# Metha: Network Verifiers Need To Be Correct Too!

# *Metha*: Network Verifiers Need To Be Correct Too!

Rüdiger Birkner* Tobias Brodmann* Petar Tsankov Laurent Vanbever Martin Vechev

*These authors contributed equally to this work.

*ETH Zürich*

## Abstract

Network analysis and verification tools are often a godsend for network operators as they free them from the fear of introducing outages or security breaches. As with any complex software though, these tools can (and often do) have bugs. For the operators, these bugs are not necessarily problematic except if they affect the precision of the model as it applies to their specific network. In that case, the tool output might be wrong: it might fail to detect actual configuration errors and/or report non-existing ones.

In this paper, we present Metha, a framework that systematically tests network analysis and verification tools for bugs in their network models. Metha automatically generates syntactically- and semantically-valid configurations; compares the tool's output to that of the actual router software; and detects any discrepancy as a bug in the tool's model. The challenge in testing network analyzers this way is that a bug may occur very rarely and only when a specific set of configuration statements is present. We address this challenge by leveraging grammar-based fuzzing together with combinatorial testing to ensure thorough coverage of the search space and by identifying the minimal set of statements triggering the bug through delta debugging.

We implemented Metha and used it to test three well-known tools. In all of them, we found multiple (new) bugs in their models, most of which were confirmed by the developers.

## 1 Introduction

It's Friday night and you are about to push an important (network) configuration update in production. Usually, you would feel terribly nervous doing so as there is always the possibility that you may have missed something. You are only too aware that misconfigurations happen frequently and can lead to major network outages [22,24,27]. Tonight though you feel confident when pressing "deploy" as you have confirmed the correctness of your configuration update using a state-of-the-art configuration verifier. A few minutes later, your phone rings: none of your customers can reach the Internet anymore.

This fictitious situation illustrates an intrinsic problem with validation technologies: their results can only be completely trusted if their analysis is sound and complete. As with any complex software though, these tools can (and often do) have bugs. To be fair, this is not surprising: building an accurate and faithful network analysis tool is extremely difficult. Among others, one not only has to precisely capture all the different protocols' behaviors, but also all of the quirks of their specific implementations. Unfortunately, every vendor, every OS, every device can exhibit slightly different behaviors under certain conditions. For all it takes, these behaviors might be the results of bugs themselves. And yet, failing to accurately capture these behaviors—as we show—can lead to incorrect and possibly misleading analysis results.

A fundamental and practical research question is therefore: *How can developers make sure that their network analysis and verification tools are correct?*

**Metha** We introduce Metha, a system that thoroughly tests network analysis and verification tools to find subtle bugs in their network models using black-box differential testing. Metha automatically finds model discrepancies by generating input configurations and comparing the output of the tool under test with the output produced by the actual router software. For every discovered discrepancy, Metha provides a minimal configuration that helps developers pinpoint the bug. Later on, these configurations can be used to build up an adequate test suite for current and future network tools.

**Challenges** Precisely identifying bugs in network analyzers' models is challenging for at least three reasons. First, the search space of possible configurations is gigantic: there are hundreds of configuration statements, each of which can take many possible parameters. And yet, as our analysis reveals, most of the bugs only manifest themselves when specific configuration statements/values are present. Second, systematically exploring the search space is highly non-trivial (independently of its size) as one not only needs to generate syntactically-valid configurations, but also semantically-valid ones that involve all features and their interactions. Failing to do so could lead to miss bugs, hence lowering coverage.

Finally, upon finding a configuration triggering a discrepancy, figuring out the exact subset of statements requires to solve another tricky combinatorial search.

**Insights** Metha addresses the above challenges by first reducing the search space through restricting the parameters to their boundary values. Metha ensures thorough coverage of the search space by phrasing the search as a combinatorial testing problem and targeting the search towards single and pairwise interactions of configuration statements. To ensure syntactically- and semantically-valid configurations Metha relies on a hierarchical grammar-based approach. Finally, Metha employs delta debugging to identify the minimal bug-inducing set of configuration statements to help the developer better understand and reproduce the bug.

**Bugs found** We demonstrate Metha's effectiveness in practice by finding 62 *real-world bugs* across three popular network analysis and verification tools – Batfish [17], NV [13], and C-BGP [28] – 59 of which have been confirmed by the tools' developers. The majority of the discovered bugs are subtle, silent bugs that undermine the soundness of the tools' results. That is, they could lead operators to incorrect conclusions that their networks behave correctly, while, in fact, they do not.

Our experiments also demonstrate that Metha's key components are essential for its effectiveness. In particular, a random baseline only found 3 bugs, while Metha found 20 bugs with the same number of tests. Last but not least, through our interactions with the tools' developers, we confirm that Metha's minimal configuration examples are indeed useful: some of the developers are already using them to analyze and fix the bugs Metha has discovered.

**Contributions** In summary, our main contributions are:

- A testing system capable of finding bugs in the network models of state-of-the-art network analyzers (§3).
- A formulation of the search problem in terms of combinatorial testing (§5).
- A precise localization procedure relying on delta debugging to isolate bugs and pinpoint the configuration statements causing them (§6).
- An end-to-end implementation of Metha[1] supporting both Cisco IOS and Juniper configurations (§7).
- An evaluation showing that Metha finds real (and unknown) bugs in all the tested tools (§8).

**Limitations** Metha treats the network verification tool it is testing as a black box. Hence, it cannot localize the bug within the tool's actual code. This task is left to the developer. However, by identifying the configuration statements responsible for the bug and by creating a minimal configuration that showcases it, the developer has a good starting point for her work. Similarly, Metha cannot detect whether two observed bugs that are triggered by different configuration statements are caused by the same bug in the underlying network model.

[1] Available at https://github.com/nsg-ethz/Metha

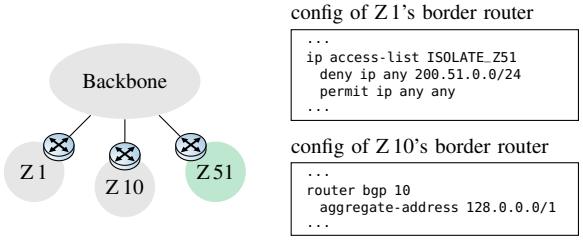



Figure 1: Zone 51 has to be isolated from all the other zones. This is achieved through access-lists at the border routers with the exception of zone 10 where it was forgotten.

## 2 Motivation

We now illustrate how subtle bugs in the network model of network analyzers can lead operators to deploy erroneous configuration changes. We start with two case studies on common configuration features known for easily causing forwarding anomalies: route aggregation and redistribution. In these situations, validating the change with an analyzer is of utmost importance, provided the analysis is correct. We end with a collection of Cisco IOS configuration statements whose semantics were not correctly captured by Batfish [17]. The bugs in this Section were discovered by Metha.

### 2.1 Example 1: Excess Null Route

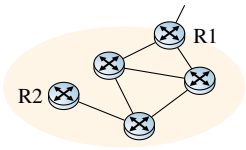
Consider the network in Fig. 1. It consists of a backbone with multiple zones attached to it. The backbone and the zones are interconnected using BGP. Each zone receives a default-route from the backbone. Zone 51 hosts critical infrastructure in the prefix `200.51.0.0/24`, which should not be accessible from any other zone. To enforce this, the routers connecting the zones to the backbone have an access-list (ACL) in place to filter that traffic. However, in zone 10, this ACL was forgotten and, instead, there happens to be a left-over statement from a previous configuration: "`aggregate-address 128.0.0.0 128.0.0.0`". This statement directs the router to advertise the specified aggregate route *if* any more-specific BGP routes in that range exist in the routing table.

**Property violation** Due to the lack of an ACL on the border router, the requirement that mandates to keep zone 51 isolated is violated: traffic from zone 10 can reach zone 51.

**Analyzer mishap** When used on the network above, Batfish, a recent network validation tool, will falsely assert that zone 51 is isolated. The problem is due to the semantics of the left-over `aggregate-address` statement. Batfish wrongly activates the aggregate because of a non-BGP route in the routing table and installs the `null` route. Because of this `null` route, Batfish wrongly assumes that all traffic in zone 10 falling within the aggregate range will be dropped. In practice, the routers only install a `null` route if a BGP route within the aggregate is present, which is not the case here.

| Feature | Description | Possible Consequence |
|---|---|---|
| `max-metric router-lsa` | The model sets maximum metric not only for point-to-point links, but also for stub links. This should only be done when the keyword `include-stub` is used. | A router might appear to be free of traffic and safe to reboot, even though it is not. |
| `default-information originate` | The OSPF routing process should only generate a default-route if the route table has a default-route from another protocol. The model, however, also announces a default-route if there is one in the routing table of different OSPF type, i.e., E1 type. | Additional default-routes might appear in the routing tables. |
| `distance XX` | The model does not consider any changes to the administrative distance. | The forwarding state could be completely wrong. |
| `area X range A.B.C.D/Y` | When summarizing routes between OSPF areas, the model does not insert a null route for the summary to prevent routing loops. | A routing loop could be falsely detected. |
| `set community no-export` | When redistributing a static route into BGP and setting the `no-export` community, the model still advertises the route to its eBGP neighbors. | Reachability properties could be falsely asserted. |
| `neighbor A.B.C.D maximum-prefix X` | Even when a BGP neighbor advertises more prefixes than the specified threshold, the model does not drop the peering to the neighbor. | Reachability properties could be falsely asserted. |

Table 1: A selection of Cisco IOS configuration features that are not correctly modelled by Batfish [17] as discovered by Metha.




config of Z10's border router

```
...
ip route 200.0.0.0/20 Null
...
router ospf 1
  redistribute static
...
```

Figure 2: All routers should be able to reach the Internet. The static route at R2 creates a blackhole and violates that.

## 2.2 Example 2: Incomplete Redistribution

Consider the small company network depicted in Fig. 2. It consists of a single OSPF area. R1 acts as Internet gateway and announces a default-route internally. A static route on R2 drops all the traffic for `200.0.0.0/20` by directing it to the `null` interface. This is intended. What is not intended, however, is the `redistribute static` command at R2.

**Property violation** The following reachability property must always hold: all routers, with the exception of R2, are able to reach the entire Internet. However, this property, is violated since R2 redistributes the static route in the network and, in turn, creates a blackhole for `200.0.0.0/20`.

**Analyzer mishap** When run on this network, Batfish will falsely attest that all routers, with the exception of R2 can reach the entire Internet. The problem is the redistribution command. By default, Cisco routers only redistribute classful networks [7] and only by specifying the `subnets` keyword, they also redistribute any subnets of them. Less-specific networks however are redistributed regardless of the `subnets` keyword (e.g., `200.0.0.0/20` is less-specific than the corresponding class C network `200.0.0.0/24`). Batfish's network model does not incorporate that as it only redistributes classful networks and not less-specific networks.

## 2.3 Selection of Bugs

In addition to the two bugs illustrated in the previous examples, we found several other configuration statements that trigger bugs. We present a selection of them in Table 1 alongside a short description of the observed behavior and possible consequences. All of the presented bugs concern Cisco IOS configuration statements. In our tests, we also used Juniper configurations and found that, in many cases, the same bugs occur. Hence, some of these bugs are not due to vendor-specific behaviors, but due to general inaccuracies of the network model.

# 3 Overview

In this section, we first present the key insights enabling Metha to efficiently uncover bugs in network analyzers. Then, we provide a high-level overview of Metha.

## 3.1 Key Insights

The main challenge in testing network analyzers is that bugs may occur rarely and only for very specific configurations, which we address with a combination of five insights:

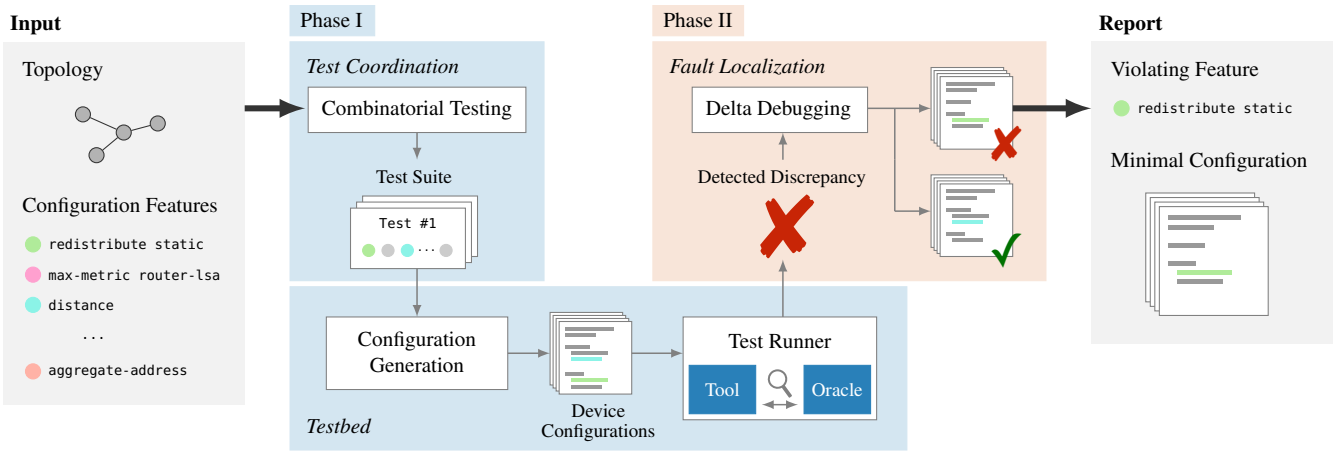



Figure 3: Metha generates a test suite based on the test topology and supplied configuration features. The testbed runs one test after another and compares the computed routing tables of the tool under test to those of an oracle. It then analyzes every discrepancy to localize all the bugs and creates a report for each one of them.

**Producing valid inputs with grammar-based generation** When testing network analyzers, it is of utmost importance to use syntactically- and semantically-valid configurations, meaning the configurations need to be parseable and constraints have to be met such that actual computation takes place in the network. Our key insight is to use a hierarchical grammar-based approach. Approaching it hierarchically allows to resolve the intra- and inter-device constraints. This provides the structure that is then completed using grammar-based configuration generation ensuring syntactical validity.

**Reducing the search space through boundary values** The search space of all possible configurations is prohibitively large. Even a single parameter, such as an OSPF cost, for example, already has $2^{16}$ possible values to test. By focusing the testing on the boundary values (the minimum, maximum, and a normal value), we reduce the search space significantly.

**Exploring the search space with combinatorial testing** Network devices support a wide variety of configuration features that all need to be tested not just by themselves, but also their interactions. Hence, we use combinatorial testing to design a test suite that systematically covers all pairwise interactions of configuration features.

**Comparing the tested tool's output to ground truth** Detecting crash bugs is straightforward as the tool will just fail or report an error. Silent bugs, on the other hand, can only be detected by comparing the output to a ground truth, which is hard to come by. We address this by leveraging a testbed running real router images as an oracle.

**Isolating bugs with delta debugging** Lastly, once one identifies a network configuration that triggers a bug, one needs to identify the configuration statements causing it to provide any useful insights to the tool's developer. Therefore, we use iterative delta debugging to obtain a minimal configuration example reproducing the bug.

## 3.2 Metha

Metha operates in two phases as shown in Fig. 3: First, it aims to find network configurations exhibiting discrepancies between the tool under test and the oracle. To that end, the test coordination determines all the tests to run in the testbed. Second, it identifies the configuration statements responsible for the observed discrepancies through fault localization.

**Input and output** Metha takes two inputs: *(i)* a physical topology, i.e., an undirected graph; and *(ii)* a set of configuration features to be tested, such as, route-maps, and route-summarization. For every discovered bug, Metha creates a report, which consists of the identified discrepancy between the routing tables of the tool and the oracle, the configuration statements causing it and a configuration set to reproduce it.

**Phase I: Test coordination (§4, §5)** The configuration features and the topology provided as input define the search space of Metha's testing efforts which consists of all possible configurations that can be built using these features.

This search space of network configurations is prohibitively large. Therefore, Metha first reduces the values of all parameters to their boundary values, which means it only uses the two extreme values (i.e., the minimum and the maximum) and one "normal" value. Even with this reduction, it is difficult to systematically cover the search space. Hence, Metha creates a test suite relying on combinatorial testing, which allows it to cover all pairs of feature and parameter combinations requiring a minimal number of tests. Each test in the test suite is a set of configuration statements that should be active.

**Phase I: Testbed (§7)** For every single test, Metha generates the device configurations based on the statements as defined by the test suite. Then, it runs these configurations in the tool and the oracle. Once both converged, Metha analyzes the routing tables of the two tools and reports any discrepancies.

| | | |
|---|---|---|
| *BGPProcess* | → | `router bgp` *Integer16* [*Options*] |
| *Options* | → | *Option* \| *Options Option* |
| *Option* | → | *Redistribute* \| *Neighbor* \| *Network* \| ⋯ |
| *Redistribute* | → | `redistribute` *Source* |
| *Source* | → | `direct` \| `static` \| ⋯ |
| *Neighbor* | → | `neighbor` *Address Property* |
| *Property* | → | *RemoteAS* \| *RouteMap* \| ⋯ |
| *RemoteAS* | → | `remote-as` *Integer16* |
| *RouteMap* | → | `route-map` *String Direction* |
| *Direction* | → | `in` \| `out` |

Figure 4: Partial BNF grammar for device configurations.

**Phase II: Fault localization (§6)** A discovered discrepancy can be caused by multiple bugs in the network analyzer. Therefore, Metha applies delta debugging to identify every single bug and the configuration statements causing it. It does so by iteratively testing subsets of the active configuration statements until the entire discrepancy is resolved.

## 4 Search Space

In this section, we define the search space of all possible configurations. We also show how we reduce the search space by restricting the parameter values used in configuration statements to *boundary* values.

### 4.1 Network Configurations

The search space is given by all possible configurations that one can deploy at the network's routers.

**Configurations** A device configuration defines the enabled features along with their parameter values. Formally, the set of all possible configurations is defined by a context-free grammar whose terminals consist of feature names and parameter values. To illustrate this, in Fig. 4 we show a subset of the production rules in Backus-Naur form (BNF). An example configuration derived from this grammar is:

```
1  router bgp 100
2    redistribute static
3    neighbor 1.1.1.2 remote-as 50
4    neighbor 1.1.1.2 route-map map10 out
```

This configuration defines the AS identifier, neighbors, neighbor properties, and route redistribution associated with the BGP routing process `100`. Here, `router bgp`, `redistribute`, `neighbor A.B.C.D remote-as`, and `neighbor A.B.C.D route-map` are configuration statements, while the values to the right define their parameter values. We distinguish three types of parameter values:

**Keywords** are used in configuration statements parameterized by a value drawn from a fixed set of options. For example, the configuration statement `neighbor A.B.C.D route-map` is parameterized by a direction, which is set to either `in` or `out`. For some statements, one can also omit the parameter value altogether, which we model with the designated value ∅. For example, `redistribute connected` is parametrized by a value drawn from the set {∅,`subnets`}, and so both `redistribute connected` and `redistribute connected subnets` are valid statements.

**Integers** are used to define 16- and 32-bit numbers. For example, the configuration statement `router bgp` is parameterized by a 16-bit integer defining the AS number.

**Strings** are used in configuration statements parameterized by custom names. For example, `neighbor A.B.C.D route-map` is parameterized by the route-map's name.

**Semantic constraints** Besides conforming to the syntax in Fig. 4, configurations must also comply with semantic constraints. For example, consider the following configurations:

```
1  interface FastEthernet0/0
2    ip address 1.1.1.1 255.255.255.0
3  !
4  router bgp 100
5    neighbor 1.1.1.2 remote-as 50
6    neighbor 1.1.1.2 route-map map10 out
7  !
8  route-map map10 permit 10
9    match ip address prefixList
```

```
1  interface FastEthernet0/0
2    ip address 1.1.1.2 255.255.255.0
3  !
4  router bgp 50
5    neighbor 1.1.1.1 remote-as 100
```

The top configuration ($C_1$) defines a BGP process with AS number `100` (Line 4), and declares that announcements sent to its BGP neighbor with IP `1.1.1.2` (Line 5) are processed using route-map `map10` (Line 6). The bottom configuration ($C_2$) defines a BGP process with AS number 50 (Line 4), and declares `1.1.1.1` in AS `100` as a neighbor (Line 5). These two configurations illustrate two kinds of semantic constraints:

**Intra-device constraints,** which stipulate conditions that must hold on any (individual) configuration. For example, the route-map `map10` used at Line 6 must be defined within the configuration $C_1$. This constraint holds as `map10` is defined at Line 8.

**Inter-device constraints,** which stipulate conditions across multiple configurations. For example, the AS number assigned to neighbor `1.1.1.2` in $C_1$ at Line 5 must match the AS number declared in $C_2$ at Line 4. This constraint holds as at both lines the AS number is 50.

Finally, we note that we specify the semantic constraints separately from the syntactic production rules as some are not context-free and thus cannot be encoded in the grammar.

**Search space** The search space used by Metha is defined by the set of configurations that one can deploy at the network's routers. As the set of configurations derived from the grammar is, in general, infinite, we restrict all recursive rules so that its language consists of finitely many configurations. For instance, for the grammar given in Fig. 4, we fix the set of BGP options (such as route redistribution) that can appear when defining a BGP routing process. Finally, the search space of Metha is defined as $C^R$, where $C$ is the set of all configurations and $R$ is the set of routers. Note that each element of $C^R$ defines a *network-wide configuration*, assigning a configuration from $C$ to each router in $R$.

## 4.2 Boundary Values

The search space is extremely large due to the enormous number of configurations and the exponentially many combinations in which they can be deployed at the routers. To cope with the large set of configurations, we apply a *boundary values* reduction by restricting the parameters to a small set of representative values. The intuition behind this reduction is that most parameter values lead to the same behavior such that testing them individually provides no additional insights.

The reduction to boundary values ensures that various behaviors of a feature are exercised. For example, the Cisco BGP feature `neighbor X.X.X.X maximum-prefix n` terminates the session when the neighbor announces more than `n` prefixes. When randomly choosing `n`, the feature will most likely not come into action. However, with the boundary values, both the minimum and maximum value are tested, ensuring that the feature is at least once active and once not.

For integer parameters, the values are restricted to: the maximum value, the minimum value, and a non-boundary value. For example, for 16-bit integers, which contain all integers in the range $[0, 65535]$, our boundary value reduction selects three values: 0, 65535, and a value $x$ such that $0 < x < 65535$. Similarly, we reduce the values assigned to string parameters by predefining a fixed set of strings.

# 5 Effective Search Space Exploration

Metha must cover a wide variety of different network configurations to thoroughly test the tool, including many combinations of device features and parameter values. The key challenge is that it is impossible to iterate through every single combination of features and their respective parameter values, even after considering our reduction to boundary values. To address this, Metha relies on *combinatorial testing* [16, 20], which is able to uncover all bugs involving a small number of interacting features. In the following, we first provide relevant background on combinatorial testing, and then we show how Metha uses it to effectively test network tools.

## 5.1 Combinatorial Testing

Combinatorial testing is a black-box test generation technique which is effective at uncovering *interaction bugs*, i.e., bugs that occur because of multiple interacting features and their parameter values. The main assumption behind combinatorial testing is that interaction bugs are revealed by considering a small number of features and parameter values. In this case, one can generate a test suite, called *combinatorial* test suite, that uncovers all such bugs.

To use combinatorial testing, one needs to define a specification of the system's parameters and their values:

**Definition 1** (Combinatorial specification)**.** A combinatorial specification $\mathcal{S}$ is a tuple $(P, V, \Delta)$, where $P$ is a set of parameters, $V$ is a set of values, and $\Delta\colon P \to 2^V$ defines the domain of values $\Delta(x) \subseteq V$ for any parameter $x \in P$.

For example, the combinatorial specification for a program that accepts three boolean flags as input has parameters $P = \{a, b, c\}$, values $V = \{0, 1\}$, and domains $\Delta(a) = \Delta(b) = \Delta(c) = \{0, 1\}$. A *test case* is a total function $tc\colon P \to V$ mapping parameters to values from their respective domains, i.e., with $P(x) \in \Delta(x)$ for any $x$. An example test case for our program is $tc = \{a \mapsto 0, b \mapsto 0, c \mapsto 1\}$. In contrast to test cases, a *t-wise combination* maps only some parameters to values:

**Definition 2** ($t$-wise combination)**.** Given a combinatorial specification $\mathcal{S} = (P, V, \Delta)$, a $t$-wise combination for $\mathcal{S}$ is a function $c\colon Q \to V$ such that $Q \subseteq P$ with $|Q| = t$ and $c(x) \in \Delta(x)$ for any $x \in P$.

An example pairwise combination (i.e., $t = 2$) for our example is $c = \{a \mapsto 0, b \mapsto 1\}$. We write $\mathcal{C}_t^S$ to denote the set of all $t$-wise combinations for a given combinatorial specification $\mathcal{S}$. Note that a test case can cover multiple $t$-wise combinations:

$$comb_t(tc) = \{c \subseteq tc \mid |c| = t\}$$

For instance, our example test case above covers the following three pairwise combinations: $\{a \mapsto 0, b \mapsto 0\}$, $\{a \mapsto 0, c \mapsto 1\}$, and $\{b \mapsto 0, c \mapsto 1\}$.

**Definition 3** ($t$-wise combinatorial coverage)**.** Given a combinatorial specification $\mathcal{S}$, we define the $t$-wise combinatorial coverage of a test suite $T$ as:

$$cov_t(T) = \frac{|\bigcup_{tc \in T} comb_t(tc)|}{|\mathcal{C}_t^S|} .$$

A test suite $T$ is called a *t-combinatorial test suite* if $cov_t(T) = 1$. If the assumption that interaction faults are caused by up to $t$-wise interactions holds, then $T$ finds all bugs. The goal of combinatorial testing is to generate the smallest $t$-wise combinatorial test suite.

## 5.2 Combinatorial Testing of Configurations

In Metha, we apply pairwise combinatorial testing to the generation of network configurations. Concretely, we phrase the search space defined in §4 as a combinatorial specification $\mathcal{S} = (P, V, \Delta)$ as follows. First, each statement that can appear in the configuration, such as route redistribution or route-map as defined in §4, defines a configuration feature. We set $F$ to be the set of all configuration features. The set of parameters $P$ is then given by $R \times F$, where $R$ is the set of routers. Namely, the parameters consist of all configuration features one can define in the device configurations.

Second, the domains of values for each configuration feature contain the boundary values that can be used in the given configuration statement, along with the designated value $\perp$, which indicates whether the configuration feature is enabled or not. That is, $\perp$ results in omitting the configuration statement altogether. We note that for configuration statements with multiple parameters, we take the product as the domain of possible values. For example, the Cisco OSPF configuration feature `default-information-originate` has three optional parameters: `always`, `metric` combined with an integer value, and `metric-type` combined with 1 or 2. After reduction to boundary values this leads to the following three parameters:

$$A = \{\varnothing, \texttt{always}\}$$
$$B = \{\varnothing, \texttt{metric 1}, \texttt{metric 100}, \texttt{metric 1677214}\}$$
$$C = \{\varnothing, \texttt{metric-type 1}, \texttt{metric-type 2}\}$$

The domain of values for this configuration feature is then given by $\{\perp\} \cup (A \times B \times C)$.

Finally, Metha uses the above combinatorial specification to derive a test suite of configurations that covers all pairwise combinations.

# 6 Fault Localization

A discovered discrepancy between the network model and the oracle is only of limited use as the developer still needs to isolate its cause. Often understanding the bug is the most time-consuming part of the debugging process, and fixing it can be done relatively quickly. To help with this, Metha pinpoints the configuration features that cause a discrepancy and finds a minimal configuration, i.e., a configuration with as few configuration features enabled as possible. To do this, Metha uses *iterative delta debugging*, an extended version of classic delta debugging, which lifts the assumption that a single fault causes failures. This extension is important as network configurations are large and complex, and discrepancies are often caused by multiple faults. In the following, we first introduce classic delta debugging and then present its iterative extension.

## 6.1 Delta Debugging

Delta debugging [35] is a well-established fault localization technique, which finds minimal failure-inducing inputs from failing test cases. Below, we present delta debugging in our context, and then define its assumptions and algorithmic steps.

**Terminology** As defined in §5, a test case $tc$ assigns configuration features $F$ to either parameter values or $\perp$, where $\perp$ indicates that a given feature is disabled (i.e., it is omitted from the configuration). Given a test case $tc$ and features $Q \subseteq F$, we write $tc|_Q$ for the test case obtained by disabling all features in $tc$ that are not contained in $Q$:

$$tc|_Q(f) = \begin{cases} tc(f) & \text{if } f \in Q \\ \perp & \text{otherwise} \end{cases}$$

Given a failing test case $tc$, the goal of delta debugging is to find the minimal set $Q$ of features such that $tc|_Q$ fails. We denote the complement of $Q$ by $\bar{Q} = F \setminus Q$.

**Assumptions** Delta debugging relies on three assumptions: *(i)* test cases are *monotone*, i.e., if $tc|_Q$ fails, then for any superset $Q' \supseteq Q$ of features $tc_{Q'}$ also fails; *(ii)* test cases are *unambiguous*, meaning that for a failing test case $tc$ there is a unique minimal set $Q$ that causes the failure; and *(iii)* every subset of features is *consistent*, meaning that for any $Q \subseteq F$, $tc|_Q$ terminates with a definite fail or success result.

**Algorithm** Given a test case $tc$, delta debugging finds a minimal set of features $Q$ that causes a failure. Initially, $Q$ contains all enabled features in $tc$, i.e., $Q = \{f \in F \mid tc(f) \neq \perp\}$. Then it applies the following steps:

1. *Split:* Split $Q$ into $n$ partitions $Q_1, \ldots, Q_n$, where $n$ is the current granularity. Test $tc|_{Q_1}, \ldots, tc|_{Q_n}$ for failures. If some $tc|_{Q_i}$ fails, then use $Q_i$ as the new current set of features and continue with step 1.

2. *Complement:* If none of the new tests $tc|_{Q_1}, \ldots, tc|_{Q_n}$ fail, check the complement of each partition by testing $tc|_{\bar{Q}_1}, \ldots, tc|_{\bar{Q}_n}$. If some $tc|_{\bar{Q}_i}$ fails, then use $\bar{Q}_i$ as the new current set of features and continue with step 1.

3. *Increase Granularity:* If no smaller set of features is found and $n < |Q|$, then set $n$ to $\min(2n, |Q|)$ and continue with step 1.

4. *Terminate:* If it is not possible to split the current set of features into a smaller set, terminate and return $Q$.

## 6.2 Iterative Delta Debugging

In our setting, test cases are often ambiguous as a discrepancy often arises due to multiple faults in the network model. To this end, we apply the delta debugging algorithm *iteratively* and find all minimal sets of features that cause a given discrepancy. Intuitively, starting with a test case $tc$ with enabled features $Q$, we first apply the delta debugging steps (given

**Algorithm 1:** Iterative Delta Debugging

**Input** : Test case $tc$, initially enabled features $Q$ in $tc$.
**Output** : A set of minimal feature subsets $\mathcal{S} = \{Q_1, \ldots, Q_n\}$.

```
1  S = ∅
2  Queue = queue()
3  put(Queue, Q)
4  while ¬empty(Queue) do
5      H = head(Queue)
6      if run(tc|_H) = failure then
7          Q' = minimize(H)
8          for f in Q' do
9              put(Queue, H \ {f})
10         S = S ∪ {Q'}
11 return S
```

in §6.1) to find a minimal configuration feature set $Q'$ such that $tc|_{Q'}$ triggers the discrepancy. Then, we generate new test cases $tc_1, \ldots, tc_{|Q'|}$, by disabling a feature from $Q$ in each new test case $tc_i$, and iteratively apply delta debugging to these. We apply this process repeatedly until no further failing test cases are found. Once Metha identifies all minimal sets of configuration features that trigger a given bug, Metha creates a minimal configuration for the developer to reproduce it.

We present our iterative delta debugging algorithm in Algorithm 1. We start from a set of initially enabled features $Q$ in $tc$ and return all minimal subsets of $Q$ that trigger a discrepancy. We keep all sets of features to be checked in a queue and continue until the queue is empty (Line 2 - Line 4). For every set $H$ of features in the queue, we check if the test case $tc|_H$ triggers a discrepancy (Line 5, Line 6). If this is the case, then we find a minimal subset $Q \subseteq H$ of features that triggers the discrepancy using classic delta debugging, and create new subsets that need to be checked (Line 8, Line 9). For example, if we find a minimal set of features $Q = \{a, b\}$ that triggers the discrepancy, then we check if there are any other minimal sets of features that do not contain $a$ or $b$ (and are thus non-comparable to $Q$). We note that we generate two new sets of features $H \setminus \{a\}$ and $H \setminus \{b\}$ instead of a single one $H \setminus \{a, b\}$ because there may be overlapping discrepancies. For example, even though we know that $b$ can trigger a discrepancy with $a$, $b$ might also trigger a discrepancy with another feature $c$. Finally, the algorithm keeps all found minimal feature subsets and returns them (Line 10, Line 11). We conclude by stating the correctness of our algoirthm:

**Theorem 1.** *For any test case tc with enabled features Q, Algorithm 1 finds all minimal fault-inducing subsets of features.*

We present the proof of this theorem in App. A.

**Runtime** The running time of Algorithm 1 is $O(|Q|!)$. The worst-case behavior is when the size of the set $H$ of features is reduced by 1 element in each step, introducing $|H-1|$ new features sets to the set $\mathcal{S}$. To improve the running time, we cache (not shown in Algorithm 1) feature sets that have been added to the queue. This strictly reduces the algorithm's running time and yields a worst-case running time complexity of $O(2^{|Q|})$. We note that the running time in practice is reasonable as the reduction of the set $H$ by the delta debugging minimization step (Line 7) is significant (down to $2-3$ elements in practice).

**Limitations** As with classic delta debugging, there may be a fault in the interaction between a set of parameters, say $a$, $b$, and $c$, as well as a different fault in the interaction between a subset of these parameters, say $a$ and $b$. We cannot distinguish these two faults and will only identify the latter fault. However, once the identified fault is fixed, our algorithm will then identify the fault in the interaction among $a$, $b$, and $c$ as well, assuming it is still present in the verification tool.

## 7 System

We have fully implemented Metha in 7k lines of Python code.[2] This covers the entire testing pipeline from the input, the list of configuration features to be tested and the topology, to the outputs, the bug reports. In the following, we highlight key points of Metha's implementation, which consists of a vendor- and tool-agnostic core that uses runners to interface with the different network analysis and verification tools.

**Semantic constraints** To run the tests, Metha uses a logical topology, which consists of the physical topology extended with logical groupings. These groupings map the routers to BGP ASes and their interfaces to OSPF areas. This trivially ensures that the base configuration meets all the necessary semantic constraints (cf. §4.1). In a next step, Metha starts to randomly assign IP subnets to links and IP addresses to the router interfaces on these links. Specifically, every router is assigned a router ID, which is also assigned to the loopback interface of that router. Finally, Metha generates additional resources that are needed to test specific configuration features. For example, Metha adds several prefix-lists and static routes which can then be used in the test generation, for example, for a match statement of a route-map and route redistribution, respectively. All these additional resources are generated based on the predefined logical topology. Hence, a prefix-list, for example, will only consist of prefixes that are actually defined in the network, such that a route-map statement using that list for a match will also be reachable.

**Testing coordination** Once Metha laid the groundwork, it has to define a test strategy based on the specified configuration features. At the moment, Metha supports configuration features pertaining to four categories: static routes, OSPF, BGP and route-maps. As part of that, the system supports additional constructs such as prefix-lists and community-lists. These are currently not tested on their own, but added when needed to test the main features, such as route-maps. Metha

[2] Available at https://github.com/nsg-ethz/Metha

then uses all features and the logical topology to prepare the parameters to come up with the test suite. To do that, Metha passes all the parameters and their possible values to a state-of-the-art combinatorial testing tool: PICT [25]. PICT devises a test suite that consists of a set of tests ensuring complete coverage of all pairwise feature interactions.

**Configuration generation** A single test from the PICT test suite is an abstract network configuration. It simply specifies which feature and corresponding value needs to be activated and where (i.e., on which router and, if applicable, at which interface). Metha then translates the abstract network configuration to concrete device configurations using a grammar-based approach to ensure lexical and syntactical validity.

Metha implements a large portion of both Cisco IOS and Juniper grammars for which we relied on the respective official command references. This means Metha can generate both Cisco IOS and Juniper configurations for the tests. Metha even supports to test hybrid networks in which devices of both vendors are used at the same time.

**Testbed** Metha runs the generated configurations in parallel on both the tool under test and the oracle. After both of them converged, it retrieves the routing tables and compares them. Metha is able to test any tool that takes the device configurations as inputs and provides direct access to the computed routing tables out-of-the-box. Otherwise, Metha uses tool-specific runners to process the inputs such that they meet the tool's requirements and map the output back to Metha's format. Metha comes with runners for three well-known network analysis and verification tools: Batfish [17], NV [13] and C-BGP [28]. For NV, for example, Metha first has to compile the simulation program from the network configurations. As a source of ground truth, Metha uses a virtualized network running real device images of both Cisco and Juniper routers. It connects to these devices over Telnet and retrieves the routing tables (e.g., `show ip route` for Cisco devices). To ensure full convergence, Metha retrieves the routing tables every 10 seconds and proceeds once the tables have not changed for ten consecutive checks. With this setup, Metha allows to freely choose any oracle (e.g., hardware testbed) as long as it exposes the computed routing tables.

**Output** Finally, Metha localizes all bugs within a discovered discrepancy by relying on delta debugging. For every single bug, it generates a report highlighting the observed difference in the routing tables of the tool under test and the oracle, such as a mismatch in a route's metric or a missing route. This helps the developer understand the expected behavior. In addition, it identifies the configuration statements required to trigger the bug and comes up with a minimal network configuration to reproduce the bug. This allows Metha to provide actionable feedback to the developers of the tool, helping them to faster locate and understand the bug. The minimal configuration example can also be used as an extra test case for traditional system testing.

# 8 Evaluation

In this section, we evaluate Metha to address the following research questions:

- RQ1 How does Metha's semantical configuration generation, the search space reduction using boundary values and the test suite from combinatorial testing contribute to Metha's effectiveness? We show that Metha finds 20 bugs and achieves a higher combinatorial coverage than the random baseline, which only discovers 3 bugs with the same number of tests.
- RQ2 How many test cases does Metha need to localize all bugs in a single discrepancy between the tool under test and the oracle? Metha requires on average 14.1 test cases to isolate all the bugs causing a discrepancy.
- RQ3 Is Metha practical? We ran Metha on three different state-of-the-art network analysis and verification tools and found a total of 62 bugs, 59 of them have been confirmed by the respective developers.

## 8.1 Comparison to Random Baseline

We begin our evaluation by studying how the three components of Metha contribute to its effectiveness. To this end, we compare a random baseline to three versions of Metha: step-by-step, we enable each component starting with semantic Metha, then we add the reduction to boundary values, and finally, we use full Metha using combinatorial testing to define a test suite. The results show that the semantical configuration generation is the most fundamental part of Metha. Reducing the parameters to boundary values and applying combinatorial testing help to find additional bugs as both manage to increase the combinatorial coverage.

In the following, we introduce the four approaches:

**Random baseline** The random baseline relies on random syntactic test generation, meaning it uses a traditional grammar-based fuzzing approach. Thanks to the grammar, the configurations generated by the baseline are lexically- and syntactically-valid, but they are not necessarily semantically-valid: the baseline generates device configurations that are parseable and look realistic. However, the configurations might not always be practical: for example, referenced route-maps and prefix-lists do not always exist, and IP addresses on interfaces might not match those of their neighbors. Inter- and intra-device dependencies are not factored in.

**Semantic Metha** The initial Metha approach implements random semantic test generation. Similar to the random baseline, it uses a grammar-based fuzzing approach with the only difference that it ensures semantical validity within the configuration: while, for example, interface costs are completely random, other values are more constrained based on inter- and intra-device dependencies. This approach ensures, for example, that only defined route-maps are referenced, and that BGP sessions are configured with matching parameters.

| Approach | # Discovered Bugs |
|---|---|
| Random Baseline | 3 |
| Semantic Metha | 16 |
| Bounded Metha | 17 |
| **Full Metha** | **20** |

Table 2: Every component of Metha allows it to find more bugs with the same number of test runs.

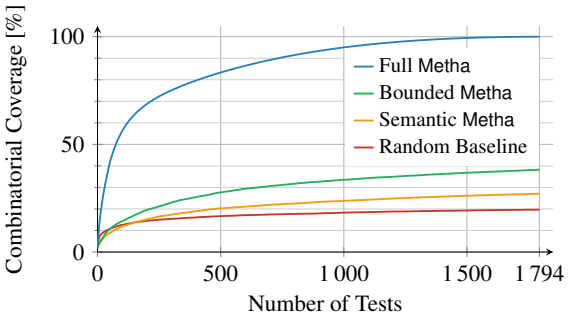



Figure 5: The achieved combinatorial coverage increases with every single component of Metha. Full Metha achieves complete combinatorial coverage.

**Bounded Metha** The bounded approach adds the reduction to boundary values as introduced in §4.2 to semantic Metha. This means instead of assigning completely random numeric values, the approach reduces the allowed values to three options: the minimum, the maximum, and a "normal" value, randomly chosen between the two extremes.

**Full Metha** Finally, we run the full testing system. We add combinatorial testing as introduced in §5 to define a test suite that maximizes combinatorial coverage on top of the semantic configuration generation and the boundary values reduction.

**Experiment setup** We ran all four approaches for the same number of tests and used them to test Batfish [17]. Whenever one of them detected a discrepancy between Batfish and the oracle, we applied the full fault localization procedure as described in §6 to detect the underlying bugs and the features causing it. Thanks to that, we are able to detect duplicates and count only the unique bugs that each approach discovered.

For all the tests, we used the same simple topology consisting of four routers connected in a star topology and tested configuration features belonging to the following four categories: static routes, BGP, OSPF, and route-maps. For the entire experiment, we used Cisco IOS configurations. For the given configuration features, combinatorial testing generated a test suite consisting of 1 794 tests. While the full Metha approach followed the test suite, the other approaches randomly chose the active configuration statements for every single test.

**Results** Table 2 shows the number of unique bugs that every approach found within the 1 794 test runs. The full Metha detected 20 unique bugs, while the random baseline only found 3 bugs. The semantic configuration generation is the most fundamental component of Metha. It comes as no surprise as without semantical validity, many of the configurations do not allow for any meaningful control plane computations and will not fully exercise the network model of the tool under test.

Boundary values and combinatorial testing allow finding 1 and 3 additional bugs within the 1 794 test runs, respectively. This is because both approaches achieve higher combinatorial coverage and therefore test a wider variety of features. These results show that the boundary values reduction strikes a good balance between testing different parameter values, while keeping the search space tractable. It is important to note that the detected bugs are inclusive, meaning that full Metha detected all 17 bugs that bounded Metha detected and 3 additional bugs. There is one exception: the baseline found a bug in the parser, which the other approaches did not find.

The random baseline is strong at discovering parser bugs since that is where grammar-based fuzzing excels. Two out of its three discovered bugs are parser bugs. In both cases, the problem was an, according to the specification, unsigned 32-bit integer being parsed as signed. For example, `ip ospf 100 area 3933914791` could not be parsed. Metha did not catch this bug as it uses fixed area numbers as part of the logical topology. By adding the area numbers to the set of configuration features being tested, Metha also finds this bug.

Fig. 5 shows the combinatorial coverage achieved by the four approaches, i.e., it shows the pairwise feature combinations covered during testing. We focus on feature instead of code coverage for two reasons: First, one can easily achieve high code coverage with random, semantically-invalid configurations. Second, code coverage is specific to the tool under test and makes it difficult to compare. To measure the combinatorial coverage of the random baseline and semantic Metha, we partitioned the input space in the same manner as we did for bounded Metha, i.e., into minimum, maximum, and middle values. Any configuration which did not specifically use the minimum or maximum value for a parameter was then considered as a middle configuration. Metha achieves full combinatorial coverage by design as it is guaranteed by combinatorial testing. These results underline the importance of semantically-valid configurations. While both the random baseline, which relies on syntactically-valid configurations, and semantic Metha achieve a similar combinatorial coverage, semantic Metha finds many more bugs as its configuration actually ensures control plane computations.

**Performance** Running a single test case took an average of two minutes. We run both the tool under test as well as the virtualized testbed in parallel and found that most of the time is spent waiting on the testbed to converge. The generation of a combinatorial test suite with PICT for the baseline network with 4 routers took an average of 6 minutes. Over the entire test suite, this time is negligible. Running the entire setup took us several days. The runtime depends highly on the number of discrepancies and the number of bugs causing them.

| | Bugs | | Type | | Feature Category | | | |
|---|---|---|---|---|---|---|---|---|
| | discovered | confirmed | crash | silent | OSPF | BGP | route-filter | other |
| **Batfish** [17] | 29 | 29 | 5 | 24 | 10 | 10 | 9 | 0 |
| **NV** [13] | 30 | 30 | 5 | 25 | 13 | 9 | 7 | 1 |
| **C-BGP** [28] | 3 | ? | 0 | 3 | 1 | 1 | 1 | 0 |

Table 3: Bugs discovered by Metha for Batfish, NV and C-BGP and classification.

## 8.2 Fault Localization

Whenever Metha detects a discrepancy between the routing tables of the tool under test and those of the oracle, it goes into fault localization to isolate all independent bugs. Fault localization relies on delta debugging (cf. §6) which creates additional test cases to identify the configuration statements causing the bugs. In the following, we evaluate its overhead, i.e., the number of additional test cases Metha had to create.

**Experiment setup** For this experiment, we ran Metha using the same topology as before and tested the full set of configuration features. Whenever Metha detected a discrepancy, we recorded the number of additional test cases required to find all independent bugs and the number of discovered bugs.

**Results** On average, Metha used 14.1 additional test cases to locate all bugs within a test case. The number of additional test cases ranged from as low as 7, to localize a single bug, up to as high as 58, to localize 5 independent bugs. The number of additional test cases mostly depends on the number of independent bugs within a single detected discrepancy. The number of configuration statements that actually cause the bug plays a minor role. Also, we have observed that the detected bugs are all caused by a few configuration statements (one or two), even though multiple configuration statements were active during the tests. This confirms the observation that bugs are often caused by the interaction of few features [20, 31] and shows that combinatorial testing is a useful technique in this setting.

## 8.3 Real Bugs

In addition, we showcase our end-to-end implementation of Metha by testing three different network analysis and verification tools: Batfish [17], NV [13], and C-BGP [28]. We show that Metha finds real bugs and report them in Table 3.

**Experiment setup** We ran Metha for several days on all three tools and with several different setups. Batfish is the most complete and advanced tool as it can handle configurations of many different vendors and supports a wide variety of configuration features. NV itself is an intermediate language for control plane verification that allows to build models of any routing protocols and their configurations. It provides simulation and verification abilities. We tested the simulation only, the discovered bugs, however, most likely also exist in the verification part as both rely on the same network model. For Batfish and NV, we used both Cisco IOS and Juniper configurations. C-BGP has its own configuration language.

**Results** As shown in Table 3, Metha found a total of 62 bugs. The developers of both Batfish and NV confirmed the discovered bugs to be real bugs. To better understand the nature of the bugs, we classified them by their type (i.e., whether they lead to a crash or go unnoticed) and by the configuration feature category itself (e.g., OSPF). Only a few of the bugs produce a clear error. This is most likely also because these are noticed more often and reported. The large majority of the bugs are silent semantic bugs which are extremely difficult to notice. These are the sneakiest bugs and can lead to false analyses and answers by the verifier. These bugs include all the configuration features discussed in §2 showing that they affect the analysis of commonly used features, such as route redistribution and aggregation, and named communities.

The bugs are distributed quite evenly among all tested parts of the network model. We did not find one specific protocol or configuration feature that is especially error-prone.

# 9 Discussion

**What about the testbed?** Metha detects bugs by looking for discrepancies between the tool under test and an oracle. For the oracle, Metha uses a testbed running real router firmwares. The testbed just needs to be large enough to fully exercise all configuration features. Normally, a small testbed of few routers suffices and also helps speed up the testing. In this paper, we rely on a virtualized testbed. To use a physical testbed instead, one simply has to change the SSH/Telnet configurations to connect to the physical devices.

A virtualized testbed comes with several advantages. It provides more flexibility in terms of the settings one can test and the time needed to setup. For example, there is no rewiring needed to test different topologies. In addition, it is very simple to test the same topology with a different device category or with devices from another vendor: one simply has to exchange the router image.

**What about more targeted tests?** Metha's test suite can be adjusted to the developers requirements by restricting the set of configuration features, adjusting the number of values per feature, and changing the number of interacting features. The tests required to cover the search space mainly depend on the

number of values per feature and the number of simultaneous feature interactions, while the set of features is secondary. By default Metha tests three values per feature and considers pairwise interactions. This choice strikes a good balance between the number of tests required and thorough testing, as our results confirm: Metha found all bugs that the random approaches discovered with fewer tests, despite using "only" the boundary values; and all discovered bugs are caused by one or two interacting configuration features, despite considering interactions of more than just two features.

Metha does not replace traditional unit and system testing, but provides an additional way to find latent bugs anywhere in the system. The advantage of Metha is that it requires minimal developer involvement and can be run alongside traditional tests without any additional effort. If desired one can run extensive tests by considering more elaborate feature interactions and more than three values per feature. Often with fuzz testing, one just lets the testing system run indefinitely and collect bug reports along the way.

## 10 Related Work

In this section, we first discuss current network analysis and verification tools. Then, we survey related work on testing static analyzers and verifiers, the various testing initiatives in the field of networking, delta debugging, and fuzz testing.

**Network analyzers & verifiers** Our work aims to facilitate the development of network analysis and verification tools through thorough testing. Over the years, we have seen a rise in tools that simulate networks [28], verify properties of networks and their configurations [3, 14, 19, 30], and tools that analyze aspects of networks [11, 12, 18, 23]. All of these tools have in common that they in some way or another use a network model to analyze and verify the network. Any bug or inaccuracy that exists within that network model undermines the soundness of the tools' results and analyses.

In contrast, CrystalNet [21] is a cloud-scale, high-fideltiy network emulator running real network device firmwares instead of relying on a network model. Hence, it accurately resembles the real network (e.g., vendor-specific behaviors and bugs in device firmwares are captured).

**Testing analyzers and verifiers** The problem of ensuring the correctness of analysis and verification tools is not specific to networks. In the field of static analysis, several works exist that pursue the same goal. Bugariu et al. [5] apply a unit testing approach, meaning they do not test the entire system but components thereof which simplifies the test generation. Since Metha treats the tool under test as a black box it cannot test certain components separately. Cuoq et al. [8] randomly generate input programs. This technique is mostly effective at testing the robustness of the analyzers. Similar to Metha, Andreasen et. al [1] apply delta debugging to find small input programs that help developers understand the bug faster.

**Testing in networking** Prior work on testing in networking has mainly focused on testing the network and its forwarding state [36], and SDN controllers [2, 6, 29].

Closest to Metha is Hoyan [32], a large-scale configuration verifier, in which the results of the verifier (i.e., network model) are continuously compared to the actual network for inaccuracies. It does so during operation and only covers cases that have actually occurred in the network. Metha in contrast proactively tests to detect the bugs before deployment.

**Delta debugging** In automated testing tools, delta debugging is a well-established technique [33, 35] that allows to automatically reduce a failing test case to the relevant circumstances (e.g., lines of code or input parameters). Over the years, researchers came up with several extensions to the general delta debugging algorithm, such as a hierarchical approach [26] that takes the structure of the inputs into account. It first explores the more important inputs allowing to prune larger parts of the input space and hence, requiring fewer test cases.

Traditional delta debugging finds one bug at a time even if the test case is ambiguous and exhibits multiple independent bugs. The developer then fixes one bug and reruns delta debugging to find the next. Metha automatically detects the causes of all independent bugs without developer involvement.

**Fuzz testing** Fuzz testing [15, 34] is an umbrella term for various testing techniques relying on "randomized" input generation. Metha uses a form of grammar-based fuzzing. Due to the complex dependencies within network-wide configurations, Metha first builds a basic configuration structure to ensure semantical validity. Then, it uses fuzzing to test different feature combinations restricted to that structure.

## 11 Conclusion

We presented Metha, an automated testing framework for network analysis and verification tools that discovers the bugs in their network models before deploying them to production. It does so by generating a wide variety of network configurations according to a test suite defined through combinatorial testing. Metha provides developers with actionable reports about all discovered bugs including a configuration to reproduce them. We implemented Metha and evaluated it on three state-of-the-art tools. In all tools, Metha discovered a total of 62 bugs, 59 of them have been confirmed by the developers. An interesting avenue for future work would be to extend Metha so that it can also test configuration synthesizers such as [4, 9, 10] as bugs in their models would render them useless.

## Acknowledgements

We thank our shepherd Michael Schapira and the anonymous reviewers for their insightful comments and helpful feedback. The research leading to these results was partially supported by an ERC Starting Grant (SyNET) 851809.

## References


[1] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic Approaches for Increasing Soundness and Precision of Static Analyzers. In *ACM SOAP*, Barcelona, Spain, 2017.

[2] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *ACM PLDI*, Edinburgh, United Kingdom, 2014.

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

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

[5] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. Automatically Testing Implementations of Numerical Abstract Domains. In *ACM ASE*, Montpellier, France, 2018.

[6] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. A NICE Way to Test OpenFlow Applications. In *USENIX NSDI*, San Jose, CA, USA, 2012.

[7] Inc. Cisco Systems. Redistributing Routing Protocols. https://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/8606-redist.html, March 2012. Accessed: 2020-09-12.

[8] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing Static Analyzers with Randomly Generated Programs. In *NFM*, Norfolk, VA, USA, 2012.

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

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

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

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

[13] Nick Giannarakis. NV - An Intermediate Language for Network Verification. https://github.com/NetworkVerification/nv, 2020. Commit: d058c4ce5c1549ad4e22d97cb01b8ea19d07741c.

[14] Nick Giannarakis, Devon Loehr, Ryan Beckett, and David Walker. NV: An Intermediate Language for Verification of Network Control Planes. In *ACM PLDI*, London, UK, 2020.

[15] Patrice Godefroid. Fuzzing: Hack, Art, and Science. *Communications of the ACM*, 63(2), 2020.

[16] Linghuan Hu, W Eric Wong, D Richard Kuhn, and Raghu N Kacker. How does combinatorial testing perform in the real world: an empirical study. *Empirical Software Engineering*, 25(4), 2020.

[17] Intentionet. Batfish. https://github.com/batfish/batfish, 2020. Commit: 95099bc5ad77af57d92c484e2e5634827f63e724.

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

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

[20] D Richard Kuhn, Dolores R Wallace, and Albert M Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering*, 30(6), 2004.

[21] Hongqiang Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. CrystalNet: Faithfully Emulating Large Production Networks. In *ACM SOSP*, 2017.

[22] Mallory Locklear. Google accidentally broke the internet throughout Japan. https://www.engadget.com/2017-08-28-google-accidentally-broke-internet-japan.html, August 2017. Accessed: 2020-09-12.

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

[24] Doug Madory. Widespread Impact Caused by Level 3 BGP Route Leak. https://blogs.oracle.com/internetintelligence/widespread-impact-caused-by-level-3-bgp-route-leak, November 2017. Accessed: 2020-09-12.

[25] Microsoft. PICT - Pairwise Independent Combinatorial Testing. https://github.com/microsoft/pict, 2020.

[26] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical Delta Debugging. In *ICSE*, Shanghai, China, 2006.

[27] Matthew Prince. August 30th 2020: Analysis of CenturyLink/Level(3) Outage. https://blog.cloudflare.com/analysis-of-todays-centurylink-level-3-outage/, August 2020. Accessed: 2020-09-12.

[28] Bruno Quoitin and Steve Uhlig. Modeling the Routing of an Autonomous System with C-BGP. *IEEE Network*, 19(6), November 2005.

[29] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B Terry, and George Varghese. Correct by Construction Networks Using Stepwise Refinement. In *USENIX NSDI*, Boston, MA, USA, 2017.

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

[31] W. Eric Wong, Xuelin Li, and Philip A. Laplante. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. *Journal of Systems and Software*, 133, 2017.

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

[33] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT FSE-7*, 1999.

[34] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. Fuzzing: Breaking things with random inputs. In *The Fuzzing Book*. Saarland University, 2020. Accessed: 2020-09-12.

[35] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2), 2002.

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

## A Proof of Theorem 1

*Proof.* By contradiction. Assume that there is a minimal subset $Q' \subseteq Q$ such that $tc|_{Q'}$ fails which is not returned in $\mathcal{S}$. We check at least one superset of $Q'$ for a failure since we will always check the initial set $Q$. Assume $C \supseteq Q'$ is a smallest superset of $Q'$ which is checked. By the assumption of monotonicity, $tc|_C$ must fail, therefore we will minimize $C$. If $C = Q'$, then we must minimize to $Q'$ since $Q'$ is assumed to be minimal, violating the assumption that $Q'$ is not returned by the algorithm. If $Q' \subset C$, then $C$ will either minimize to $Q'$ (again violating the original assumption that $Q'$ is not returned by the algorithm) or to a different minimal subset $P$. In this case, we generate additional sets to be tested. However, both $Q'$ and $P$ are minimal subsets of $C$, therefore $Q' \not\subset P$ and $P \not\subset Q'$. Since $Q' \neq P$, we know that there must be an element $e \in P$ which is not in $Q'$, i.e., such that $Q' \subseteq C \setminus \{e\}$. The set $C \setminus \{e\}$ is both strictly smaller than $C$ and will be added to the sets to check by the algorithm in Line 9 and therefore violates our assumption that $C$ was a smallest superset of $Q'$ which is checked. □