification is chosen (Line 18–Line 25), *Config2Spec* picks a set of policies to verify via `PickPolicies`. It then calls the verifier via `CPVerification`. The result is a counterexample `cex`, which may be $\perp$, to indicate that all policies hold, or a concrete environment if some of the policies are violated. If `cex` is $\perp$, all policies are added to `verified` and T^{TP}_{verify} is set to the ratio

of the execution time and all policies (since all have been classified). If `cex` is not $\perp$, then *Config2Spec* removes from `cands` the policies which are violated by `cex`, and sets T_{verify}^{TP} to the ratio of the execution time and violated policies (since only they are classified).

Correctness We next discuss the correctness of *Config2Spec*. First, *Config2Spec* is precise. That is, it returns *all* policies which hold and *only* the policies which hold. The correctness argument relies on the data plane analysis and control plane verification being precise with respect to their tasks: the data plane analysis returns all and only those policies which hold for the given concrete environment, while the control plane verification returns a counterexample if and only if some of the given policies do not hold. With this assumption, we can prove the invariant that *(i)* `cands` always contains the network specification (i.e., the specification is a subset of it) and *(ii)* `verified` is always contained in the network specification. Because the algorithm terminates when these sets are equal, we get the guarantee.

Second, *Config2Spec* always terminates. For this, we rely on the data plane analysis and control plane verification to always terminate. We then make the claim that at each iteration either a new concrete environment is analyzed (guaranteed by `PickCE`) or at least one policy is classified (guaranteed by the control plane verification). Since the number of concrete environments and policies is finite, at some point either all policies are classified – at which point `cands=verified` and the algorithm terminates – or all concrete environments have been analyzed – at which point, *Config2Spec* sets `verified` to `cands` (Line 16), thereby terminating the algorithm.