Network Can Help Check Itself: Accelerating SMT-based Network Configuration Verification Using Network Domain Knowledge

Xing Fang†, Feiyan Ding†, Bang Huang†, Ziyi Wang†, Gao Han†, Rulan Yang†,
Lizhao You⋆, Qiao Xiang†, Linghe Kong♢, Yutong Liu♢, Jiwu Shu†‡,
†Xiamen Key Laboratory of Intelligent Storage and Computing, School of Informatics, Xiamen University,
⋆School of Informatics, Xiamen University, ♢Shanghai Jiao Tong University, ‡Minjiang University

***Abstract*—Satisfiability Modulo Theories (SMT) based network configuration verification tools are powerful tools in preventing network configuration errors. However, their fundamental limitation is efficiency, because they rely on generic SMT solvers to solve SMT problems, which are in general NP-complete. In this paper, we show that by leveraging network domain knowledge, we can substantially accelerate SMT-based network configuration verification. Our key insights are: given a network configuration verification formula, network domain knowledge can (1) guide the search of solutions to the formula by avoiding unnecessary search spaces; and (2) help simplify the formula, reducing the problem scale. We leverage these insights to design a new SMT-based network configuration verification tool called NetSMT. Extensive evaluation using real-world topologies and synthetic network configurations shows that NetSMT achieves orders of magnitude improvements compared to state-of-the-art methods.**

I. Introduction

Network misconfigurations are prevalent in all types of networks (*e.g.*, enterprise networks, wide area networks, and data center networks). They could cause network errors such as forwarding loops, blackholes, and waypoint violations and lead to disastrous financial and social consequences [1]–[4]. A major advance to prevent such misconfigurations is network configuration verification (also called control plane verification interchangeably, or CPV for short). It analyzes the configuration files of network devices [5]–[16] to determine whether these configurations would compute forwarding tables conforming to pre-specified network invariants (*e.g.*, reachability, waypointing, and loop-freeness) had they been deployed on the network devices. Among various CPV tools, SMT-based tools [5], [7], [17] are advantageous over simulation-based [6], [8], [9], [14], [15] and graph-based [12], [13] tools in terms of capability, because they support verifying diverse routing protocols in all possible converged states. Specifically, they encode the conjunction of network configurations and the negation of network invariants as an SMT formula (we call it the *verification formula*), and use off-the-shelf SMT solvers (*e.g.*, [18], [19]) to examine its satisfiability. A satisfiable solution to the verification formula indicates that the network configurations are erroneous.

SMT-based CPV tools are inefficient. Despite the capability advantage, these SMT-based tools suffer from low performance efficiency. Specifically, deciding the satisfiability of an SMT formula is NP-complete. When the size of the network (in terms of the number of nodes and links) or network configurations (in terms of configuration lines) increases, the scale of the corresponding verification formula also increases, resulting in an exponentially increasing latency to verify the correctness of network configurations. As a result, it is difficult to deploy SMT-based CPV in large-scale production networks. For example, Minesweeper [7] cannot verify k-failure-reachability within 1 hour for a 125-node network where k is set to 3.

Fundamental reason for low efficiency: ignorance of network domain knowledge. First, SMT-based CPV tools rely on off-the-shelf SMT solvers, which ignore the control flow information in network configurations, leading to unnecessary testing on assignments with the same satisfiability. For example, consider a BGP route policy matching the destination IP prefix of received BGP route announcements and setting n attributes of matched announcements. For any announcement, if it matches the policy, all its n attributes will be set according to the configuration. Situations such as an IP-matched announcement with m ($m < n$) attributes set will not happen and hence should not be explored during SMT solving. However, SMT solvers are unaware of such information and may have to explore up to $2^n - 1$ redundant situations. Second, these tools ignore the context of network configurations and invariants and use a fixed encoding to construct the verification formula. For example, BGP local preferences are encoded as 16-bit integers in these tools. However, if a set of configurations only sets four distinct values of local preferences, we can use 2-bit unsigned bit vectors to encode local preferences, substantially reducing the scale of the verification formula.

Some studies [10], [17], [20], [21] have attempted to leverage network domain knowledge to tackle the efficiency issue of SMT-based network configuration verification. However, they are point solutions whose applicability is tightly coupled with different assumptions and perform poorly when verifying more complex invariants (*e.g.*, reachability under k-link-failure). For example, Bonsai [10] leverages the symmetry of data center network topologies to compress the verification formula into a smaller one, but the conditions of compression highly rely on topology symmetry. Kirigami [20] divides the network into several partitions to verify each partition with a smaller verification formula using an assume-guarantee framework, but its performance relies on a manual, experience-driven partitioning of the network and cannot verify invariants

Lizhao You and Qiao Xiang are co-corresponding authors.

under k-link-failure. Lightyear [21] leverages the monotonicity to semi-automate the assume-guarantee framework for BGP configuration verification. However, it may have false positive verification results due to model over-approximation. BiNode [17] leverages the Gao-Rexford condition in typical interdomain BGP policies to simplify the verification formula by reducing variable dependencies, but it has limited effects on configurations not conforming to the G-R condition.

In this paper, we systematically investigate the important problem of how to leverage network domain knowledge to accelerate SMT-based network configuration verification in generic large-scale networks. A fast SMT-based CPV tool can quickly find configuration errors in large networks under all converged states, improving the reliability of networks and the efficiency of network operation and maintenance.

To this end, we design NetSMT, a network-aware SMT-based network configuration verification tool. Instead of continuing to search for accelerating methods for specialized settings (*e.g.*, topologies, configurations, and protocols), we aim to make full use of network domain knowledge in verifying generic networks. In particular, as depicted in Fig. 1, NetSMT presents a new *planner* block for arranging SMT solving order (Section III) and redesigns the *encoder* block for generating SMT formulas (Section IV) to accelerate CPV. The design of these two blocks is based on the following two key insights:

Key insight 1: network domain knowledge allows a more efficient search of a satisfiable solution in the verification formula. Modern SMT solvers (*e.g.*, Z3 [18] and CVC5 [19]) use the DPLL(T) algorithm to decide the satisfiability of a given SMT formula. However, DPLL(T) treats all the variables in the formula as independent ones and may have to explore the combinations of all possible variable assignments, a huge search space, before finding a satisfiable solution. In contrast, given a network verification formula, network domain knowledge allows us to divide this huge search space into a smaller number of equivalent classes, where variable assignments in the same class have the same SMT satisfiability. As such, it could substantially prune the search space and improve the efficiency of SMT-based CPV.

To leverage this insight, before the exploration, NetSMT first systematically arranges the exploration order of variables in the network verification formula to avoid exploring variable assignments in the same equivalent class as much as possible. Specifically, given a formula, before the exploration starts, NetSMT arranges branching variables (*e.g.*, variables corresponding to matching conditions in route policies) to be explored firsts. Within the set of branching variables, NetSMT arranges the ones corresponding to configurations in routers closer to the destination to be explored first. Second, during the exploration, NetSMT leverages the intent of operators to first explore assignments that are more likely to be satisfiable (*i.e.*, network error). With these two designs, NetSMT can substantially improve the efficiency of solving the network verification formula over off-the-shelf SMT solvers.

Key insight 2: network domain knowledge allows a more succinct verification formula, accelerating verification by reducing the problem scale. On one hand, the time complexity of SMT solving is exponential to the scale of SMT formulas. On the other hand, network configurations can be complex but redundant (*e.g.*, a route policy matching on an obsolete IP prefix due to configuration updates). As a result, if we can prune such redundant configurations while not affecting the correctness of the verification, we can build a network verification formula with fewer variables and constraints and hence further improve the efficiency of SMT-based CPV tools.

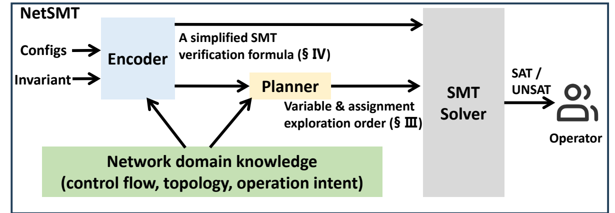


Fig. 1. The architecture and workflow of NetSMT

To leverage this insight, during the construction of the network verification formula, NetSMT first scans the configuration files to prune configurations that are unrelated to the network invariant. For the remaining configurations, NetSMT adopts abstract interpretation to lift the level of abstraction of long-width configuration variables (*e.g.*, BGP local preference) to short-width, abstract configuration variables to reduce the size of variables in the formula. In contrast to Shapeshifter [11], a simulation-based verification tool that sacrifices correctness for efficiency, we prove that our lifting design does not affect the correctness of the verification result.

Evaluation (Section V). We implement a prototype of NetSMT and make it open-source [22]. We evaluate its performance extensively using real-world topologies and synthetic configurations. NetSMT remarkably outperforms Minesweeper [7] and Binode [17].

II. Motivation

In this section, we provide an overview of SMT-based network configuration verification tools and identify the fundamental reason for their low efficiency. We then elaborate on our key insights on how to leverage network domain knowledge to accelerate SMT-based CPV.

A. Background and Fundamental Issue

SMT-based network configuration verification. Such tools [5], [7], [17] encode the configuration files of routers as an SMT formula N and the network invariant (*e.g.*, reachability, loop-freeness, and waypointing) to be verified as another SMT formula P. They then verify whether the configuration files can guarantee the network invariant by checking whether $N \implies P$ is a tautology, which is equivalent to checking whether $N \wedge \neg P$ is unsatisfiable. If $N \wedge \neg P$ is satisfiable, it means the configuration files cannot guarantee the given network invariant. Otherwise, the configuration files are correct in upholding the given network invariant.

An example network. Consider a network of four routers running BGP in Fig. 2. Router D is connected to two subnets, 10.0.0.0/24 and 10.1.0.0/24. Suppose the network invariant to be verified is S does not reach 10.1.0.0/24. The key

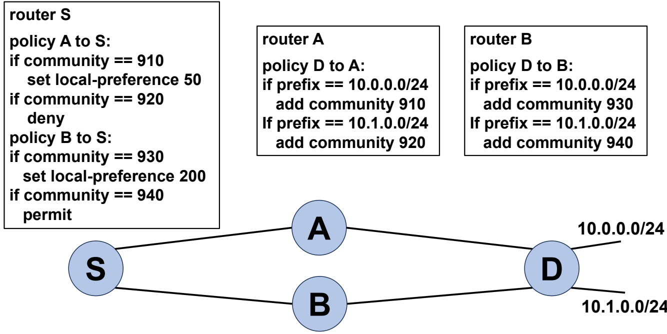


Fig. 2. Example network.

```
1. if out_DA.valid == true
2. then
3.   if failed_A,D == 0
4.   then
5.     if out_DA.prefix == 10.0.0.0/24
6.     then
7.       in_DA.valid = true
8.       in_DA.comm910 = true
9.       in_DA.comm920 = out_DA.comm920
10.      in_DA.comm930 = out_DA.comm930
11.      in_DA.comm940 = out_DA.comm940
12.      ...
13.    else if out_DA.prefix == 10.1.0.0/24
14.    then
15.      in_DA.valid = true
16.      in_DA.comm910 = out_DA.comm910
17.      in_DA.comm920 = true
18.      in_DA.comm930 = out_DA.comm930
19.      in_DA.comm940 = out_DA.comm940
20.      ...
21.    else in_DA.valid = false
22.  else in_DA.valid = false
23. else in_DA.valid = false
```

Fig. 3. SMT encoding for the import policy of router A.

configuration snippets of the routers are shown in the figure. Specifically, A and B attach different BGP community tags to route announcements they receive from D based on the IP prefixes of these announcements. S sets different local preferences to route announcements it receives from A and B or filters them based on their attached community tags.

Step 1: Encoding network configurations. To construct the configuration SMT formula N, SMT-based configuration verification tools model each router's sent and received route messages (*e.g.*, BGP route announcements and OSPF link state announcements) and its selected route as symbolic records, and model how configuration files process these symbolic records using predicate logic. Fig. 3 illustrates how they model the configuration of router A in Fig. 2. out_{DA} is the symbolic BGP route announcement D sends to A. If out_{DA} satisfies some basic conditions (*i.e.*, it is a valid message conforming to D's export policy configurations and link (D, A) is up), router A processes it based on A's import policy configurations to generate another symbolic record in_{DA}. For example, when the destination IP prefix of out_{DA} is 10.0.0.0/24, a Boolean variable $in_{DA}.valid$ is set to true. $in_{DA}.comm910$ is also set to true, indicating that A will attach a community tag of value 910 to in_{DA}. The corresponding Boolean variables indicating the validness of all other possible values of the community tag of in_{DA} (*i.e.*, $in_{DA}.comm920$, $in_{DA}.comm930$ and $in_{DA}.comm940$) are set to be the same as those of out_{DA}, which are false because the configuration of D does not attach any community tag to any route announcement.

Step 2: Encoding the network invariant. An invariant SMT formula P is usually composed of Boolean variables. Consider the invariant in our example (*i.e.*, S cannot reach subnet 10.1.0.0/24). It is represented by $\neg canReach^1_S$, where $canReach^1_S$ is a Boolean variable representing the reachability from S to 10.1.0.0/24 and rewritten as:

$$\mathrm{canReach}^1_{\mathrm{S}} \Longleftrightarrow \bigvee_{R\in\{A,B\}} \left(\mathrm{datafwd}^1_{\mathrm{S,R}} \wedge \mathrm{canReach}^1_{\mathrm{R}}\right), \quad (1)$$

which means that S can reach 10.1.0.0/24 if and only if it forwards traffic destined to 10.1.0.0/24 to at least one neighbor that can reach 10.1.0.0/24. $datafwd^1_{S,R}$ is another Boolean variable that is true if and only if S selects R as the next hop to 10.1.0.0/24, and the access control list (ACL) of R does not block traffic to 10.1.0.0/24, where R is A or B.

Step 3: Solving $N \wedge \neg P$ using an SMT solver. SMT-based configuration verification tools rely on off-the-shelf SMT solvers (*e.g.*, Z3 [18] and CVC5 [19]), which use the DPLL(T) algorithm [23] to decide the satisfiability of a given SMT formula. Fig. 4 shows its basic idea. In a nutshell, DPLL(T) is composed of a Boolean satisfiability (SAT) solver [24]–[26] and a theory solver. The latter is a procedure that takes as input a collection of propositions and decides whether they are satisfiable under a pre-defined theory (*e.g.*, whether $(x + 1 < 0) \wedge (x > 0)$ where x is an integer is satisfiable under the linear integer arithmetic theory).

Given a network verification formula $N \wedge \neg P$, during initialization, DPLL(T) adopts the Tseitin's transformation method [27] to transform it into Ψ, an equivalent SMT formula in conjunctive normal form (CNF), and replaces all atoms in Ψ with Boolean variables to get an SAT formula $\mathcal{B}(\Psi)$ called the *Boolean abstraction*. It then uses the SAT solver to find $\mathcal{M}$, a partial satisfiable assignment to $\mathcal{B}(\Psi)$ (*i.e.*, after the assignment, all clauses in $\mathcal{B}(\Psi)$ are either true or unknown, but not false). Next, in each iteration, DPLL(T) operates in two steps. First, it uses the theory solver to examine whether the propositions on the atoms in Ψ corresponding to $\mathcal{M}$ is theory-satisfiable. Second, if so, it uses the SAT solver to update $\mathcal{M}$ by assigning as many unassigned variables in $\mathcal{B}(\Psi)$ as possible while keeping $\mathcal{M}$ partially satisfiable. Otherwise, it updates $\mathcal{M}$ by backtracking its assignment history to find a new partially satisfiable assignment. The algorithm stops when (1) $\mathcal{B}(\Psi)$ is found unsatisfiable, indicating $N \wedge \neg P$ is also unsatisfiable and the configurations are correct, or (2) $\mathcal{M}$ becomes a complete satisfiable assignment (*i.e.*, all clauses become true) and its corresponding propositions are theory-satisfiable, indicating $N \wedge \neg P$ is satisfiable and the configurations are erroneous. Fig. 4 gives a simple example to illustrate DPLL(T). We refer readers to [23] for a comprehensive tutorial on DPLL(T).

Issue: Ignoring network domain knowledge may result in low efficiency of SMT-based CPV. Specifically, this impact acts in two aspects. First, the DPLL(T) algorithm used in off-the-shelf SMT solvers ignores the control flow information in network configurations, *leading to redundant explorations on similar assignments of $\mathcal{B}(\Psi)$ with the same satisfiability*. To be concrete, consider an even simpler example of one router in Fig. 5. The CNF formula Ψ of these 6 statements and one invariant has 7 atoms and its Boolean abstraction has 7 Boolean variables. The last variable $v_{in_1.med<50}$ is always false because it corresponds to the assertion $\neg(in_1.med < 50)$. DPLL(T) treats the first 6 variables as independent ones. If

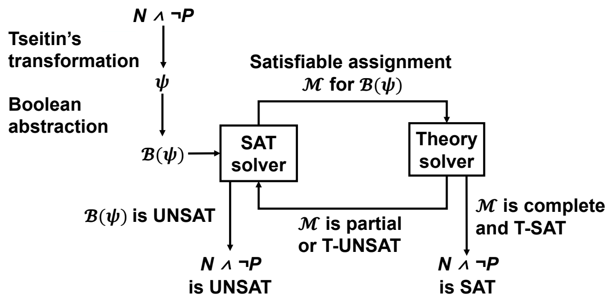


Fig. 4. Basic idea of DPLL(T).

1.	if (¬fail$_1$)	v1
2.	in$_1$.valid = true	v2
3.	in$_1$.lp = 10	v3
4.	in$_1$.comm1 = true	v4
5.	in$_1$.comm2 = false	v5
6.	else	
7.	in$_1$.med = 100	v6
8.	assert(in$_1$.med<50)	v7
	[SMT encoding of configurations]	[variables of Boolean abstraction]

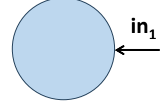


Fig. 5. SMT encoding and variables of Boolean abstraction of one router

DPLL(T) explores the value of v_2 through v_6 before exploring v_1, and tries to first assign false to variables during the exploration, it needs to explore all $2^6 = 64$ assignments of $\mathcal{B}(\Psi)$ and uses the theory-solver to examine each assignment.

In contrast, the control flow information decides that the 64 assignments of the Boolean variables in $\mathcal{B}(\Psi)$ can be categorized into only 4 *equivalent classes* based on the value assignment of the v_1, the Boolean variable corresponding to the branching condition $failed_1 == false$: (1) one containing 2 assignments where the five variables are all set to true; (2) one containing $((2^4 - 1) * 2) = 30$ assignments where the v_1 is set to true and at least one of the variables from v_2 to v_5 is set to false; (3) one containing 16 assignments where v_1 and v_6 are set to false; and (4) one containing 16 assignments where v_1 is set to false and v_6 is set to true.

Among these equivalent classes, classes 1 and 4 are all satisfiable assignments. Unaware of these two classifications, DPLL(T) may need to send all 16+2=18 assignments to the theory solver. However, each class only needs to send *one* assignment (*i.e.*, v_1, v_2, v_3, v_4, v_5 are true representing class 1 and $(v_1 = false, v_6 = true)$ representing class 4) to the theory solver. It is because the propositions of atoms corresponding to unassigned variables in these two assignments need not be considered by the theory solver. For example, because the propositions corresponding to $(v_1 = false, v_6 = true)$ are always theory satisfiable in upholding the assertion, none of the propositions corresponding to any assignment in class 4 can make Ψ unsatisfiable. Similarly, because the conjunction of propositions corresponding to v_1 through v_5 being true is theory satisfiable in upholding the assertion, the original formula Ψ is satisfiable regardless of the truthfulness of the proposition corresponding to v_6.

Assignments in classes 2 and 3 are unsatisfiable and hence do not need to be sent to the theory solver. However, if DPLL(T) explores v_2 through v_6 before v_1 and assigns false first, it may need to check all 30+16=46 assignments in these two classes. Instead, if DPLL(T) explores the assignments of v_1 first and assigns false to variables first, its SAT solver only needs to evaluate the Boolean satisfiability of 6 partial assignments to find out none in classes 2 and 3 is satisfiable.

Second, current SMT-based CPV tools ignore the context of network configurations and the network invariant and use a fixed encoding to construct the configuration formula N, *resulting in a verification formula with redundant variables* and hence impairing the efficiency. Take the same example in Fig. 5, the network verification formula Ψ has 6 variables. However, the invariant to verify only considers the value of $in_1.med$ and is not interested in the value of $in_1.valid$, $in_1.lp$, $in_1.comm1$ or $in_1.comm2$. Therefore, we can verify the correctness of configurations in upholding this invariant with a formula Ψ' with only 2 variables.

B. Key Insights

To address the aforementioned fundamental issue, we now elaborate on the key insights of NetSMT to leverage network domain knowledge to accelerate SMT-based CPV.

Key insight 1: network domain knowledge can guide the search for a solution to the network verification formula by avoiding redundant search space. Given a network verification formula $N \wedge \neg P$ and its CNF form Ψ, our first insight consists in (1) leveraging the control flow of the network configurations and network topology to arrange the exploration order of variables in $\mathcal{B}(\Psi)$, its Boolean abstraction, and (2) leveraging operators' intent to first explore the assignments that are more likely to lead to a satisfiable solution to Ψ (*i.e.*, a network error).

Specifically, NetSMT first computes the exploration order by designing and enforcing three guidelines of partial order of variable exploration in $\mathcal{B}(\Psi)$: exploring branching variables first (*e.g.*, variables corresponding to branching statement in $N \wedge \neg P$), exploring variables whose residing routers are closer to the destination router first, and exploring variables in the same residing router in an order of received route announcements over selected routes and over sent route announcements. Second, during the exploration, NetSMT assigns values to variables in $\mathcal{B}(\Psi)$ by leveraging facts such that operators have a clearer picture of their intent on a single router than on the whole network and a network error must contain a data plane path that is different from the ones specified in the invariant P. Such assignments are closer to the actual forwarding behavior of the configurations. As such, if the configurations are erroneous, these assignments are more likely to be satisfiable. These two designs substantially prune the redundant search space of Ψ and improve the efficiency of solving network verification formulas.

Key insight 2: network domain knowledge can help simplify $N \wedge \neg P$, reducing the problem scale. As illustrated from our example in Fig. 5, the network configuration formula N may have a lot of redundancy, leading to unnecessary large scale of the verification formula $N \wedge \neg P$. As such, our second insight consists of (1) scanning the configuration files to remove configurations that are unrelated to the invariant to

verify, and (2) lifting the level of abstraction of long-width, concrete variables (*e.g.*, BGP local preference and MED) to short-width, abstract variables that cover the number of distinct values of the corresponding concrete variables in the configurations. We prove the correctness of these two designs by finding an equivalent simplification of $N \wedge \neg P$ with the same verification result. Although finding a minimal equivalent simplification of $N \wedge \neg P$ with no redundancy is NP-complete, our evaluation shows that these two designs can lead to a considerable reduction on the scale of the network verification formula, and hence accelerating SMT-based CPV.

III. Guiding SMT solving

This section details how we use network domain knowledge to guide the variable and assignment exploration order of SMT solving, enhancing efficiency by avoiding redundant search.

A. Arranging the Variable Exploration Order

As previously mentioned, Ψ denotes the SMT formula after Tseitin's transformation, $\mathcal{B}(\Psi)$ denotes the Boolean abstraction of Ψ. Additionaly, let V denote the set of all Boolean variables in $\mathcal{B}(\Psi)$. V^b is the set of all branching variables where $V^b \subseteq V$, which are variables corresponding to branching statement in the network verification formula (*e.g.*, $in_{DA}.valid$ and $failed_{A,D}$ shown in Fig. 3). Let v_1 and v_2 be two variables in V, we use $\preceq$ to express their exploration order. We say v_1 is prior to v_2 if $v_1 \preceq v_2$.

We propose three guidelines to arrange the exploration order of variables: all branching variables are prior to other variables, exploring branching variables whose residing routers are closer to the destination first, and prioritizing branching variables whose residing routers are the same by the corresponding route announcement type. Next, we are going to detail each guideline and the design principle of them.

Guideline 1: All branching variables are prior to other variables. This guideline is formally expressed as follows:

$$\forall v_1 \in V^b, v_2 \in V \backslash V^b \Longrightarrow v_1 \preceq v_2. \qquad (2)$$

This guideline is inspired by control flow-guided SMT solving for program verification [28], [29]. We observe that, like software programs, network configurations contain many branching statements (*e.g.*, the matching conditions in route policies), which control the assignment of the variables within its condition. However, generic SMT solvers are not aware of this domain knowledge and treat the variables independently, which may need 2^n explorations for n variables in the worst case. In fact, as Section II shows, many explorations can be avoided by first exploring branching variables.

Considering A's import policy shown in Fig. 3, one of $in_{DA}.comm910$ and $in_{DA}.comm920$ must be set to true if $in_{DA}.valid$ is set to true. However, given that the solver treats these variables independently, it remains possible for $in_{DA}.comm910$ and $in_{DA}.comm920$ to be both set to false even when $in_{DA}.valid$ is true. If we first explore branching variables, this redundant exploration can be reduced because their assignment is restricted by the branching variables.

Guideline 2: Branching variables whose residing router is closer to the destination are explored first. Because there are a lot of branching variables, simply preferring branching variables to other variables is not enough. We further prune search space by ordering the branching variables with the following guideline::

$$\forall v_1, v_2 \in V^b, dist(v_1) < dist(v_2) \Longrightarrow v_1 \preceq v_2, \qquad (3)$$

$Dist(v)$ represents the distance from the router, where variable v resides, to the destination. This guideline is motivated by the execution process of the path-vector routing protocol. Like software programs, the routing protocol has an execution order. For example, router A can only send routing messages to router S after it has received the routing message sent from router D. Therefore, the variable exploration order should be consistent with the routing behavior. We force it by prioritizing branching variables based on topological position.

Consider the example network shown in Fig. 2, 10.1.0.0/24 is the destination and router D is the route origination. So, the branching variable belonging to router D, such as $out_{DA}.valid$, has the highest priority due to the closest distance. Since the best symbolic route of router D is determined by its possession of the connected route, the conflict can be quickly detected by the constraint of D's export policy even if we wrongly assigned $out_{DA}.valid$ to be false.

Instead, without this order, variables such as $out_{AS}.valid$ can be assigned before $out_{DA}.valid$, which can be interpreted as router A sends a routing message to S before D send out any route message, which can not happen in reality. In this case, if $out_{AS}.valid$ is wrongly assigned to be false, the solver may take a long time to detect this conflict. For example, the solver may also wrongly assign $best_A.valid$ as false, meaning that A has no route for the destination, which is wrong but does not conflict with $out_{AS}.valid$. These conflicts will not be detected until $out_{DA}.valid$ is correctly assigned, which infers $in_{DA}.valid$ to true and cause conflict between $in_{DA}.valid$ and $best_A.valid$. Our experiment indicates that this scenario happens frequently. As such, without this ordering, many redundant explorations may happen.

Guideline 3: For branching variables residing in the same router, we order them based on their corresponding route announcement type. Because there are many variables residing in the same router, we use this guideline to further order them, which is expressed as:

$$\forall v_1, v_2 \in V^b, dist(v_1) = dist(v_2), type(v_1) < type(v_2) \Longrightarrow v_1 \preceq v_2, \qquad (4)$$

where $type(v)$ represents the corresponding route announcement type of v. In our example, there are three types of route announcement: $import$, $best$, and $export$. We define the relation $import < best < export$ according to the routing behavior. Considering the route exchange process, the best route is determined by all import routes it received, and the export route is determined by the best route. The idea behind it is the same as Guideline 2, aiming to force the explore order to comply with the routing behavior.

B. Arranging the Assignment Exploration Order

Although lots of explorations can be avoided by guiding the solver to explore the variables in a particular order, con-

flicts still happen regularly with the default value assignment method. As a result, we propose two guidelines to guide the assignment order of branching variables. The basic idea is guiding the solver to assign the variable that could lead to a solution to the network verification formula, a data plane violating invariant.

Guideline 4: For branching variables, we prefer the value assignment that is consistent with operating intent. Considering that most routers usually operate correctly despite some failures, we can prioritize exploring assignments that align with the operator's intent, effectively minimizing conflicts.

Considering the example network, for prefix 10.1.0.0/24, the operator only intends to deny route messages at router S. Therefore, when deciding values $out_{DA}.valid$, $best_A.valid$ and $out_{AS}.valid$, we can use this knowledge to guide assigning their value to true, consequently avoiding conflicts. Furthermore, let us consider the data center network, which barely uses route policy to deny route messages due to its intent to guarantee connectivity. In this case, using Z3 as the SMT solver may encounter a lot of conflicts because it always first explores the false assignment. Thus, Z3 will falsely judge that every router denies all route messages and does not have a route to the destination. In contrast, conflicts can be significantly reduced if we explore the true assignment first.

Guideline 5: For branching variables that reside in the suspicious error routers, we prefer the assignment that is contrary to operating intent. Since the resulting data plane we are searching for must violate the operating intent, we cannot simply guide all variables first to explore the assignments consistent with intent. Therefore, we need to adjust the assignments of variables in routers suspected of errors to be contrary to the operator's intent. Considering our example network, the invariant is router S can not reach 10.1.0.0/24. If the operator suspects that the import policy of router S for router B is incorrectly configured, we can guide the assignment of the variable in_{BS} to true, contrary to the intended false assignment. Here, in_{BS} denotes the symbolic BGP route announcement sent from B to S.

C. Guiding SMT Solving with Variable and Assignment Order

With the variable and assignment exploration order above, we need to force the SMT solver to obey the exploration order. We implement it by modifying the unassigned variable selection and assignment part of the DPLL(T) algorithm. Concretely, we use two structures to store the exploration order: a queue q to store all branching variables ordered by the guideline of variable exploration and a map m that stores the preferred assigned values for each branching variable generated based on the guideline of assignment exploration. In the modified algorithm, when the solver tries to select an unassigned value, we get the first unassigned variable from the branching variables queue q and it will be assigned according to the map m. If all branching variables are assigned, the default exploration approach is used for unassigned variables.

Correctness of guided SMT solving. Like existing branching guidelines [28]–[31], our algorithm only affects the order in which variables are considered, but not the logical reasoning used to derive the solution. The core DPLL(T) algorithm still needs to explore the entire search space, either finding a satisfiable assignment or proving unsatisfiability. Therefore, changing the decision method only affect the search order and efficiency but not the algorithm's correctness.

D. Discussion

Generality of the proposed guidelines. Variable exploration order guidelines, grounded in path-vector protocol semantics, can easily extend to other protocols like OSPF and ISIS, as distance-vector and link-state protocols can be modeled similarly [6], [7]. Assignment exploration order guidelines depend on network-specific operating intents. Nevertheless, they remain general because an overall intent often suffices to reduce conflicts significantly. For the data center network example shown in guideline 4, simply prioritizing true assignment is adequate. Besides, the intent can be automatically generated by analyzing configurations or based on configuration rules such as the Gao-Rexford rule [32]. However, a trade-off exists between efficiency and generality: more precise intent can prevent more conflicts and accelerate solving, but at the cost of being more network-specific.

Exploration of potential additional guidelines. Our guidelines provide a robust foundation but only encompass a subset of possible strategies. First, we currently overlook certain variables like link variables, which could be helpful in scenarios like k-link failure verification. Second, leveraging extra domain knowledge, such as the symmetry structure, could enhance the solving process. Furthermore, while our exploration order is set in advance, integrating it with the solving process offers a potential improvement through adaptive changes.

IV. Simplifying SMT formula

This section details how we use network domain knowledge to enhance the SMT formula encoding. Concretely, we simplify the formula in two ways: (1) pruning the configurations that are unrelated to the invariant; (2) abstracting the long-width variables into short-width variables.

A. Pruning Unrelated Configurations

Current SMT-based verification techniques simply encode whole configurations of all routers into a big SMT formula, leading to poor scalability for large and complex networks. However, many configuration statements are not required to be encoded because of their ineffectiveness in verification.

The basic idea of configuration pruning is to reduce unnecessary constraints and variables based on the invariant. Specifically, we implement this approach in three steps. First, the statements not configured for any address in the property's prefix are reduced from the constraint, and we denote these statements with S^R. Second, the communities only introduced in S^R are directly reduced from each symbolic route, denoted as C^R. Last, the statements configured for the community c that satisfy $c \in C^R$ are reduced from the constraint.

Considering our example, since the property only concerns the forwarding behavior of packets destined for 10.1.0.0/24,

the configuration statements configured for prefixes that not intersected with the destination do not need to be encoded into the SMT formula, such as the statements for 10.0.0.0/24. This pruning can not only simplify the complexity of constraint but also efficiently reduce the number of variables. Because community attributes are costly encoded by introducing a Boolean variable $r.c$ for every symbolic route r and community c that appears in some router's configuration. For example, ten variables can be reduced even if only considering the symbolic routes of router A (community 910 and 930 are not encoded in two symbolic import routes, two symbolic export routes, and one symbolic best route).

B. Abstracting Long-width Variables

Inspired by the use of abstract interpretation to simplify network simulation [11], we design a variable abstraction method to simplify the SMT formula without sacrificing the correctness of verification.

Our basic idea of variable abstraction is to narrow the assignment range by replacing the concrete value with the abstract value. Concretely, we focus on the attribute used to select the best path and express with the integer value, such as weight, local preference, and med. For non-transitive attributes such as weight, we do the abstraction for each router independently, meaning that the abstraction value range only depends on the number of different values in one router's configuration. For transitive attributes, we do the abstraction for routers in the same domain that the attribute can transmit. For example, the local preference attribute can only be transmitted between iBGP neighbors. Therefore, we can abstract the local preference value for each AS separately.

Considering our example network, the local preference can only be assigned to three different values, which are 50, 200, and 100 (default value). Therefore, we can use the abstract values 1, 2, and 3 to represent them, which can be cheaply expressed with a 2-bit unsigned bit vector instead of the original expensive 16-bit integers. Moreover, the 2-bit unsigned bit vector can be further abstracted in two ways. First, since all routers in our network connect with the eBGP connection, the local preference attribute will not be transmitted between them, meaning that every router only needs to keep the local preference value in its configuration. For example, routers A, B, and D only need a 1-bit unsigned bit vector to represent the default value. Second, the abstraction can be combined with the reduction method. Since router A's statements for communities 910 and 930 can be reduced based on the invarient, the local preference values can be reduced consequently, leading that router S can also use a 1-bit unsigned bit vector to represent the local preference values.

C. Correctness of Formula Simification

The general idea to prove the correctness of our formula simplification approach is to show that the data plane result is consistent before and after the abstraction. To prove it, we use the stable path problem (SPP) [33] to express the network model and routing semantics of path vector protocols. For a destination, SPP uses $\mathcal{P}$ to represent all permitted paths and their priority for each node, which determines the possible convergences of the protocol. Therefore, the data plane result is consistent if $\mathcal{P}$ remains the same after abstraction.

$\mathcal{P}$ depends on the route transfer and selection functions, while the transfer function determines the elements in $\mathcal{P}$, and the selection function determines their order. Considering our configurations pruning approach, because the pruned statements and community tags are ineffective, the semantics of transfer and selection functions remains unchanged. Second, abstracting attributes also retain the semantics of these two functions since the abstract domain captures all possible values and preserves the priority order. Therefore, the elements and order of $\mathcal{P}$ will not change after the abstraction.

Network	#Nodes	#Lines	Network	#Nodes	#Lines	Network	#Nodes	#Lines
REN	34	4.76×10^3	CUS	86	1.15×10^4	CL	323	4.41×10^4
ARN	35	4.96×10^3	CLT	154	2.06×10^4	LDTC	537	7.43×10^4
BIC	49	6.90×10^3	USC	174	2.20×10^4	TCCL	621	8.49×10^4
ESN	69	9.20×10^3	COG	198	2.71×10^4	KDL	755	1.02×10^5
LAT	70	9.11×10^3	CD	267	3.71×10^4			

Fig. 6. WAN datasets statistics.

V. Performance Evaluation

We implement a prototype of NetSMT and evaluate its performance extensively to study the following two questions: (1) How does NetSMT perform compared to the state-of-the-art SMT-based CPV tools? (2) What is the effect of the guided SMT solving and simplified SMT formula, respectively?

A. Implementation

We implement NetSMT in C++ and Java, including guided SMT solving based on Z3 4.12.2 [18], (named the *original* z3) and simplified SMT formula based on Minesweeper [7].

B. Experiment Setting

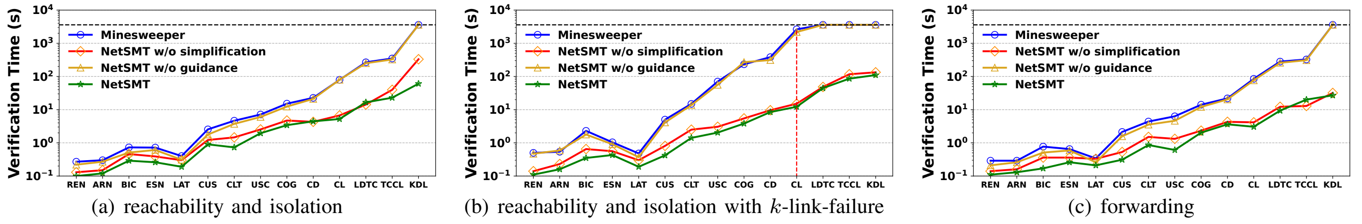
1) Dataset: We use the synthesized network configurations to evaluate NetSMT, including wide-area networks (WAN) and data center networks (DCN). For WAN, we use the topology with router size from 34 to 755, and synthesize base configurations [34]–[36] for each topology. These topologies are obtained either through direct selection or synthesis from topologies in topology zoo [37]. Fig. 6 summarizes the statistics for the synthesized WAN configurations.

For DCN, we use the fat-tree architecture [38] and follow the RFC 7938 [39] to generate base BGP configurations for connection. The fat-tree structure ranges from 4 ports to 20 ports in each switch. Furthermore, we introduce a modest amount of errors to the base configuration to emulate invariant violations for both WAN and DCN.

2) Invariants: We evaluate following five invariants.

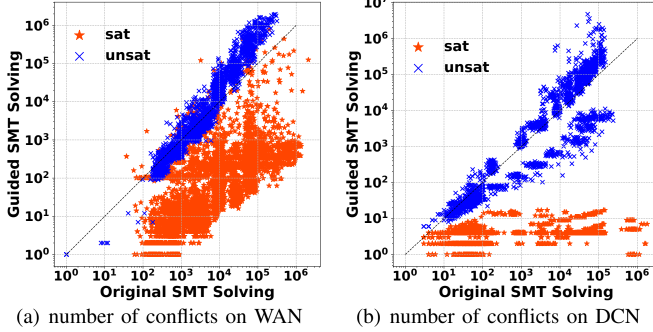
- **Pair-wise reachability/isolation**: Specific pairs of nodes are able/unable to reach each other.
- **Pair-wise reachability/isolation with k-link-failure**: Specific pairs of nodes are able/unable to reach each other, even if any k links fail. We set k=3.
- **Forwarding**: Generate a stable data plane.

3) Baseline: For WAN, we use Minesweeper with the original z3 as the baseline. The reason why we do not choose Kirigami [20] and Lightyear [21] is that Kirigami [20] cannot verify the k-link-failure, and Lightyear [21] has false positives in verification results due to model over-approximation.



(a) reachability and isolation
(b) reachability and isolation with k-link-failure
(c) forwarding

Fig. 7. The SMT verification time on the satisfiable benchmarks on WAN.



(a) number of conflicts on WAN
(b) number of conflicts on DCN

Fig. 8. Number of conflicts on all benchmarks.

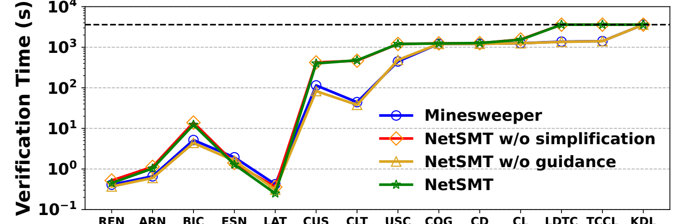


Fig. 9. The SMT verification time on the unsatisfiable benchmarks on WAN.

For DCN, we also use BiNode [17] as the baseline in addition to Minesweeper. BiNode uses a policy-aware model to speed up the verification time for networks with routing policies following the Gao-Rexford [32] condition. The reason why we do not choose BiNode [17] for WAN is that our WAN configurations do not meet the condition, while the condition applies to DCN with the fat-tree structure.

4) Setup: We evaluate the performance of NetSMT by verifying different invariants on topologies of various sizes. For each topology, we randomly select 100 node pairs or destination IPs for invariant verification. These queries contain both satisfiable and unsatisfiable ones. We evaluate these verification systems on a Linux server with an Intel Xeon Silver 4210R 2.40GHz CPU and 128GB memory.

We use the 90th quantile of verification time as the main metric for evaluation. This choice is due to the potential occurrence of outliers in our experimental results. We also measure the number of conflicts that indicate the tried times in the search procedure of SMT solving. We set a timeout period of 1 hour for each verification process.

C. Performance on WAN

1) Verification Time: We consider Minesweeper using the original z3, NetSMT without simplification, NetSMT without guidance, and NetSMT.

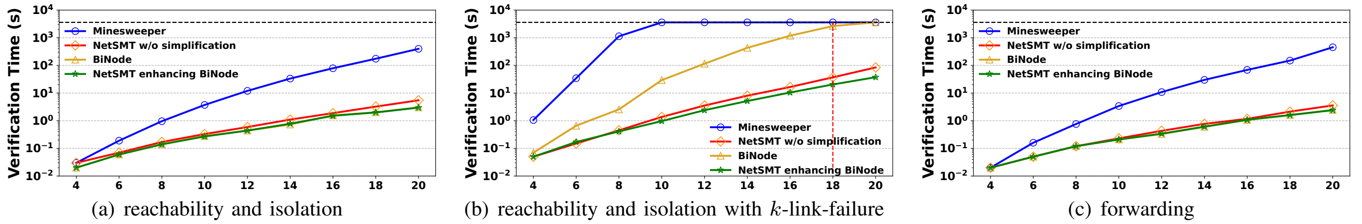
Due to the similarity between reachability and isolation, we calculate the average of their statistics (i.e. the average of the 90th quantile of verification time on each invariant) and show them on a single figure. Fig. 7 shows the verification time on the satisfiable benchmarks of different invariants. NetSMT effectively reduces verification time for both the guided SMT solving and simplified SMT formula techniques, with a more substantial decrease in the guided SMT solving. NetSMT results in a speedup of up to 215.8× compared to Minesweeper (shown in Fig. 10(b) with a red line). Guided SMT solving can be up to 178.7× faster than the original z3 on SMT formulas whether the formula simplification is applied or not. Meanwhile, simplifying the formulas can accelerate the verification process by up to 5.45×. Despite NetSMT performing best in most cases, the non-determinism of the Z3 Java API will result in fluctuations across runs, leading to some occasional outliers.

Fig. 9 shows the statistics of the verification time on unsatisfiable WAN queries. Results show NetSMT does not achieve acceleration and, in some cases, even reduces efficiency. This result is mainly because unsatisfiable formulas require full space exploration. However, such inertia does not reduce the utility of NetSMT, because network configurations are often erroneous, and quickly identifying these errors is vital for effective network management. Accelerating verification on unsatisfiable queries remains an open question. A promising approach is applying network domain knowledge to learn additional counterexamples, thus narrowing the search space.

2) Effect of Guided SMT Solving: We present the number of conflicts of all benchmarks in Fig. 8(a). The horizontal axis represents the number of conflicts generated using the original z3 solver, while the vertical axis represents the quantities using the guided SMT solver. Each point on the graph corresponds to an instance. The proximity of the mark to the lower-right corner indicates the effectiveness of the guided SMT solver. Closer proximities indicate better performance, and vice versa.

As we can see, on all satisfiable benchmarks, the number of conflicts guided by our design is much fewer than that with the original z3 and is essentially flat on the unsatisfiable benchmarks. These results indicate that NetSMT uses domain knowledge to guide the search process for finding a counterexample with fewer try times in the search procedure.

3) Effect of Simplified Formula: To evaluate the effect of formula simplification, We measure the average number of SMT variables and formulas before and after model simplification on all benchmarks. Fig. 11 shows that model simplification can significantly reduce the number of variables and formulas in topologies of various sizes. It suggests that there remains considerable redundancy in Minesweeper's SMT formulas, making our simplification of the SMT formula a key factor contributing to verification acceleration.



(a) reachability and isolation (b) reachability and isolation with k-link-failure (c) forwarding

Fig. 10. The SMT verification time on the satisfiable benchmarks on DCN.

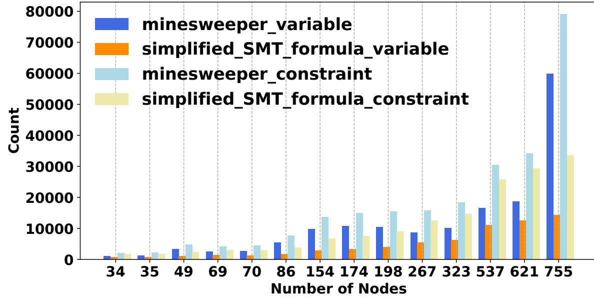


Fig. 11. Number of SMT variables and formulas.

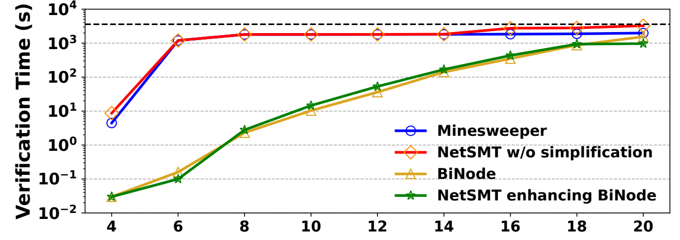


Fig. 12. The SMT verification time on the unsatisfiable benchmarks on DCN.

D. Performance on DCN

According to RFC 7938 [39], the configurations employed on fat-tree do not contain redundant policies, making simplified SMT formulas unsuitable for DCNs, hence unevaluated.

1) Verification Time: We consider Minesweeper using the original z3, NetSMT without simplification, and BiNode using the original z3. We also consider BiNode using the modified z3 in NetSMT (*i.e.*, the guided z3). Fig. 10 shows the SMT verification time of the satisfiable benchmarks on DCN topologies of various sizes. The results obtained on fat-tree networks are similar to those observed on WAN compared to Minesweeper. Additionally, when applying BiNode to address this specific network configuration, our guided SMT solving also contributes to its accelerated processing. Guided SMT solving demonstrates a substantial improvement with the k-link-failure invariant, achieving up to 129.5× improvement. Moreover, BiNode encounters a timeout when the topology size reaches the fat-tree with 20 ports, while the verification time of BiNode with guidance remains within 30 seconds. Other invariants are already efficiently verified using BiNode, leaving little room for further improvement on verification time. Fig. 12 shows the verification time of unsatisfiable benchmarks on DCN, which resembles the results on WAN.

2) Effect of Guided SMT Solving: The significant reduction in the number of conflicts for satisfiable benchmarks can be seen in Fig. 8(b), demonstrating that guided SMT solving is also effective for DCN. The number of conflicts on DCN is fewer compared to the WAN results in Fig. 8(a). On all satisfiable benchmarks, the number of conflicts guided by our design remains limited to no more than 10, contributing to a notably significant acceleration on DCN.

VI. Related Work

Network control plane verification. CPV [5]–[16] uses formal methods to analyze the correctness of configuration files to ensure they will operate as intended. SMT-based tools [5], [7], [17] have advantages over simulation-based [6], [8], [9], [14], [15] and graph-based [12], [13] tools for their capability to verify diverse routing protocols in all possible converged states. But they are inefficient in verifying large networks.

Accelerating SMT-based control plane verification. Existing techniques [10], [17], [20], [21] only work for specific scenarios and are not effective when dealing with complicated invariants, such as verifying reachability under k-link-failure. Kirigami [20] heavily relies on the operator's experience and cannot handle k-link-failure invariant and network with multiple convergences; Lightyear [21] may produce false positives results due to its over-approximation model; BiNode [17] limits in verifying configurations conforming to the G-R condition. Instead, NetSMT systematically leverages domain knowledge to accelerate SMT-based verification in generic large-scale networks.

Control flow-guided SMT solving. There are previous efforts on utilizing control flow knowledge to accelerate SMT-based program verification [28], [29], [31]. Our guideline 1 to prioritize branching variables is inspired by them. However, we go beyond substantially to design multiple guidelines to accelerate SMT-based CPV using network domain knowledge.

VII. Conclusion

We design NetSMT, a fast SMT-based CPV tool that systematically leverages network domain knowledge to guide the search for a solution to the network verification formula by avoiding redundant search space and simplifying the verification formula to reduce the problem scale. Extensive evaluation demonstrates the efficacy and efficiency of NetSMT.

The authors have provided public access to their code and data at [22].

Acknowledgments

We are extremely grateful to the anonymous INFOCOM 2024 reviewers for their wonderful and constructive feedback. This work is supported in part by the National Key R&D Program of China 2022YFB2901502, NSFC Award #62172345, MOE of China Award #2021FNA02008, Open Research Projects of Zhejiang Lab #2022QA0AB05, and NSF-Fujian-China 2021J05003 and 2022J01004, as well as by the National Natural Science Foundation of China under Grant 62302409, the Natural Science Foundation of Fujian Province of China under Grant 2021J05003.

REFERENCES

[1] "Amazon AWS Outage," https://www.whizlabs.com/blog/amazon-aws-outage/, 2020.

[2] "Prolonged AWS Outage Takes Down a Big Chunk of the Internet," https://www.theverge.com/2020/11/25/21719396/amazon-web-services-aws-outage-down-internet, 2020.

[3] "Google outage: YouTube, Docs and Gmail knocked offline," https://www.bbc.com/news/technology-55299779, 2020.

[4] "Google Cloud Hit by Outage in New Melbourne, Australia, Region," https://www.datacenterdynamics.com/en/news/google-cloud-hit-by-outage-in-new-melbourne-australia-region/, 2021.

[5] K. Weitz, D. Woos, E. Torlak *et al.*, "Scalable verification of border gateway protocol configurations with an smt solver," in *OOPSLA 2016*. ACM, 2016, pp. 765–780.

[6] A. Fogel, S. Fung, L. Pedrosa *et al.*, "A general approach to network configuration analysis," in *NSDI'15*. USENIX, 2015, pp. 469–483.

[7] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "A general approach to network configuration verification," in *SIGCOMM '17*. ACM, 2017, pp. 155–168.

[8] F. Ye, D. Yu, E. Zhai *et al.*, "Accuracy, scalability, coverage–a practical configuration verifier on a global wan," in *SIGCOMM'20*. ACM, 2020, pp. 599–614.

[9] S. K. Fayaz, T. Sharma, A. Fogel *et al.*, "Efficient network reachability analysis using a succinct control plane representation," in *OSDI'16*. USENIX, 2016, pp. 217–232.

[10] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Control plane compression," in *SIGCOMM'18*. ACM, 2018, pp. 476–489.

[11] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, "Abstract interpretation of distributed network control planes," *POPL 2019*, vol. 4, pp. 1–27, 2019.

[12] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan, "Fast control plane analysis using an abstract representation," in *SIGCOMM'16*. ACM, 2016, pp. 300–313.

[13] A. Abhashkumar, A. Gember-Jacobson, and A. Akella, "Tiramisu: Fast multilayer network verification," in *NSDI'20*. USENIX, 2020, pp. 201–219.

[14] S. Prabhu, K. Y. Chou, A. Kheradmand *et al.*, "Plankton: Scalable network configuration verification through model checking," in *NSDI'20*. USENIX, 2020, pp. 953–967.

[15] S. Steffen, T. Gehr, P. Tsankov *et al.*, "Probabilistic verification of network configurations," in *SIGCOMM '20*. ACM, 2020, pp. 750–764.

[16] N. Giannarakis, D. Loehr, R. Beckett, and D. Walker, "Nv: An intermediate language for verification of network control planes," in *PLDI 2020*. ACM, 2020, pp. 958–973.

[17] X. Shao and L. Gao, "Verifying policy-based routing at internet scale," in *INFOCOM 2020*. IEEE, 2020, pp. 2293–2302.

[18] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS 2008*. Springer, 2008, pp. 337–340.

[19] H. Barbosa, C. Barrett, M. Brain *et al.*, "cvc5: A versatile and industrial-strength smt solver," in *TACAS 2022*. Springer, 2022, pp. 415–442.

[20] T. A. Thijm, R. Beckett, A. Gupta, and D. Walker, "Kirigami, the verifiable art of network cutting," in *ICNP 2022*. IEEE, 2022, pp. 1–12.

[21] A. Tang, R. Beckett, S. Benaloh *et al.*, "Lightyear: Using modularity to scale bgp control plane verification," in *SIGCOMM'23*, 2023, p. 94–107.

[22] NetSMT, https://github.com/sngroup-xmu/NetSMT.

[23] H. Ganzinger, G. Hagen, R. Nieuwenhuis *et al.*, "Dpll (t): Fast decision procedures," in *CAV 2004*. Springer, 2004, pp. 175–188.

[24] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

[25] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[26] J. M. Silva and K. A. Sakallah, "Grasp-a new search algorithm for satisfiability," in *ICCAD 1996*. IEEE, 1996, pp. 220–227.

[27] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus*. Springer, 1983, pp. 466–483.

[28] J. Chen and F. He, "Control flow-guided smt solving for program verification," in *ASE'18*. ACM, 2018, pp. 351–361.

[29] J. Chen and F. He, "Leveraging control flow knowledge in smt solving of program verification," *TOSEM 2021*, vol. 30, no. 4, pp. 1–26, 2021.

[30] M. W. Moskewicz, C. F. Madigan, Y. Zhao *et al.*, "Chaff: Engineering an efficient sat solver," in *DAC'01*. ACM, 2001, pp. 530–535.

[31] H. Fan, W. Liu, and F. He, "Interference relation-guided smt solving for multi-threaded program verification," in *PPoPP'22*, 2022, pp. 163–176.

[32] L. Gao and J. Rexford, "Stable internet routing without global coordination," *TON 2001*, vol. 9, no. 6, pp. 681–692, 2001.

[33] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," *TON 2002*, vol. 10, no. 2, pp. 232–243, 2002.

[34] T. Benson, A. Akella, and A. Shaikh, "Demystifying configuration challenges and trade-offs in network-based isp services," in *SIGCOMM'11*. ACM, 2011, pp. 302–313.

[35] H. Kim, T. Benson, A. Akella, and N. Feamster, "The evolution of network configuration: A tale of two campuses," in *IMC'11*. ACM, 2011, pp. 499–514.

[36] T. Benson, A. Akella, and D. A. Maltz, "Mining policies from enterprise network configuration," in *IMC'09*. ACM, 2009, pp. 136–142.

[37] S. Knight, H. X. Nguyen, and Falkner, "The internet topology zoo," *JSAC 2011*, vol. 29, no. 9, 2011.

[38] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *SIGCOMM'08*, pp. 63–74, 2008.

[39] P. Lapukhov and A. Premji, "Rfc 7938: Use of bgp for routing in large-scale data centers," RFC Editor, 2016.