Incremental Network Configuration Verification

Peng Zhang
Xi’an Jiaotong University

Yuhao Huang
Xi’an Jiaotong University

Aaron Gember-Jacobson
Colgate University

Wenbo Shi
Xi’an Jiaotong University

Xu Liu
Xi’an Jiaotong University

Hongkun Yang[*]

Zhiqiang Zuo[†]
Nanjing University

ABSTRACT

Network configurations are constantly changing, and each change poses a risk of catastrophic network outages. Consequently, the networking community has put significant effort into developing and optimizing configuration verifiers. However, we observe existing configuration verifiers still have a significant drawback: they are not optimized for configuration *changes*. That is, they always check a snapshot of network configuration from scratch, even though the configuration often changes slightly since the last verification. In this paper, we demonstrate the benefits, opportunities, and challenges of *incremental network configuration verification (INCV)*. We also demonstrate the feasibility of INCV by introducing *RealConfig*, an incremental configuration verifier that can check configuration changes within one second.

CCS CONCEPTS

• **General and reference** → *Verification*; • **Networks** → *Network reliability*; *Network manageability*.

KEYWORDS

network verification, incremental verification, configuration changes

ACM Reference Format:
Peng Zhang, Yuhao Huang, Aaron Gember-Jacobson, Wenbo Shi, Xu Liu, Hongkun Yang, and Zhiqiang Zuo. 2020. Incremental Network Configuration Verification. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets ’20), November 4–6, 2020, Virtual Event, USA* . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3422604.3425936

1 INTRODUCTION

Network configurations are constantly changing to support new services, accommodate more devices, enhance network security, etc. [11, 29, 40]. For example, Facebook conducts an average of 12.5 changes per device per week in their backbone network [40]; a large online service provider conducts tens of changes per month in over half of their data center networks [19]; and two large universities each changed over a million lines of configuration over 5 years [29].

Each configuration change poses a risk of introducing catastrophic network outages [4, 30, 32, 46]. For example, configuration update errors account for 56% of network incidents in Alibaba during 2016 and 2017 [32], and a survey of over 200 network operators showed that 89% of respondents were unsure whether a change to configuration will introduce bugs [30]. Consequently, researchers and large-scale network operators have put significant effort into developing and optimizing configuration (a.k.a. control plane) verifiers [5, 7, 14, 16, 18, 26, 37, 41]. Configuration verifiers statically analyze a snapshot of a network’s current or proposed configurations to proactively identify forwarding policy violations that manifest under various network conditions—e.g., internal link failures and external route announcements.

We observe that existing configuration verifiers have a significant drawback: *they are not optimized for configuration changes*. Existing verifiers analyze snapshots of network configurations from scratch each time they change, even though configuration changes are often (relatively) small: e.g., in 85% of the data center networks operated by a large online service provider, the average change only impacts one or two devices [19], and 90% of all changes made in two large university campus networks impact a single type of configuration stanza [29]. Consequently, a significant fraction of the analysis that was performed for the previous configurations is still relevant. Without properly reusing these results, a configuration verifier will be unnecessarily slow when checking small changes in large-scale networks.

This paper asks: can we leverage the internal state and outcomes of the previous verification process to significantly speed-up the verification of configuration changes? We answer this question in the affirmative by demonstrating the benefits, opportunities, and feasibility of *incremental network configuration verification* (INCV).

Incremental verification has been extensively applied to data plane verification [25, 27, 28, 43, 47], but INCV is significantly more challenging. In particular, data plane changes (e.g., inserting and deleting forwarding rules) have simple semantics and directly affect the forwarding behavior of a single device, whereas control plane changes (e.g., adding neighbors [15] or modifying routing policies [45]) have complex semantics and can affect route selection on several, or even all, devices in the network.

Existing configuration verifiers have proposed several methods to model the relationship between configurations and forwarding

[*] Now at Google.
[†] Also with State Key Laboratory for Novel Software Technology.

HotNets ’20, November 4–6, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8145-1/20/11…$15.00
https://doi.org/10.1145/3422604.3425936

behaviors (§3.2), but making these verifiers incremental requires *determining which previous routing/forwarding state is still valid* after a configuration change, and *recomputing only the affected state.* This seems like a simple task, because distributed routing protocols are designed to explicitly signal when previous state is invalid and incrementally converge to a new state. However, for scalability reasons, configuration verifiers do not model low-level protocol messages and hence cannot leverage protocols' built-in signaling mechanisms to determine which state is still valid. Hence, we need to augment and/or redesign existing configuration verifiers to support incremental computation.

We categorize existing configuration verifiers along two dimensions (Table 1): (1) whether they *simulate* the control plane to produce concrete data planes or *analyze* the control plane to characterize the space of data planes the control plane may produce; and (2) whether they check policies using *general-purpose tools* (e.g., constraint solvers) or *domain-specific algorithms.* For each category in our taxonomy, we discuss how verifiers in that category might be modified to support INCV.

We elaborate on one of the possible designs, which uses general-purpose methods to simulate the control plane. Our focus on this design is motivated by two observations:

First, data plane verification [25, 27, 28, 43, 47] is already fast, e.g., APKeep [47] can incrementally check invariants in less than a millisecond after a rule update. Thus, we may achieve INCV by reducing it to a data plane verification problem, i.e., generating data plane updates based on configuration changes, and checking data plane updates against network policies. Clearly, the reduction should also be incremental as generating the new data plane from scratch can be quite inefficient.

Second, differential computation engines [2, 34] already enable general and automatic incremental computation. One can model a system in a declarative language, and the computation engine can effectively track and re-use state for incremental computation. Thus, we can model configuration semantics in a general way, and incrementally generate data plane updates, rather than writing customized algorithms to handle each type of configuration change.

We exploit these observations to create RealConfig, an incremental network configuration verifier. Based on large-scale synthetic networks, we find for common changes like disabling a link or changing a local preference or link cost, the time to compute data plane changes ranges from 0.12 to 0.39 seconds, which is 20× to 92× faster than computing the data plane from scratch.

The rest of this paper proceeds as follows. First, §2 demonstrates several opportunities and benefits of INCV. Next, §3 discusses the challenges and possible approaches for INCV. §4 discusses in detail one design to achieve INCV, which is embodied by our prototype system, RealConfig. §5 quantitatively demonstrates the benefits of INCV by applying RealConfig to large-scale synthetic networks. Finally, §6 discusses remaining issues, and §7 concludes.

2 MOTIVATION

In this section, we identify important tasks in network configuration management that could benefit from INCV.

Regular maintenance. Many networks experience *small, frequent configuration changes* to support new services, accommodate new customers/devices, and enhance network security. For example, two large universities change up to 55 stanzas per router per month and up to 19 stanzas per switch per month; router and switch configurations are most frequently changed to accommodate new "customers" (e.g., departments or specific users) [29]; on average, ≤ 20 lines of configuration are changed at the same time [35]. Similarly, Facebook conducts an average of 12.5 changes per device per week in their backbone and 2.5 changes per device per week in their data centers [40]; the average backbone change impacts 157 lines of configuration and the average data center change impacts 738 lines of configuration, which is relatively small compared to the scale of the backbone and data centers. Another large online service provider similarly conducts tens of configuration changes per month in over half of their data center networks, but the median change in 75% of the networks modifies only three devices' configurations [19].

Although many configuration changes are small, it is still important for network operators to verify the changed configurations satisfy basic safety policies (e.g., no loops) and avoid unintended side effects. Plankton, a state-of-the-art configuration verifier, can verify reachability under single link failures for real networks with 63 devices in just a few seconds based on packet equivalence classes (PECs) [37]. However, Plankton takes 3 minutes for synthetic fat tree topologies with 245 devices running OSPF [36]. Only re-verifying PECs that are impacted by a change, akin to realtime data plane verification [25, 27, 28, 47], could substantially reduce verification times and allow operators to receive instantaneous (e.g., on the order of seconds) feedback on changes. In turn, operators can complete routine maintenance more quickly.

Planning large-scale changes. Networks also occasionally experience large-scale changes. For example, a study of two large university networks over five years identified instances of significant network growth (e.g., a single month in which the number of routers increased by 30%), configuration clean-up (e.g., purging unused ACLs), and network-wide deployment of new functionality (e.g., IP multicast) [29]. Similarly, as part of an upgrade of Alibaba's WAN, ACLs were migrated from core routers to dedicated gateways, which required re-configuring 30% of the WAN routers [41].

Planning large-scale changes is often a difficult, time-consuming task: e.g., it took Alibaba's network operators multiple weeks to design, assess, and execute the aforementioned ACL changes [41]. INCV can greatly simplify and expedite the process: operators can plan the upgrade in small steps, and incrementally verify the (partial) plan after finishing each step. In this way, operators can rapidly detect and fix bugs, which speeds up the planning process. In contrast, if operators wait until the whole plan has been finished before verifying the configurations, it can be difficult to localize the exact changes responsible for any policy violations, thereby prolonging the planning process.

Checking configuration changes as they are designed is analogous to continuous integration (CI), a widely adopted software engineering practice in which software changes are continually validated through automated testing. Any errors uncovered by the tests are quickly brought to software engineers' attention so the errors can be corrected prior to deploying or further changing the software. CI greatly automates the process of software development, and we believe this idea can also be valuable to the process of network (upgrade) planning.

Specification mining. Specifications (i.e., policies) are a key input to network verification, but it is often hard to accurately express network specifications. A long-running network can contain very complex configurations updated by different operators [10, 11, 35]. Over time, it becomes harder to determine what network behaviors operators intend. Current approaches to specification mining require checking all possible policies, which can take a prohibitively long time: e.g., for a network consisting of 158 routers and 189 links, Config2Spec takes over 12 hours to infer all policies [12]. A major reason for the high cost is the huge space of network conditions (e.g., link failures), such that generating data planes with Batfish [1], a state-of-the-art configuration analysis tool, can take a long time. However, since each link failure only affects a small portion of the data plane, and a small number of policies, INCV can exploit the similarity among those conditions and significantly improve the speed of specification mining. Our experiments show that even without any domain-specific optimizations, incremental data plane generation for link failures is 20× faster than non-incremental data plane generation (§5).

3 CHALLENGES AND APPROACHES

Conducting INCV to simplify and/or expedite configuration management tasks is challenging. Unlike incremental data plane verification, INCV requires reasoning about complex semantics. Moreover, unlike non-incremental configuration verification, INCV requires determining which previous routing/forwarding state is still valid. This section discusses these challenges and considers various ways to address them through modifications to existing verifiers.

3.1 Challenges of INCV

Reasoning about complex semantics. Network data planes typically perform simple actions (e.g., forward, drop, rewrite) on specific packets [44]. Consequently, data plane verifiers can easily reason about network forwarding behaviors for specific equivalence classes (ECs) and only reanalyze ECs or parts of forwarding paths that are affected by data plane updates. In contrast, configurations contain a large variety of statements with complex semantics (e.g., link costs, local preferences, route aggregation, route redistribution). Furthermore, configurations on one device can affect the packet forwarding behaviors of multiple devices. This makes it difficult to determine how forwarding behaviors will be affected by a configuration change.

Determining which routing/forwarding state is still valid. Configuration changes can cause previous route advertisements, routing information base (RIB) entries, and forwarding rules (FIB entries) to change. For example, adding a route policy to assign a lower local preference to routes learned from a BGP neighbor can cause the router to change its best path, which may trigger a chain reaction of best path changes on other routers. Distributed routing protocols can incrementally converge to a new state as they explicitly signal when previous state is invalid: e.g., OSPF sends link state advertisements (LSAs) with new sequence numbers, and BGP sends route withdrawals. However, configuration verifiers do not model low-level protocol messages, due to scalability issues. Consequently, configuration verifiers cannot leverage protocols' built-in signaling mechanisms to determine which routing/forwarding state is still valid following a configuration change.

	Uses general-purpose tools	Uses domain-specific algorithms
Analyze	Minesweeper [7] Bagpipe [42]	ARC [18] Tiramisu [5] ShapeShifter [9] ERA [14]
Simulate	Batfish (original) [16] Plankton [37]	Batfish (current) [1] FastPlane [33] C-BGP [38]

Table 1: Taxonomy of existing configuration verifiers.

3.2 Approaches to INCV

We now show how existing configuration verifiers handle complex configuration semantics, and we discuss how these verifiers might be modified to determine which routing/forwarding state is still valid in order to support INCV.

We categorize existing configuration verifiers along two axes: (1) whether they *simulate* the control plane to produce concrete data planes or *analyze* the control plane to characterize the space of data planes the control plane may produce; and (2) whether they check policies using *general-purpose tools* (e.g., constraint solvers) or *domain-specific algorithms*. Table 1 summarizes our taxonomy.

Simulate the control plane using general-purpose tools. One of the first configuration verifiers, Batfish (original) [16], as well as a recently-proposed verifier, Plankton [37], adopt this approach. Batfish (original) encodes configurations and routing algorithms using LogicQL (LogicBlox's Datalog language) [6], while Plankton implements a variation of the simple path vector protocol [21] using Promela (SPIN's modeling language) [24]. Running the LogicQL program using LogicBlox or executing the Promela program using the SPIN model checker produces a data plane. The data plane is fed to a constraint solver (Z3 [13]) or custom function to determine whether the data plane satisfies the policies of interest.

Simulating the control plane has several advantages: (1) it produces a complete set of forwarding rules, enabling *any* forwarding policy to be checked post hoc; (2) simulators scale to large/complex networks better than tools that analyze the space of possible data planes [37]; and (3) as we show in §4, generating concrete data planes provides an opportunity to take advantage of innovations in data plane verification. Similarly, using general-purpose tools allows this category of configuration verifiers to take advantage of existing optimization techniques in model checkers (e.g., SPIN) and constraint solvers (e.g., Z3).

However, making such verifiers incremental requires a Datalog engine or explicit-state model checker capable of incremental computation. In particular, such a tool must be able to deduce which program state—e.g., route advertisements, RIB entries, and forwarding rules—is no longer valid and which new computations must be performed. To the best of our knowledge, no existing explicit-state model checkers possess such capabilities, but we show in §4 that recent advances in differential computation [34] give rise to a Datalog engine [2] supporting incremental computation.

Simulate the control plane using domain-specific algorithms. To take advantage of domain-specific optimizations, the current version of Batfish [1], as well as FastPlane [33] and C-BGP [38], use custom algorithms to simulate the control plane. These algorithms exploit the characteristics of network control planes to more efficiently simulate the control plane. For example, FastPlane employs a generalized form of Dijkstra's algorithm for monotonic networks where preference of routes decrease during route propagation.

However, making such configuration verifiers incremental requires customizing the simulation algorithms to reuse part of the state from a prior invocation. Different configuration changes—e.g., changing a link cost versus enabling route redistribution—can have vastly different impacts on the control plane—e.g., re-computation of shortest paths versus the introduction of additional routes—so we must customize algorithms for each type of configuration change to support incremental verification.

Analyze the space of possible data planes using general-purpose tools. Configuration verifiers like Minesweeper [7] and Bagpipe [42] use general-purpose tools to analyze the space of possible data planes the control plane may generate. In particular, they model configurations, routing algorithms, and policies of interest using a system of satisfiability modulo theories (SMT) constraints, which can be checked with off-the-shelf SMT solvers (e.g., Z3).

In contrast to simulation tools, analysis tools avoid the need to iteratively explore every possible network condition (e.g., every combination of link failures). Additionally, as noted above, using general purposes tools allows these verifiers to take advantage of existing optimization techniques in constraint solvers.

Making such configuration verifiers incremental requires a means to incrementally explore the search space—i.e., the space of possible data planes. Current SMT solvers offer multiple options for incremental solving [3], each of which may be amenable to certain types of configuration changes: e.g., disabling filtering rules can be achieved through assumption-based solving. However, further innovations in control plane modeling and/or SMT solving are necessary to fully achieve INCV using constraint-based approaches: e.g., changing a link cost requires modifying a constraint, which is not supported by assumption- or stack-based incremental solving.

Analyze the space of possible data planes using domain-specific algorithms. Lastly, several configuration verifiers analyze the space of possible data planes using domain-specific algorithms. ARC [18] and Tiramisu [5] model the control plane as a graph and use polynomial-time graph algorithms (e.g., max-flow) and custom integer linear programs to analyze the space of possible data planes and verify policy compliance. ShapeShifter [9] models the control plane using routing algebra [22, 39] and uses abstract interpretation to check forwarding policies. These domain-specific optimizations allow these configuration verifiers to scale better than control plane analysis tools that rely on constraint solvers.

However, making such verifiers incremental requires customizing each of the domain-specific algorithms these verifiers use to analyze the space of possible data planes. For example, ARC would need to employ incremental depth-first search, max-flow, and shortest-path algorithms to support incremental verification of forwarding policies. BFA [31] introduces a method to make the max-flow algorithm used by ARC incremental. However, BFA only supports incremental verification of specific properties (k-reachability) after specific changes (insertion and deletion of links).

In the next section, we choose one of these points in the design space to demonstrate the feasibility of INCV.

4 ACHIEVING INCV

As demonstrated above, there are multiple possible avenues for achieving INCV. This paper focuses on one such direction: *using general-purpose tools to explicitly generate the data plane for verification.* Using general-purpose tools avoids the need to write customized algorithms for different types of configuration change, while explicitly generating data planes allows a diverse set of debugging functionalities like dumping the full packet traces (what rules they match, which path they take, etc.)

4.1 Key Enablers

A design that uses general-purpose tools to incrementally simulate and verify the control plane is made possible by recent advances in differential computation and realtime data plane verification.

Leveraging differential computation for incremental data plane generation. As discussed in §3.1, the re-convergence of routing protocols after a change in configuration or network conditions is inherently incremental. We observe differential computation, e.g., Differential Dataflow [34], can effectively track and re-use state to automate incremental re-convergence. This computation model enables us to incrementally generate data planes in a general way, rather than write customized algorithms for each type of change.

Leveraging realtime data plane verification for incremental policy checking. We argue that the data plane after a small configuration change will not differ significantly. For example, when the local preference is changed for an IP prefix received from one interface, only the best routes falling within this IP prefix received from the interface may change. Since realtime data plane verifiers like APKeep [47] can check a data plane update in a millisecond or less, we can quickly check the data plane change, which is a batch of rule updates, using realtime data plane verifiers. In our design, we break the data plane verification process into two stages: updating the data plane model, and checking policies. Existing incremental techniques mostly focus on how to make the first stage incremental, but we show there are also opportunities to check policies incrementally.

4.2 Our Design: RealConfig

Based on the above insights, we introduce our design, *RealConfig*, for achieving INCV. Our design (Figure 1) consists of three components chained in sequence. Each component operates incrementally.

Incremental data plane generator takes the configuration changes as input, and returns the data plane changes. Configuration changes consist of insertions or deletions of configuration lines, and data plane changes consist of insertions or deletions of forwarding, filtering, or rewriting rules.[1] Since packet filtering rules are explicitly specified in configuration files, we can directly extract filtering rule changes from the configuration changes. However, we must incrementally generate the forwarding rules based on configuration changes and protocol-specific routing algorithms.

[1] Modifications can be seen as deleting an old line/rule and inserting a new line/rule.

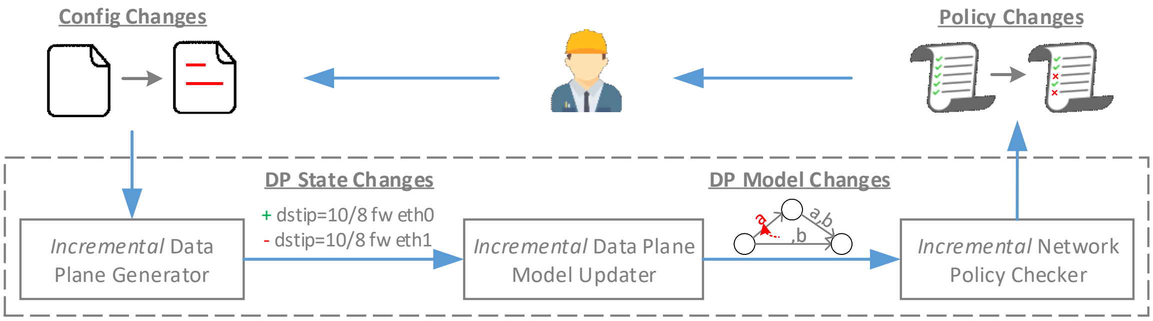
Config Changes
Policy Changes
DP State Changes
DP Model Changes
+ dstip=10/8 fw eth0
- dstip=10/8 fw eth1
a
a,b
,b
Incremental Data Plane Generator
Incremental Data Plane Model Updater
Incremental Network Policy Checker

Figure 1: The workflow of incremental network configuration verification in RealConfig.

RealConfig performs the incremental forwarding rule computations using Differential Datalog (DDlog) [2]. DDlog allows programmers to write programs in a dialect of Datalog, and synthesizes an incremental implementation running on top of Differential Dataflow (DD) [34]. We use DDlog instead of the native language of DD, because DDlog offers useful high-level abstractions. For example, DDlog offers bitvector type which can be used for declaring IP prefixes. Currently, we model a basic set of configurations including OSPF, BGP, static routes, access control lists, and route redistribution. Other routing protocols can also be easily integrated due to the generality of our modeling method.

Incremental data plane model updater takes the data plane changes as input, and outputs the changes to the data plane model. Here, the data plane model should be able to describe how each packet is forwarded in the network. Consequently, changes in the data plane model should describe what ECs are affected, and what the old and new forwarding behaviors are for each affected EC. State-of-the-art realtime data plane verifiers like APKeep [47] incrementally maintain a data plane model, and can be adapted for use here. However, data plane verifiers are designed for checking single-rule updates—i.e., for each rule update they update the model and check policies—while we want to update the data plane model according to a batch of rule updates (without checking polices between each individual update). The reason for not checking policies for each rule update is that this only captures some potential transient failures, which are of less interest since we focus on checking the correctness of the converged data plane state (e.g., FIB).

RealConfig uses a modified version of APKeep. We choose APKeep because it can incrementally maintain the minimum number of ECs, which makes it more scalable than other data plane verifiers, especially when there are rules with multiple matching fields (e.g., ACL rules). APKeep models the forwarding behaviors of ECs by maintaining a set of logical ports (encoding a specific forwarding action) for each device, and a map from each port to the set of ECs that forwarded to this port (taking the action of the port). We extend APKeep to work in *batch mode*: given a batch of rule updates, RealConfig determines an order of rule updates, and invokes the model update algorithm of APKeep for each rule update according to this order. As we show in §5, the update order can significantly affect the update speed; we leave the optimal scheduling of model updates as future work. After updating the model, RealConfig outputs all the affected ECs and their old and new forwarding behaviors, i.e., the previous and current ports of the affected ECs.

Incremental network policy checker takes the data plane model changes as input, and outputs changes in policy satisfaction. Changes in policy satisfaction includes policies that become violated and policies that become satisfied following a configuration change. The latter helps operators test whether a repair plan works. The policies include both network invariants, e.g., loop-freedom, blackhole-freedom, and operator intent, e.g., reachability, waypoint, load balance, etc. A reachability policy specifies what packets can traverse between two end points, e.g., "only HTTP traffic should be allowed between subnet A and subnet B".

Compared to universal invariants (e.g., loop-freedom), policies like reachability only "register" to a small set of packets. For example, each reachability policy is only related to some packets, e.g., HTTP packets. Therefore, there are opportunities to incrementally check policies: i.e., *only check policies related to the affected ECs*. This can significantly improve the efficiency of policy checking, because real networks can have a large number of policies [12].

To incrementally check all-pairs reachability, RealConfig tracks the relationship between ECs, node pairs, and forwarding paths with two maps: (1) a map from each EC to the set of paths the EC traverses; (2) a map from each node pair (s, d) to the ECs that can be sent from s to d. After receiving the affected ECs from the model updater, RealConfig identifies all affected paths and modifies these paths according to their new forwarding behaviors. Then, RealConfig identifies the pairs affected by the modified paths (based the end points of the paths), and updates the ECs of those pairs. By checking all-pair reachability, RealConfig can output each node pair (s, d) whose set of ECs changes.

5 PRELIMINARY RESULTS

Setting. We use a fat tree topology with 180 nodes and 864 links. We run OSPF or BGP on this topology. For BGP, each node is a different AS, and establishes peer relation with all its neighbors. We make three types of changes to the configuration of each node: (1) LinkFailure: failing a link by deactivating the corresponding interface; (2) LC: changing the OSPF link cost of one interface from 1 to 100 (less preferred); (3) LP: changing the BGP local preference for routes received at one interface from 100 to 150 (more preferred). The experiments are run on a server with two 12-core Intel Xeon CPUs @ 2.3GHz (a single core is used) and 256G memory.

Data plane generation. We compute the data plane state with the most-recent version of Batfish [1] and RealConfig from scratch, and

Table 2: Average data plane generation time for the fat tree network. LinkFailure means failing a link by deactivating an interface; LC/LP means changing link cost or local preference depending on whether it is OSPF or BGP.

Protocol	Batfish	RealConfig		
	Full	Full	LinkFailure	LC/LP
OSPF	7.13s	36.11s	0.39s (1.1%)	0.39s (1.1%)
BGP	3.81s	3.92s	0.19s (4.8%)	0.12s (3.1%)

Table 3: Model update and property checking for the fat tree network running BGP. #Rules: number of affected rules; #ECs: number of affected ECs; #Pairs: number of affected node pairs; T1: model update time; T2: policy checking time.

Change	#Rules	Order	#ECs	T1	#Pairs	T2
LinkFailure	+26/-28	+, -	28	3ms	286/10224	58ms
	(0.32%)	-, +	54	10ms	(2.79%)	
LP	+54/-54	+, -	54	6ms	132/10224	61ms
	(0.64%)	-, +	108	20ms	(1.29%)	

then make the three types of changes and incrementally generate the data plane state with RealConfig. Table 2 shows that Batfish is faster than RealConfig for from-scratch full computation. However, RealConfig is much faster for configuration changes: even without domain-specific optimizations, the incremental computation time is only 1% to 6.5% of the full computation time of RealConfig. Note Plankton uses 41.2 seconds[2] for OSPF on the same size of topology [37], which is ≈ 2 orders of magnitude slower than RealConfig. The above results demonstrate the feasibility of using a general-purpose computation engine to incrementally generate data plane state.

Model update. Column 2 of Table 3 shows that less than 1% of all forwarding rules are affected by the configuration changes, for the fat tree topology running BGP. We try two simple orders to update the model, i.e., insertion-first and deletion-first. From Column 4 and 5, we can see the number of affected ECs and the time to update the data plane model are heavily related to the update order. The reason is that insertion-first will make the model updater directly modify the forwarding behavior of ECs, i.e., moving them from old ports to new ports; while deletion-first will make the model updater first move them from the old ports to a special "drop" port (since the packets belonging to the ECs will be dropped after the rule is deleted), and then from the "drop" port to the new ports. The update time is less than 10ms for both configuration changes if we apply insertion updates first.

Policy checking. Column 6 and Column 7 of Table 3 report the number of affected node pairs, and the time for checking these pairs, respectively. We can see that only 2.79% and 1.29% of all pairs are affected for link failure and local preference change, respectively. As a result, the time of policy checking is only ~60ms, and the overall data plane verification time (model update and policy checking) is less than 100ms. This demonstrates the benefits of incrementally checking affected policies.

6 DISCUSSION

Nontermination of Datalog evaluation. The evaluation of our Datalog model may never terminate, i.e., looping forever without reaching any fixed-point. This often reveals some unexpected bugs, e.g., when BGP is misconfigured and cannot converge [23], or when BGP has multiple converged states which can lead to route update racing [45]. To detect these bugs without waiting until Datalog times out, we need a way to detect the recurring state, i.e., a state that has been reached before during Datalog evaluation; we leave this as future work.

Domain-specific optimization. Currently, RealConfig does not leverage any previously-proposed domain-specific optimizations, such as leveraging symmetries in the network topology and configurations to reduce the size of the network model [8, 20], or leveraging the independence of different ECs to parallelize verification [17, 37]. Such optimizations can also be applied to RealConfig to further improve its speed; we leave this as future work.

Alternative approaches to INCV. In §3.2 we discussed how various categories of verifiers might be modified to support INCV, and in §4 we presented a detailed design based on one of these approaches. However, several of the other approaches discussed in §3.2 also have significant potential: e.g., as a framework based on Tiramisu [5] that uses incremental graph algorithms, or a framework inspired by Minesweeper that uses a control plane encoding that is more amenable to the current incremental solving capabilities of SMT solvers. We believe exploring these alternatives is important future work for the research community.

7 CONCLUSION

Conducting incremental network configuration verification (INCV) by (partially) reusing the internal state and outcomes of previous verification processes is an important, yet previously untapped, opportunity for efficiently validating configuration changes. However, leveraging this opportunity requires augmenting and/or redesigning existing configuration verifiers to support incremental computation. In particular, advances must be made in model checkers, constraint solvers, computation engines, and/or domain-specific algorithms to intelligently determine which previous conclusions are still valid. Our prototype system, RealConfig, demonstrates the feasibility of one point in the design space that leverages recent advances in incremental computation and realtime data plane verification. Our experiments show that INCV can speed-up data plane computation time by 20× to 92× compared to computing the data plane from scratch. In the future we plan to (1) integrate domain-specific optimizations and recurring state detection into RealConfig, (2) conductive experiments on real network configuration updates, and (3) explore alternative approaches to INCV to fully realize the potential of INCV.

ACKNOWLEDGMENTS

We would like to thank Leonid Ryzhyk for help on optimizing the DDlog program, and all the anonymous reviewers for their valuable comments. This work is supported by the National Natural Science Foundation of China (No. 61772412, 61802168) and the National Science Foundation (No. 1763512).

[2] The experiments run on a single core of an Intel Xeon CPU @ 3.4GHz, and the time includes policy checking.

REFERENCES

[1] [n.d.]. Batfish. https://github.com/batfish/batfish.

[2] [n.d.]. Differential Datalog (DDlog). https://github.com/vmware/differential-datalog.

[3] [n.d.]. How incremental solving works in Z3? https://stackoverflow.com/questions/16422018/how-incremental-solving-works-in-z3.

[4] [n.d.]. Human Factors Are Causing Most Of Today's Network Outages And Vulnerabilities. https://tinyurl.com/ya7qy7nm.

[5] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast and General Network Verification. In *USENIX NSDI*.

[6] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *ACM SIGMOD*.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *ACM SIGCOMM*.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2018. Control plane compression. In *ACM SIGCOMM*.

[9] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2020. Abstract interpretation of distributed network control planes. In *ACM POPL*.

[10] Theophilus Benson, Aditya Akella, and David Maltz. 2009. Unraveling the Complexity of Network Management. In *USENIX NSDI*.

[11] Theophilus Benson, Aditya Akella, and Aman Shaikh. 2011. Demystifying Configuration Challenges and Trade-offs in Network-based ISP Services. In *ACM SIGCOMM*.

[12] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. 2020. Config2Spec: Mining Network Specifications from Network Configurations. In *USENIX NSDI*.

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[14] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient network reachability analysis using a succinct control plane representation. In *USENIX OSDI*.

[15] Nick Feamster and Hari Balakrishnan. 2005. Detecting BGP Configuration Faults with Static Analysis. In *USENIX NSDI*.

[16] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A general approach to network configuration analysis. In *USENIX NSDI*.

[17] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. 2017. Automatically repairing network control planes using an abstract representation. In *ACM SOSP*.

[18] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*.

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

[20] Nick Giannarakis, Ryan Beckett, Ratul Mahajan, and David Walker. 2019. Efficient verification of network fault tolerance via counterexample-guided refinement. In *International Conference on Computer Aided Verification*.

[21] Timothy G Griffin, F Bruce Shepherd, and Gordon Wilfong. 2002. The stable paths problem and interdomain routing. *IEEE/ACM Transactions On Networking* 10, 2 (2002), 232–243.

[22] Timothy G Griffin and Joäo Luís Sobrinho. 2005. Metarouting. In *ACM SIGCOMM*.

[23] Timothy G Griffin and Gordon Wilfong. 1999. An analysis of BGP convergence properties. In *ACM SIGCOMM*.

[24] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (1997), 279–295.

[25] Alex Horn, Ali Kheradmand, and Mukul R Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms. In *USENIX NSDI*.

[26] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C. Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating datacenters at scale. In *ACM SIGCOMM*.

[27] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *USENIX NSDI*.

[28] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. 2013. Veri-Flow: Verifying network-wide invariants in real time. In *USENIX NSDI*.

[29] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. 2011. The evolution of network configuration: a tale of two campuses. In *ACM SIGCOMM IMC*.

[30] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable dynamic network control. In *USENIX NSDI*.

[31] Yifan Li, Jake Jia, Xiaohe Hu, and Jun Li. 2019. Real Time Control Plane Verification. In *Proceedings of the ACM SIGCOMM 2019 Workshop on Networking and Programming Languages*. 2–2.

[32] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. Automatic life cycle management of network configurations. In *SIGCOMM Workshop on Self-Driving Networks*.

[33] Nuno P Lopes and Andrey Rybalchenko. 2019. Fast BGP simulation of large datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*.

[34] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*.

[35] David Plonka and Andres Jaan Tack. 2009. An Analysis of Network Configuration Artifacts. In *Proceedings of the 23rd Large Installation System Administration Conference*.

[36] Santhosh Prabhu. [n.d.]. Personal communication.

[37] Santhosh Prabhu, Kuan-Yen Chou, Ali Kheradmand, P Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*.

[38] Bruno Quoitin and Steve Uhlig. 2005. Modeling the routing of an autonomous system with C-BGP. *IEEE Network* 19, 6 (2005), 12–19.

[39] Joao L Sobrinho. 2005. An algebraic theory of dynamic network routing. *IEEE/ACM Transactions on Networking* 13, 5 (2005), 1160–1173.

[40] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H. Y. Wong, and Hongyi Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *ACM SIGCOMM*.

[41] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. 2019. Safely and automatically updating in-network ACL configurations with intent language. In *ACM SIGCOMM*.

[42] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM OOPSLA*.

[43] Hongkun Yang and Simon S Lam. 2013. Real-time verification of network properties using Atomic Predicates. In *IEEE ICNP*.

[44] Hongkun Yang and Simon S Lam. 2017. Scalable verification of networks with packet transformers using atomic predicates. *IEEE/ACM Transactions on Networking* 25, 5 (2017), 2900–2915.

[45] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, et al. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *ACM SIGCOMM*.

[46] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Automatic Test Packet Generation. In *ACM CoNEXT*.

[47] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *USENIX NSDI*.