Meissa: Scalable Network Testing for Programmable Data Planes

Naiqian Zheng
Peking University

Mengqi Liu
Alibaba Group

Ennan Zhai
Alibaba Group

Hongqiang Harry Liu
Alibaba Group

Yifan Li
Alibaba Group

Kaicheng Yang
Peking University

Xuanzhe Liu
Peking University

Xin Jin
Peking University

ABSTRACT

Ensuring the correctness of programmable data planes is important. Testing offers comprehensive correctness checking, including detecting both code bugs and non-code bugs. However, scalability is a key challenge for testing production-scale data planes to achieve high coverage. This paper presents Meissa, a *scalable* network testing system for programmable data planes with *full* path coverage. The core of Meissa is a domain-specific *code summary* technique that simplifies the control flow graph of a data plane program for scalable testing without sacrificing coverage. Code summary decomposes a data plane program into individual pipelines, and summarizes each pipeline with a succinct representation. We formally prove that Meissa with code summary achieves *100% path coverage*. We use both open-source and production-scale data plane programs to evaluate Meissa. The evaluation shows that (*i*) Meissa is able to test production-scale data plane programs that cannot be supported by state-of-the-art efforts, and (*ii*) besides P4 code bugs, Meissa is able to not only identify known non-code bugs, but also detect previously-unknown non-code bugs. We also share in this paper several real cases tested by Meissa in a production programmable data plane.

CCS CONCEPTS

• **Software and its engineering** → **Formal language definitions**; • **Networks** → **Programmable networks**.

KEYWORDS

Formal Methods; Programmable Switches; P4 Testing

ACM Reference Format:
Naiqian Zheng, Mengqi Liu, Ennan Zhai, Hongqiang Harry Liu, Yifan Li, Kaicheng Yang, Xuanzhe Liu, and Xin Jin. 2022. Meissa: Scalable Network Testing for Programmable Data Planes. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3544216.3544247

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544247

1 INTRODUCTION

Bugs are detrimental to the performance, functionality, and reliability of production networks. Data plane programs used in production networks are highly complex. They compose many different match-action tables to implement a variety of functionalities with complex control flows. Despite the success of recent P4 program verification efforts [27, 56, 76, 79], program verification is fundamentally limited to code bugs, and cannot detect non-code bugs caused by the underlying target or toolchain.

Testing [19, 24, 53, 65, 72, 81, 87] offers comprehensive correctness checking, including detecting both code bugs and non-code bugs. The basic idea of testing is to generate input-output test cases for a given implementation, and then execute the implementation with each input to check if the actual output matches the desired output. In the context of testing network data planes, input-output test cases are generated in terms of input-output test packets. Input packets are injected into the switch and the output packets are captured to check whether they match the desired ones. Compared to program verification, because testing executes compiled data plane programs with test cases on actual hardware targets, it can detect non-code bugs in addition to code bugs.

Scalability is, however, a key challenge when applying testing to large-scale programs. To thoroughly test a program, it is desirable to generate a test case to cover every execution path. But the number of possible paths grows exponentially with the size of a program, which is widely known as the *path explosion* problem in the literature of software testing [20, 51, 75, 83, 84]. With regard to our problem, the advancements in research and practice of programmable networks have significantly increased the complexity of data plane programs. Modern data plane programs used in production networks span multiple pipelines and even multiple switches (§2), which are beyond the reach of standard testing techniques. As a concrete example, a data plane program used in our production networks has $O(10^4)$ lines of P4 code and $O(10^{197})$ possible paths. While existing work has applied testing to data planes [18, 28, 61, 68], they are not scalable to large data plane programs due to path explosion (§5).

In this paper, we present Meissa, a *scalable* network testing system for programmable data planes with *full* path coverage. Meissa scales to large multi-switch multi-pipeline data plane programs, which enables network developers and operators to test complex data plane programs used in real-world networks. More importantly, Meissa achieves so without sacrificing coverage. Meissa is a

rigorous approach that provides 100% path coverage guarantees, and is able to identify both code bugs and non-code bugs.

The core of Meissa is a domain-specific technique which we name *code summary* to simplify the control flow graph (CFG) of a data plane program for scalable testing without sacrificing coverage. Code summary exploits the structure of large multi-switch multi-pipeline data plane programs to decompose a data plane program into individual pipelines. It *summarizes* the CFG of each pipeline with a succinct representation, which significantly reduces the number of paths in each pipeline and addresses the path explosion problem for testing multi-switch multi-pipeline programs. The summarization is based on the observation that, only a small fraction of the paths in the CFG derived from a data plane program are valid (i.e., the end of a path can be reached by a packet) and thus need to be covered. At a high level, the summarization removes invalid paths, and compactly encodes each valid path with its constraints and variable values.

Code summary leverages two mechanisms—intra-pipeline redundancy elimination and inter-pipeline public pre-condition filtering—that combine local and global information to summarize the CFG of a pipeline. Intra-pipeline redundancy elimination analyzes the CFG of an individual pipeline, and removes invalid paths stemming from the code logic of the pipeline itself. Inter-pipeline public pre-condition filtering analyzes the paths from the entry point of the program to the target pipeline, and identifies the common conditions shared by these paths, which we call public pre-conditions of the target pipeline. Then Meissa filters the paths in the target pipeline that cannot be satisfied under the public pre-conditions. Code summary does not affect the code coverage properties. We formally prove that Meissa with code summary achieves 100% path coverage (§3.4).

We note that similar ideas like code summary have been proposed [33, 67] to generate test cases for general-purpose imperative languages. However, they are mainly targeted at programs featuring function calls and object-oriented programming, where the vertical composition of components renders it hard to simplify lower-layer library functions. In comparison, this paper focuses on programmable data planes, where components follow a pipelined layout both logically and on the implementation level. More specifically, we observe that engineers tend to design the data plane such that packets belonging to the same workload follow similar sequences of table execution, and hit the same entries in most tables. Meissa leverages this horizontal composition of components and adopts novel domain-specific techniques to significantly simplify each individual pipeline before generating test cases for the entire program.

We have implemented Meissa and used it in production. We use both open-source and production-scale data plane programs to evaluate Meissa. The evaluation shows that (*i*) Meissa is able to test production-scale data plane programs that cannot be supported by state-of-the-art efforts, and (*ii*) besides P4 code bugs, Meissa is able to not only identify known non-code bugs, but also detect previously-unknown non-code bugs. We also share our deployment experience with several real cases tested by Meissa in a production programmable data plane, including checksum fail-to-update, bf-p4c backend bug C (setValid), and misuse of optimization pragmas.

Finally, there is a common sentiment that while testing is more comprehensive in terms of the types of bugs it can detect, it is generally considered to have less coverage than verification. In this paper, we show that in the context of programmable data planes, testing is able to achieve full path coverage by the design of our domain-specific code summary technique. We further implement a practical system, perform an extensive evaluation, and deploy Meissa in production to demonstrate this point.

2 MOTIVATION

2.1 Testing Data Plane Programs

Bugs affect the performance and functionality of production networks, and even turn down the entire network. Finding bugs is critical for the development process of data plane programs. Despite the success of recent data plane program verification systems [27, 56, 76, 79], they are fundamentally limited to code bugs. Non-code bugs cannot be detected by verification efforts. Ruffy *et al.* [68] reported 59 bugs from P4C, 5 bugs from BMv2, and 32 bugs from bf-p4c. Issues caused by the underlying target or toolchain (e.g., compiler and driver API) are especially frustrating and difficult to debug. For example, in a real service failure event occurred in our production network, the operators observed incorrect parsing results in the data plane led to memory fault in the control plane, making the entire gateway load balancing service (implemented in P4) down. Sadly, the P4 program running on this failed gateway has been carefully verified. In many-hour analysis and troubleshooting, the operators localized the problem with the vendor's help—it was caused by a bug in the P4 compiler. Because of incomprehensive bug checking, verification does not check whether the program hits a compiler bug.

Testing data plane programs. In order to comprehensively guarantee the correctness of programmable data planes, we decided to build a testing system. In principle, testing techniques generate input-output test cases for a given implementation and tests whether the implementation can pass them. In the context of programmable data planes, the inputs and outputs are packets, and testing a data plane program is to inject input packets to a switch and test whether the output packets match the desired behaviors.

Manually generating test cases for a production-scale data plane is not practical. A typical automatic program test case generation approach is to construct a CFG and then enumerate the paths in the CFG [8, 18, 61]. For each path, symbolic execution can be used to traverse the path and check whether the path is *valid*, *i.e.*, whether the end of the path can be reached by an input. For a valid path, symbolic execution can be used to generate a *test case template*, which specifies the *pattern* of inputs that can trigger this path and the pattern of outputs at the end of the path. One or more input-output test cases can be generated based on the template for a path.

2.2 Key Challenge: Scalability

Scalability is a key challenge for testing data plane programs. Real-world data plane programs implement a wide variety of features and protocols [7, 46, 59, 62, 95]. Innovations in networking are continuously producing new features and protocols [15, 52, 55, 101, 102].

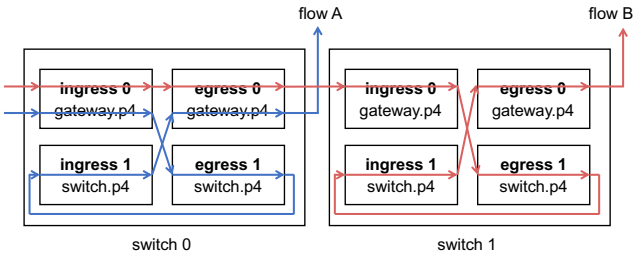


Figure 1: An example multi-switch multi-pipeline data plane.

Data plane programs do not simply contain a single forwarding table with longest-prefix matching. Instead, each feature or protocol includes one or more match-action tables, and a data plane program uses a complex control flow to compose many features and protocols that controls which set of tables is used to process an arrival packet and in which order.

In addition, modern switching ASICs have multiple ingress and egress pipelines connected by a traffic manager, which allows users to develop increasingly complex data plane programs. When a packet leaves an ingress pipeline, it can be sent to any egress pipeline via the traffic manager, and when it leaves an egress pipeline, it can go back to one of the ingress pipelines via internal loopback. A production data plane program can compose multiple pipelines together to implement more functionalities and utilize more hardware resources. For example, Figure 1 shows two multi-pipeline switches for a production edge network scenario. Each switch has two ingress pipelines and two egress pipelines. Ingress pipeline 0 and egress pipeline 0 implement custom gateway functionalities for encapsulation, decapsulation and statistics. Ingress pipeline 1 and egress pipeline 1 implement standard switch functionalities [7]. The packets of flow A are processed by multiple pipelines in switch 0, following a path ingress 0 → egress 1 → ingress 1 → egress 0. The scale of such a multi-pipe data plane program is in the order of O(1K) to O(10K) lines of P4 code (§5). It is challengable to test these complex data plane programs.

Even worse, production networks deploy multi-switch data plane programs, where the functionalities of a data plane can span multiple pipelines across multiple switches. Multi-switch data planes are deployed due to operational needs for reliability and performance and the constraints of hardware resources of a single switch. In Figure 1, the two switches form a multi-switch multi-pipeline data plane for our production edge network. The traffic is split between the two switches, which increases the total bandwidth. While flow A is only processed by switch 0, flow B is processed by both switches. The packets of flow B traverses a path with ingress 0 → egress 0 in switch 0 and then ingress 0 → egress 1 → ingress 1 → egress 0 in switch 1. The two switches serve as the backup of each other, which improves the reliability of the service. Multi-switch data planes further increase the complexity of the program. A production multi-switch data plane program used in our production networks contains more than 20K lines of P4 code (§5). Several solutions have been proposed for testing data plane programs [8, 18, 61]. They follow the general approach described in §2.1. They analyze a given program to generate test cases to cover execution paths of the program. They cannot scale to multi-switch multi-pipeline data plane programs (§5).

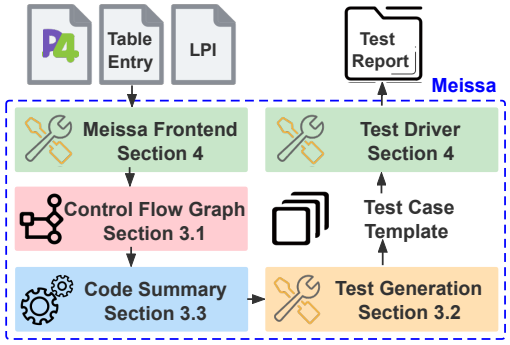


Figure 2: An overview of Meissa architecture.

field_id	::=	('pkt.' | 'hdr.') string	Header field name
aop	::=	'+' | '-' | '&' | '|'	Arithmetic operators
aexp	::=	field_id	Header field variable
	|	int	Concrete value
	|	aexp aop aexp	Arithmetic operations
bop	::=	'&&' | '||' | '~'	Boolean operators
cop	::=	'==' | '!=' | '>' | '<'	Comparison operators
bexp	::=	'True'	
	|	'False'	
	|	aexp cop aexp	aexp comparison
	|	bexp bop bexp	bexp composition
stmt	::=	field_id '←' aexp	Action
	|	assume bexp	Predicate
V	::=	$\{v_0, ..., v_n\}$	Set of vertices
succ	$\in$	$V \rightarrow 2^V$	Succ function
code	$\in$	$V \rightarrow$ stmt	Associated stmt
G	::=	$(V, v_0, succ, code)$	CFG
path	::=	ϵ	Empty path
	|	v :: path	Path concatenation

Figure 3: Syntax of the CFG.

3 MEISSA DESIGN

Meissa overview. Figure 2 presents an overview for Meissa's workflow. Developers express their high-level intents with LPI (Language for programmable network Intent) [79]. LPI is a state-of-the-art declarative specification language used to describe interesting properties of programmable data planes. Meissa takes (*i*) a specification expressed in LPI, (*ii*) a data plane program in P4, and (*iii*) a table rule set as inputs, and encodes them into a control flow graph (§3.1) by the frontend (§4). Then, our domain-specific *code summary* (§3.3) technique simplifies the control flow graph. After that, our test case generation framework (§3.2) produces test case templates with 100% path coverage. Finally, our test driver (§4) injects test packets to the switches under test, and compares the actual output packets with the expected ones. Meissa reports passed and failed test cases to the developer as the testing result. We formally prove that Meissa achieves 100% path coverage (§3.4).

3.1 Control Flow Graph (CFG)

Meissa converts a data plane program to an intermediate representation for test case generation. The intermediate representation

$$\frac{\langle aexp_1, s\rangle \to v_1 \quad \langle aexp_2, s\rangle \to v_2}{\langle aexp_1\ aop\ aexp_2; s\rangle \to v_1\ aop\ v_2}$$ Arithmetic expr

$$\frac{\langle bexp_1, s\rangle \to b_1 \quad \langle bexp_2, s\rangle \to b_2}{\langle bexp_1\ bop\ bexp_2; s\rangle \to b_1\ bop\ b_2}$$ Boolean expr

$$\frac{\langle aexp_1, s_1\rangle \to val}{\langle id_1 \leftarrow aexp_1; s_1\rangle \to s_1[id_1 \leftarrow val]}$$ Action stmt

$$\frac{\langle bexp_1, s\rangle \to True}{\langle assume\ bexp_1; s\rangle \to s}$$ Predicate stmt

$$\frac{\langle code(v), s_1\rangle \to s_2}{\langle v :: path; s_1\rangle \to \langle path; s_2\rangle}$$ Sequential evaluation

Figure 4: Evaluating statements along a path.

represents a data plane program as a control flow graph (CFG). Figure 3 shows the syntax of the CFG. A CFG G contains a set of nodes V and a special entry point v_0, where $v_0 \in V$. Each node v has 0, 1, or multiple successors, denoted by $succ(v)$. Each node v is associated with a statement $stmt(v)$. A statement operates on a set of header variables ($field\_id$), which represents a particular slice of data in the packet. A special variable is used to specify the sequence of headers present in the packet, which we omit in Figure 3 for brevity.

There are two types of nodes in the graph, depending on the type of the associated statement. One type is `predicate`, and the other is `action`. A `predicate` node represents constraints on the packet header, *e.g.*, $etherType == 0x0800$ (it is an IPv4 packet). `Predicate` nodes correspond to the branching statements (*e.g.*, if-else statements) in the control block and the match fields in the match-action table rules in a P4 program. An `action` node represents operations on the packet header, *e.g.*, $dstIP \leftarrow$ 192.168.0.1 (assign 192.168.0.1 to destination IP). `Action` nodes correspond to the action fields in the match-action table rules in the program.

Note that the CFG generated from a P4 program is acyclic, since we can always unroll recursive structures because the depth of recursion is bounded. This entails that all possible transitions in the CFG are of finite length.

Definition 1 (Possible path in a CFG)**.** A possible path π in a CFG $G(V, v_0, succ, code)$ consists of a sequence of vertices $v_{\pi_0} \rightsquigarrow v_{\pi_1}, ..., \rightsquigarrow v_{\pi_k}$, such that each step in the path follows the $succ$ relation,

$$\forall i < k, v_{\pi_i} \in V \wedge v_{\pi_{i+1}} \in succ(v_{\pi_i})$$

and the path spans from v_0 to an ending vertex

$$v_{\pi_0} = v_0 \wedge succ(v_{\pi_k}) = \emptyset.$$

$\Pi(G)$ denotes the set of *all* possible paths in G.

Concrete execution in the CFG: valid path. We use s to denote a concrete execution state, which is a mapping from header field variables to their corresponding concrete values

$$s \in field\_id \to int.$$

Figure 4 depicts the evaluation rules along a path. The evaluation of an action statement results in an updated state. The predicate statement is simply skipped if its condition evaluates to true. Otherwise, there is no valid rule for evaluating a false predicate, *i.e.*, the execution does not make sense if the guarding condition is not met. Finally, the evaluation along a path consists of sequential evaluation of the corresponding statement of each node.

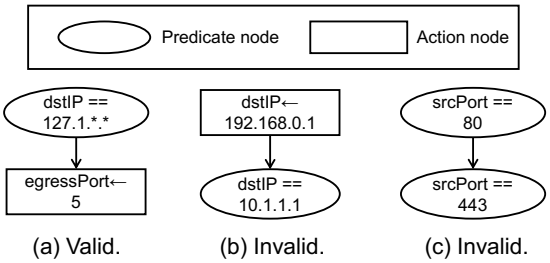


Figure 5: Examples of valid and invalid paths.

Definition 2 (Valid path in a CFG)**.** A valid path in a CFG, $G(V, v_0, succ, code)$, is a possible path π, such that there exists a concrete initial state that evaluates along this path

$$valid\_path(\pi, G) \equiv \left(\pi \in \Pi(G) \wedge \exists s_1, s_2, \langle \pi; s_1\rangle \to s_2\right)$$

3.2 Basic Test Case Generation Framework

Meissa employs symbolic execution [50] to analyze the CFG and generate test cases. The basic idea amounts to enumerating all possible paths in the CFG. For each path, Meissa evaluates the effects of its execution using symbolic values and checks whether the conjunction of all guard conditions is satisfiable or not, *i.e.*, whether the path is valid or not. Meissa generates a test case template to produce input packets that execute along each valid path. Figure 5 demonstrates (non-exhaustively) different path patterns and their satisfiability.

- Figure 5(a) is a valid path, *e.g.*, when $dstIP$ is 127.1.2.3, the condition evaluates to True. This pattern can be seen in the execution of a table, where the action is performed after key matches.
- Figure 5(b) is an invalid path because $dstIP ==$ 10.1.1.1 evaluates to False after assigning 192.168.0.1 to $dstIP$. This is a common pattern where a table matches on keys that are modified by a previous action.
- Figure 5(c) is an invalid path because $srcPort == 80 \wedge srcPort ==$ 443 cannot hold. This pattern is common among adjacent branching conditions or tables.

Meissa uses depth-first search (DFS) to enumerate all possible paths in the CFG. Notice that the choice of enumeration methods (DFS, breadth-first search, *etc.*) is orthogonal to the design of Meissa.

Symbolic execution in the CFG. During the symbolic execution along a possible path, Meissa maintains two stacks, the value stack V and the condition stack C. V maps each header field to either a concrete value or a symbolic arithmetic expression, and C is a boolean expression representing constraints on symbolic variables.

$$V \in field\_id \to aexp$$
$$C := bexp$$

Figure 6 depicts how the above two variables are updated along a possible path. Here, $[\![V]\!]a$ denotes the result of substituting all variables in expression a with values defined by V. For action statements, Meissa updates V to record the variable's current value. For

$$[\![V]\!]id \to V(id) \quad \text{Subst-var}$$

$$[\![V]\!]val \to val \quad \text{Subst-int}$$

$$\frac{[\![V]\!]a_1 \to a_1' \quad [\![V]\!]a_2 \to a_2'}{[\![V]\!](a_1\ op\ a_2) \to a_1'\ op\ a_2'} \quad \text{Subst-aexp}$$

.....

$$\frac{[\![V]\!]a_1 \to a_1' \quad V' \equiv V[id_1 \leftarrow a_1']}{\langle id_1 \leftarrow a_1; (V, C)\rangle \to (V', C)} \quad \text{Sym. action}$$

$$\frac{[\![V]\!]b_1 \to b_1' \quad SAT(C \wedge b_1')}{\langle assume\ b_1; (V, C)\rangle \to (V, C\ \&\&\ b_1')} \quad \text{Sym. predicate}$$

$$\frac{\langle code(v), (V_1, C_1)\rangle \to (V_2, C_2)}{\langle v :: path; (V_1, C_1)\rangle \to \langle path; (V_2, C_2)\rangle} \quad \text{Sequential sym. eval}$$

Figure 6: Symbolic evaluation along a possible path.

Algorithm 1 Basic Test Case Generation with DFS

- C: condition stack
- V: value stack

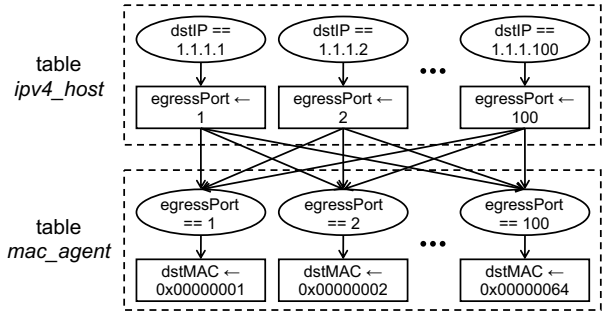
```
1:  function DFS(node)
2:      if node.type == PREDICATE then
3:          C.update(node)
4:          if Z3.solve(C, V) == SATISFIABLE then
5:              if node.children == null then
6:                  GenerateTestCaseTemplate()
7:              else
8:                  for child ∈ node.children do
9:                      DFS(child)
10:         C.restore()
11:     else if node.type == ACTION then
12:         V.update(node)
13:         if node.children == null then
14:             GenerateTestCaseTemplate()
15:         else
16:             for child ∈ node.children do
17:                 DFS(child)
18:         V.restore()
```

predicate statements, Meissa simply accumulates the guard condition in C. SAT($C \wedge b_1'$) is an early termination technique which we explain later.

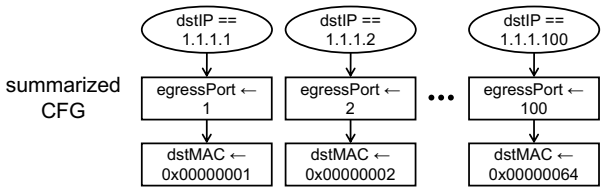
Validity checking of a path is done by feeding the accumulated guard conditions to an SMT solver (*e.g.*, Z3). If they are satisfiable, a test case template is generated for this particular path. For example, the test case template for Figure 5(a) is $dstIP = 127.1.*.*$, meaning all inputs satisfying this prefix will execute along the exact path.

Path pruning with early termination. As shown in the Sym. Predicate rule in Figure 6, Meissa adopts early termination [61] to prune the paths and reduce the search space. For example, if a previous predicate restricts the packet to IPv4, causing a further predicate `hdr.ipv6.isValid()` to never satisfy, then there is no need to continue the enumeration since the prefix is already invalid.

Algorithm 1 shows the pseudocode of the basic test case generation with DFS and early termination. For each node, based on



(a) 10000 paths before code summary.



(b) 100 path after code summary.

Figure 7: Intra-pipeline redundancy elimination.

whether the node type is a `predicate` or `action` (line 2 and 11), Meissa updates the condition stack C or value stack V, respectively (line 3 and 12). When the search reaches a leaf node of the graph, Meissa generates a test case template (line 5-6 and 13-14). Otherwise, Meissa visits each child node of the current node (line 7-9 and 15-17). Early termination is implemented by checking the satisfiability at each `predicate` node in the DFS and only proceeds if the path is satisfiable (line 4). Finally, Meissa restores the stacks and backtracks to other branches (line 10 and 18).

A potential drawback of early termination is that it may result in more calls to the SMT solver, and thus more overhead. This concern is addressed by adopting *incremental solving*, supported by state-of-the-art SMT solvers [12, 26], which allows users to push (save) and pop (restore) conditions. Meissa pushes an additional constraint into the SMT solver on a `predicate` node, and pops when it backtracks. The solver reuses intermediate results from previous invocations since most constraints stay the same.

3.3 Code Summary

The basic test case generation framework works for a single pipeline. However, production-scale data plane programs often consist of multiple switches with multiple pipelines. The composition of pipelines compounds the complexity of the search space exponentially, rendering its test case generation intractable. We develop a domain-specific *code summary* technique to address this scalability issue.

Key ideas. At a high level, code summary decomposes a data plane program to individual pipelines, and *summarizes* the CFG of each pipeline with a succinct representation to address the path explosion problem. For each pipeline, Meissa significantly reduces its number of possible paths by pruning (non-exhaustively) invalid ones, such that the composition of pipelines becomes tractable.

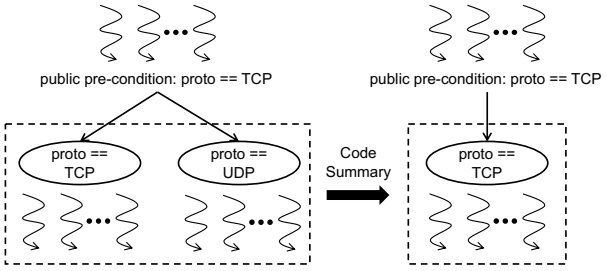


Figure 8: Inter-pipeline pre-condition filtering.

Overall, code summary reduces the time complexity of test case generation from $O(n^k)$ to $O(kn)$, where k denotes the number of pipelines and n denotes the number of possible paths within each pipeline. A detailed analysis is in Appendix A.

Code summary leverages two mechanisms to summarize the CFG of each pipe, which are intra-pipeline redundancy elimination and inter-pipeline public pre-condition filtering. (i) Intra-pipeline redundancy elimination analyzes the CFG of an individual pipeline, and eliminates invalid paths that stem from the code logic of the pipeline itself. Consider a pipeline with a table *ipv4_host* followed by a table *mac_agent*. As shown in Figure 7a, *ipv4_host* sets *egressPort* according to *dstIP*, and *mac_agent* sets *dstMAC* according to *egressPort*. There are 100 rules in each of the two tables, resulting in 10,000 possible paths in total. However, only 100 paths are valid, because after assigning *egressPort* in *ipv4_host*, only one rule in *mac_agent* can be matched. Figure 7b is the summarized CFG of this example which reduces the paths from $n = 10,000$ to $m = 100$. (ii) Inter-pipeline public pre-condition filtering analyzes all paths from the entry point of the program to a given pipeline, and identifies public pre-conditions that all paths have in common. Then, it uses the public pre-conditions to prune paths in the given pipeline. Figure 8 shows an example where $proto == TCP$ is a public pre-condition for all paths to the pipeline. Since the predicate $proto == UDP$ can never satisfy, all paths following this node are discarded. Together, the two techniques leverage both local and global information to reduce the overall search space.

Algorithm. Algorithm 2 shows the pseudocode of test case generation with code summary. Since we retain information about the entry and exit of each pipeline, the algorithm first identifies each pipeline in the CFG, performs topological sort on them (line 2), and sets the order for subsequent processing (line 3-23). This ensures that a pipeline is summarized *if and only if* all its predecessor pipelines are already summarized.

To summarize a pipeline, the algorithm first computes its public pre-conditions (line 4-7). It gathers all valid paths from the entry point of the CFG to that of the pipeline (line 5). Because of the topological sorting, all pipelines along the path are already summarized to reduce the search overhead. Then it merges all valid paths to produce a new pair of value stack V and condition stack C (line 6-7). The intuition is to compute what all these paths entail in common, such that this new state encapsulates all possible states before the merging. Finally, it uses the above public pre-condition as the initial state to perform symbolic execution within the target pipeline, collecting all valid paths in *paths* (line 8-9).

The algorithm summarizes the pipeline by compactly encoding the valid paths (line 10-23). The encoding of a valid path contains (i) the guard constraints for entering this path, and (ii) the overall effects of executing this path. Computation of the guard constraints is trivial. It is a conjunction of all boolean constraints collected along this path (line 14-15), represented by a `predicate` node. On the other hand, even though obtaining the overall execution effects through symbolic execution is straightforward, encoding them using a group of `action` nodes is tricky. For example, assume that the symbolic execution along a path results in a final value stack V', where

$$\begin{aligned} V'(srcPort) &= 10000 \\ V'(dstPort) &= srcPort + 1 \end{aligned}$$

Naively encoding it as two assignment statements does not work, since it results in assigning 10001 to $dstPort$. This is because V' represents simultaneous updates to multiple variables, while the assignment statements in a CFG lack this kind of atomicity. Instead, Meissa introduces *auxiliary variables* to address this issue. The above example is encoded as the following statements, where $@srcPort$ denotes the value of $srcPort$ at the entry of this pipeline.

$$\begin{aligned} @srcPort &\leftarrow srcPort \\ srcPort &\leftarrow 10000 \\ dstPort &\leftarrow @srcPort + 1 \end{aligned}$$

Concatenating the guard constraints with the overall effects concludes the encoding of one valid path.

After all pipelines are summarized and replaced, the algorithm performs a DFS from the entry point of the CFG using Algorithm 1 to generate test case templates.

3.4 Code Coverage Guarantee

An important notion in software testing is code coverage [10], which is a measure of how well a given program is tested. There are several metrics for coverage, including path coverage, branch coverage, statement coverage and method coverage. We focus on path coverage because it is the strongest metric among them and is widely-used to thoroughly test programs in practice. Once we achieve 100% path coverage, we get 100% coverage in other metrics as well. We formally prove that Meissa achieves 100% path coverage.

Definition 3 (Full path coverage)**.** A test case generation framework generates a group of inputs for a given CFG G, denoted by a group of constraints $\{\beta_0, \beta_1, ..., \beta_m\}$. Such generation is full-path-coverage if and only if it covers every valid path in G, *i.e.*,

$$\begin{aligned} &\forall\, \pi, valid\_path(\pi, G) \implies \\ &\left(\exists k, \forall s, \beta_k(s) \implies \exists s', \langle \pi; s\rangle \rightarrow s'\right) \end{aligned}$$

Basic framework achieves 100% path coverage. As a baseline, the basic test generation framework (naive DFS) achieves full path coverage. Appendix B formally proves this theorem. In addition to full path coverage, the basic framework also guarantees that each path condition it collects corresponds to a distinct path, such that generating test cases for all path conditions guarantees a full coverage test.

Code summary achieves full path coverage. Code summary is an optimization on top of the basic framework that avoids redundant DFS searching within individual pipelines. The intuition

Algorithm 2 Test Case Generation with Code Summary

```
// Summarize CFG
pipelines ← TopologicalSort(CFG)
for pipeline ∈ pipelines do
    // Compute public pre-conditions
    paths ← get paths from CFG.entry to pipeline.entry
    C ← ∩_{p∈paths} p.constraints
    V ← ∩_{p∈paths} p.values
    // Find paths in the pipeline
    paths ← get paths in pipeline with C and V
    // Summarize the pipeline
    pipeline.clear()
    for path ∈ paths do
        Initialize summarizedPath
        summarizedPath.append(
          new PredicateNode(AND(path.constraints)))
        for var ∈ path.variables do
            if var changes value on path then
                summarizedPath.append(
                  new ActionNode(@var, var))
        for var ∈ path.variables do
            if var.value symbolic changed then
                SubstituteWithInit(var.value)
            summarizedPath.append(
              new ActionNode(var, var.value))
        pipeline.add(summarizedPath)
// Generate test cases on summarized CFG
DFS(CFG.entry)
```

is that, despite that the summary operation prunes possible paths within each pipeline, it preserves all valid ones such that the set of valid paths does not change.

Assume that the switch consists of n pipelines, denoted in topological order as $ppl_1, ppl_2, ..., ppl_n$. Each pipeline is a single-entry single-exit subgraph in the CFG G. Pipelines are ordered such that there is no possible path from ppl_j to ppl_i if $j > i$, and Algorithm 2 summarizes each of them following this topological order. We prove the following loop invariant.

Definition 4 (Summarizing k pipelines preserves all valid paths)**.** We denote the original CFG and the CFG after summarizing k pipelines as G_0 and G_k, respectively. For each valid path π_0 in G_0 with path condition C and final symbolic state V, there must exist a unique valid path in G_k which satisfies the same path condition.

$$INV(G_0, G_k) \equiv$$
$$\forall \pi_0, valid\_path(\pi_0, G_0) \wedge (\forall s, C(s) \implies \langle \pi_0; s\rangle \rightarrow [\![V]\!]s) \implies$$
$$\exists! \pi_k, valid\_path(\pi_k, G_k) \wedge (\forall s, C(s) \implies \langle \pi_k; s\rangle \rightarrow [\![V]\!]s)$$

Assume that such invariant holds for $INV(G_0, G_k)$. We prove that summarizing one more pipeline, ppl_{k+1}, preserves the invariant, *i.e.*, $INV(G_0, G_{k+1})$ holds. In particular, we focus on the valid path $\pi_0 \in \Pi(G_0)$ and $\pi_k \in \Pi(G_k)$, and prove the existence of a corresponding valid path $\pi_{k+1} \in \Pi(G_{k+1})$.

We assume that path π_k overlaps with ppl_{k+1}. Otherwise, it is not affected by the summarization and the invariant trivially holds. Without loss of generality, we partition π_k as $v_0... \rightsquigarrow v_s... \rightsquigarrow v_e... \rightsquigarrow v_f$, where v_s and v_e denotes the entry and exit points of ppl_{k+1}, respectively. We also assume a concrete execution, where the state transitions from s_0 to s_s, s_e, s_f, respectively.

Lemma 1 (Public pre-condition encapsulates all valid paths to v_s)**.** Any valid concrete execution from v_0 to v_s must be included in the public pre-condition. In particular,

$$C_{pub}(s_0) \wedge \forall id, ((([\![V_{pub}]\!]s_0)(id) = s_s(id)) \vee (([\![V_{pub}]\!]s_0)(id) = \star)$$

Proof. Here, $(([\![V_{pub}]\!]s_0)(id) = \star$ means V_{pub} does not specify the value of id. Assume that the transition from s_0 to s_s corresponds to path constraint C_s and final symbolic state V_s, where $C_s(s_0) \wedge [\![V_s]\!]s_0 = s_s$.

- Since C_{pub} only contains constraints from C_s that are common among other valid paths, $C_{pub}(s_0)$ must hold.
- Since V_{pub} only contains assignments from V_s that are common among other paths, it either agrees with V_s or does not specify any value.

□

Lemma 2 (Symbolic execution within ppl_{k+1} finds the partial path $v_s \rightsquigarrow v_e$ to be valid)**.** Since the basic symbolic execution discovers all valid paths (Appendix B), the proof obligation is to show that $v_s \rightsquigarrow v_e$ is indeed valid.

$$\exists s_1, s_2, \langle v_s... \rightsquigarrow v_e; s_1\rangle \rightarrow s_2$$

Proof. It is trivial to show that $\langle v_s... \rightsquigarrow v_e; s_s\rangle \rightarrow s_e$, since the execution within a CFG is deterministic. Further, by Lemma 1, the initial state s_s satisfies the public pre-condition (C_{pub}, V_{pub}). Thus, this path is valid and must be discovered by the symbolic execution within ppl_{k+1}. □

Lemma 3 (Replacing ppl_{k+1} with its summary preserves the valid path)**.** Though the partial path $v_s \rightsquigarrow v_e$ is modified by the summarization, it must still allow the same state transition. We use $v_s \hookrightarrow v_e$ to denote this summarized path.

$$\langle v_0... \rightsquigarrow v_s \hookrightarrow v_e; s_0\rangle \rightarrow s_e$$

Proof. It is trivial to show that $\langle v_0... \rightsquigarrow v_s \hookrightarrow v_e; s_0\rangle \rightarrow \langle v_s \hookrightarrow v_e; s_s\rangle$, since the execution before v_s does not change.

By Lemma 2, the state transition from s_s to s_e is discovered by the local symbolic execution within ppl_{k+1}, and the overall assignment effect is captured by a final symbolic state. Algorithm 2 encodes this effect such that all action nodes represent changes against s_s. Thus, $\langle v_s \hookrightarrow v_e; s_s\rangle \rightarrow s_e$. □

As the execution after v_e stays the same, $INV(G_0, G_{k+1})$ holds. This concludes the proof of the loop invariant.

Corollary 1 (Code summary achieves full path coverage)**.** Given a CFG G, for any valid path $\pi \in \Pi(G)$, it must be discovered by Algorithm 2, and the reported path condition C satisfies the following

$$\forall s, C(s) \implies (\exists s', \langle \pi; s\rangle \rightarrow s')$$

Proof. Algorithm 2 sorts and summarizes all pipelines one by one, transforming the original CFG to G_n. By the loop invariant proved above (Definition 4), G_n preserves all valid paths and corresponding path constraints from G.

Finally, Algorithm 2 invokes the basic symbolic execution on G_n to output all path constraints. Since any valid path will be

discovered and reported with the correct constraint (Appendix B), this theorem holds. □

4 IMPLEMENTATION

We have built and deployed Meissa in production. Meissa is implemented in JAVA with 16,000 lines of code (LOC), where 12,500 for frontend (encoding p4 intermediate representation), and 3,500 for backend (test case template generation). Besides P4 programs, our implementation allows the integration of manually-encoded components, such as encoding of DPDK [6] programs. We have used it to test a hardware-software (P4-DPDK) co-designed gateway.

The test driver consists of a sender, a receiver and a checker. The sender reads each test case template and generates concrete packets satisfying these constraints. After these packets are injected to the switch, they follow the corresponding execution path and are captured by the receiver. Each packet carries a unique ID in its payload, such that the checker can relate the sent packet with the received one (or mark as absent). The checker validates packet checksums, checks whether the sent and received packets follow the high-level intent, and reports test results.

Meissa supports recirculation, resubmission, and multi-pipelines, which can be unrolled into an acyclic graph. Most of data plane programs already satisfy this requirement so as to achieve line rate. Operators claim the code and table entry set of each pipeline in the specification. They also depict topology among pipelines and traffic manager policies. Meissa parses the specification, code and table entry sets of each pipeline, encodes them into a directed acyclic control flow graph. Recirculation and resubmission are similar to multi-pipelines, because operators manually name unrolled pipelines. However, Meissa is not able to test stateful behaviors caused by recirculation or resubmission, such as modifying the same register in different rounds.

Hashing is a widely-used P4 functionality, but it is not well supported by the state-of-the-art SMT solvers. Our symbolic execution can not directly push it into condition stack or value stack, because the SMT solver calculate path satisfiability with these two stacks. Thus, we directly calculate hashing results if all keys are constrained with one value, and otherwise leave these fields as arbitrary values. After generating a complete test packet, Meissa verifies whether the keys matches the hash value, and removes unmatched ones.

Although Meissa does not aim to test stateful behaviors implemented with registers and register actions, it still models registers to test stateless register arithmetic behaviors. Meissa's frontend regards registers as header fields. For example, the register `reg[0]` is modeled as a header field `REG:reg-POS:0`. The register action `hdr.tcp.dst_port = reg[0]` is modeled as an action statement `hdr.tcp.dst_port ← REG:reg-POS:0`. It is worth mentioning that Meissa can only model registers when their indexes are constant. We will discuss our scope detailly in §7.

5 EVALUATION

5.1 Methodology

Data Plane Programs. Table 1 shows eight data plane programs used in our evaluation. mTag [17] and switch.p4 [7] are representative open-source programs. Router and ACL are simplified versions of switch.p4. We remove some features from switch.p4 that are not supported by p4pktgen [61] in order to compare Meissa with it. We also use four production-scale data plane programs deployed in our production edge networks. gw-4 uses all eight pipes of two Intel Barefoot Tofino switches, and it is one of the most complex data plane programs in our production practice.

Table rule sets. We generate random table rule sets for Router, mTag, ACL and switch.p4. We collect four table rule sets (set-1, set-2, set-3 and set-4) from actual switches deployed in our production networks for gw-1, gw-2, gw-3 and gw-4. gw-1, gw-2 and gw-3 use parts of table rule sets because of their small scale, while gw-4 fully uses the entire table rule sets. Specifically, set-2 supports twice the number of elastic IPs than that in set-1, set-3 twice of that in set-2, and set-4 twice of that in set-3. set-4 is more than 200,000 LOC, representing a large table rule set for production programs.

Baselines. We compare Meissa with four baselines, including p4pktgen [61], PTA [18], Gauntlet [68] and Aquila [79]. p4pktgen [61], PTA [18] and Gauntlet [68] are testing solutions, and Aquila [79] is a verification solution. All experiments (based on Meissa and the four baselines) are conducted on a bare metal server with 96 cores and 768 GB memory.

5.2 Experimental Results

Scalability. This experiment evaluates the scalability of Meissa in terms of the time to generate test cases for data plane programs. In this experiment, we use the model-based testing mode of Gauntlet [68], which can generate test cases for a given program. The fuzzing mode of Gauntlet fuzzes small programs to test the P4 compiler, which is not relevant to the goal of testing large-scale data plane programs in this experiment. We modify the model-based testing mode of Gauntlet to traverse all possible table rules to achieve full coverage for fair comparison, instead of *ignoring* rules and actions in its original Python implementation. Because Gauntlet and p4pktgen do not support custom table rules and other production features, we skip their evaluation on the last four production programs shown in Table 1. PTA [18] requires engineers to handwrite test cases. It is not comparable in this experiment that focuses on the capability to automatically generate test cases for full coverage.

Figure 9 shows that Meissa generates test cases for Router, mTag, ACL and switch.p4 in less than 100 seconds, which is 1.6–400× faster than p4pktgen and Gauntlet. For complex production programs gw-1 and gw-2, Meissa is 22.9× and 26.5× faster than Aquila. For gw-3 and gw-4, Aquila fails to verify them with one-hour time budget, while Meissa generates test cases with full coverage within 150 seconds.

In addition to different programs, we also vary the table rule sets of the programs as the table rule sets also affect the complexity of testing and verifying data plane programs. Because Gauntlet and p4pktgen cannot handle custom table rule sets and Aquila runs out of time on gw-3 and gw-4, we use gw-1 and gw-2 in this experiment. Figure 10 shows the running time of Meissa and Aquila on different table rule sets for gw-1 and gw-2. Meissa is 6.7–41.2× faster than Aquila under different table rule sets.

Name	Descriptions on functionality	LOC	# of pipes	# of switches
Router	A simple router based on switch.p4 that only contains layer-3 routing.	256	1	1
mTag [17]	mTag-edge [17] that inserts and removes tags in switches attached to hosts.	227	1	1
ACL	ACL filtering on *dst_addr*, *src_addr* and ECN, based on Router.	400	1	1
switch.p4 [7]	Multifunctional data plane program, including L2 switching, L3 routing, ECMP, tunnel, ACLs, MPLS, *etc.*	7086	1	1
gw-1	Production program for hardware gateway, processing VXLAN.	> 1000	1	1
gw-2	Production program for hardware gateway, processing VXLAN, ACL, routing, *etc.*	> 3000	2	1
gw-3	Production program for hardware gateway, including proprietary protocols and switch.p4.	> 10000	4	1
gw-4	Production program for hardware gateway, using two switches for higher availability and throughput.	> 20000	8	2

Table 1: Data plane programs used in evaluation.

Effectiveness of code summary. We evaluate the effectiveness of code summary under different programs and table rule sets. Figure 11a shows that code summary reduces the running time of Meissa by 1.2–5.0× when varying the data plane programs. Figure 12a shows that code summary reduces the running time of Meissa by 2.2–4.5× on gw-4 with different table rule sets. Note that the majority of the complexity in gw-4 with set-4 is inside the fifth pipeline ppl_5, so both Meissa with code summary and Meissa without code summary spend the majority of their time (~80%) on searching in ppl_5, which reduces the gap between the two compared to other cases. To further understand how code summary improves performance, we show the number of calls to the SMT solver and the number of paths examined by DFS. Figure 11b and Figure 12b show that code summary reduces the number of calls to the SMT solver by 1.8–14.9× for different programs and rule sets. Figure 11c and Figure 12c show that code summary reduces the number of paths in the CFG by 10^{60}–10^{390}× for different programs and rule sets.

Finding real bugs. In addition to the formal proof that Meissa achieves 100% path coverage (§3.4), we select 16 representative programs to demonstrate the bug finding capability of Meissa. Table 2 shows these bugs. These bugs are related to a variety of data plane features such as parser, checksum and SALU. These bugs represent a wide range of issues, from basic bugs in the code logic all the way to the bugs in the backend and compiler configurations. All code bugs have been reported to our developers, and all non-code bugs have been reported to the vendor. All bugs have been confirmed and resolved.

Meissa successfully detects bug 1, 2, 3, 4 and 5 that can be detected by Aquila. Meissa also detects bug 7, 8, 9, 10 and 11 that can be detected by Gauntlet. p4pktgen only tests a small subset of P4 functionalities, and it cannot detect bugs related to P4 compiler. PTA requires handwritten unit tests, and it does not support P4-16 in which bug 7–16 are written. Gauntlet did not detect bug 12–16 before, and it cannot scale to complex production data plane programs. Aquila verifies program logics, so it cannot detect compilation misconfigurations and compiler bugs.

Bugs shown in Table 2 are just a few representative examples in our bug finding evaluation. Besides them, we have used Meissa to

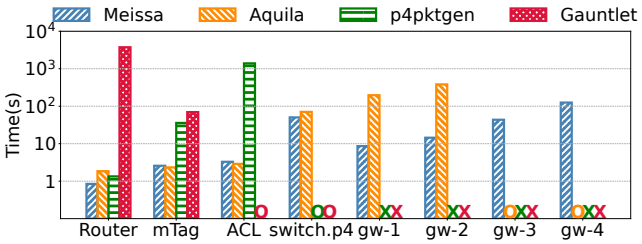


Figure 9: Running time on different data plane programs. ○ indicates timeout; × indicates no-support.

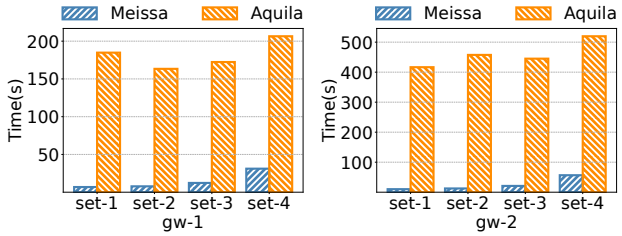


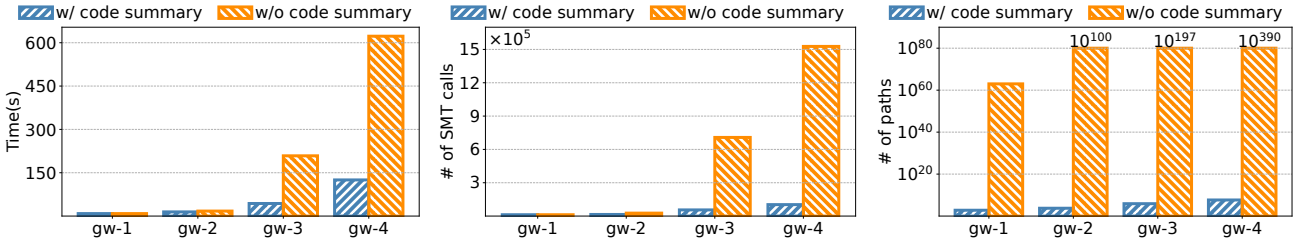
Figure 10: Running time on gw-1 and gw-2 under different table rule sets.

test both open-source and production-scale data plane programs that have been evaluated by Aquila in *Tian et al.* [79]. Meissa has successfully identified all code bugs found by Aquila. Similarly, we have reproduced all compiler bugs found by Gauntlet [68] and these compiler bugs can also be detected by Meissa.

More importantly, Meissa is able to find bug 2, 6, 12, 13, 14, 15, and 16 that were previously undisclosed. We have deployed Meissa in production and it has detected bug 2, 6, 14, 15, and 16 in our production data plane programs. We discuss bug 6, 14 and 15 in detail in §6.

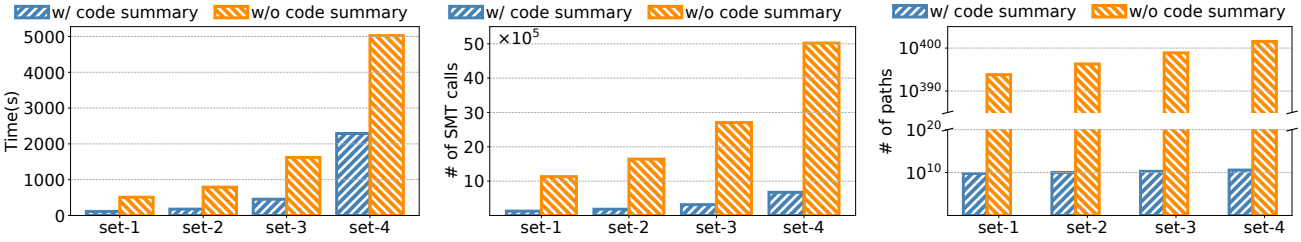
6 DEPLOYMENT EXPERIENCE

Deployment of Meissa. Meissa has been used by our network engineers to test data plane programs in production edge networks for three months. Before Meissa was developed, network engineers maintained a set of test cases, each describing a usage scenario



(a) Running time. (b) Number of SMT calls. (c) Number of paths.

Figure 11: Effectiveness of code summary on different data plane programs.



(a) Running time. (b) Number of SMT calls. (c) Number of paths.

Figure 12: Effectiveness of code summary on different table rule sets.

Type	Index	Bug	Meissa	p4pktgen	PTA	Gauntlet	Aquila
Code Bugs	1	Routing misconfiguration	✓	✗	✗	✗	✓
	2	Unrestricted ACL rules	✓	✗	✗	✗	✓
	3	Parser wrong logic	✓	✓	✓	✓	✓
	4	Ingress wrong logic	✓	✓	✓	✓	✓
	5	Wrong deparser emit	✓	✗	✓	✗	✓
	6	Checksum fail-to-update	✓	✗	✗	✗	✗
Non-code Bugs	7	p4c frontend bug 2147 [4]	✓	✓	✗	✓	✗
	8	p4c frontend bug 2343 [5]	✓	✓	✗	✓	✗
	9	bf-p4c backend bug 1 [1]	✓	✗	✗	✓	✗
	10	bf-p4c backend bug 3 [2]	✓	✗	✗	✓	✗
	11	bf-p4c backend bug 6 [3]	✓	✗	✗	✓	✗
	12	bf-p4c backend bug A (incorrect arithmetic comparison)	✓	✗	✗	✗	✗
	13	bf-p4c backend bug B (incorrect assignment)	✓	✗	✗	✗	✗
	14	bf-p4c backend bug C (setValid)	✓	✗	✗	✗	✗
	15	Misuse of optimization pragmas	✓	✗	✗	✗	✗
	16	Missing compilation flags	✓	✗	✗	✗	✗

Table 2: Comparison of the capability to find bugs of different solutions. All code bugs have been reported to our developers, and all non-code bugs have been reported to the vendor. All bugs have been confirmed and resolved.

corroborated by requirement analysis engineers. A test case was executed by first calling the control plane interface and then invoking tools (*e.g.*, iperf3) to test corresponding data plane behaviors.

After Meissa was deployed, instead of relying on arbitrary tools such as iperf3, network engineers break down the data plane behaviors and instruct Meissa to test each of them. For example, a NAT gateway processes packets going both ways (in and out), supports three protocols (TCP, UDP, and ICMP), and thus results in six sub-cases. For each sub-case, Meissa provides a set of base constraints on the input packet, *e.g.*, a valid IPv4 packet must satisfy ethernet.type == 0x800. Then, network engineers specify test-case-specific constraints, *e.g.*, ipv4.dst_addr must equal the NAT public address. Specifying the end-to-end behavior is also straightforward:

```
action nat_encap_ip(...){
    // add VXLAN, inner_ip
    hdr.innerIpv4.srcAddr = hdr.ipv4.srcAddr;
    ......
}

action nat_encap_tcp(...){
    // set inner TCP packet
    hdr.innerTcp.ackno = hdr.tcp.ackno;
    ......
}

control{
    ......
    // encap ingress TCP packets
    nat_encap_ip();
    nat_encap_tcp();
    ......
}
```

Figure 13: Code snippet related to issue #15 in Table 2.

the received packet should contain the same headers as the input, except that certain IP address and port number are updated according to the NAT rule. In this way, it is easy for network engineers without a formal method background to attach Meissa to existing test cases. They benefit from the full-path-coverage testing and indeed discover previously-hidden bugs. We detail three real, representative bugs detected by Meissa in the production deployment.

Issue #6 in Table 2: Checksum fail-to-update. Our elastic IP product offloads ingress packet tunneling to a Tofino switch at gateways. In our network, a `<IPv4 TCP>` packet is encapsulated to a `<IPv4-out UDP-out VXLAN-out IPv4-in TCP-in>` packet, and then sent to virtual machines. In this case, the inner header's IP and port are modified, so its layer-4 checksum needs to be updated. In order to fit such complex P4 program into the actual hardware, we split the inner checksum update into two pipelines in practice. The ingress calculates the layer-4 checksum. The egress pipeline identifies packet type by header validity, and puts the layer-4 checksum into the corresponding position. However, our engineers forgot to parse inner TCP in the egress pipeline, so inner TCP would never be valid and its checksum would never be updated.

Verification tools such as p4v [56] could not detect this bug, because verifying checksum is not well supported by SMT solvers. Meissa generated test cases that covered the tunnel logic, and reported *No Pass* due to inner TCP checksum error. Moreover, Meissa's test report also indicated that inner TCP header is invalid in all paths.

Issue #14 in Table 2: bf-p4c backend bug C (setValid). One of our applications of programmable data planes is to implement a custom traffic generator. It takes as input a seed packet, adds some variations, and outputs the altered packet. This program relies on invoking the standard P4 function *setValid* to alter the output packet structure as desired. However, when the compiled program executes certain program path, the invocation of setValid does not take effect and the corresponding headers remain *invalid*. As a result, our traffic generator does not work with these kinds of seed packets.

Verification tools such as Aquila [79] and p4v [56] cannot detect this bug, because the code logic is correct. This bug was not detected by the state-of-the-art compiler tester (*e.g.*, Gauntlet [68]) because its model-based testing does not scale to large programs. Our existing test suites did not find this bug, because this program was under early development and lacked comprehensive test cases at that time. As a comparison, Meissa generates test cases for the buggy path and detects inconsistency between the expected output and the actual output. In addition, Meissa concludes that this bug is not in the P4 program's code logics. Our developers then worked with the vendor to confirm that this was indeed caused by the compiler and finally resolved this issue. The vendor accepted this bug and fixed it in the latest release of P4 compiler. It is worth mentioning that this bug did not appear in previous or later versions of our program. Our experiences show that many compiler bugs are only triggered by specific versions of programs, and they are hard to reproduce in other programs. Thus, it is important to generate extensive test cases to fully cover the paths of a program.

Issue #15 in Table 2: Misuse of optimization pragmas. The ingress pipeline of our elastic IP program encapsulates incoming packets into VXLAN tunnels. Figure 13 shows its logic for processing TCP packets. However, the compiled program does not behave as specified by its P4 code logics. The reason is that, our developers employed optimization pragmas to guide the program's compilation, which in turn disabled safety checks. As a result, `hdr.tcp.ackno` overlapped with `hdr.innerTcp.srcAddr`, while they should hold independent values simultaneously. Thus, `nat_encap_ip` incorrectly modified ACKNO of TCP header, and `nat_encap_tcp` propagated this buggy update to inner TCP header.

Verification tools such as Aquila [79] could not detect this bug, because the code logic is correct. This bug was not detected by our existing test cases because it only happened to packets with specific types, and existing tests do not check ACK number. The state-of-the-art P4 compiler tester (*e.g.*, Gauntlet [68]) did not detect this bug before, because its fuzz testing does not provide full coverage and its model-based testing does not scale to programs that are large enough to trigger this bug. Meissa generated test cases for the buggy path, detected an inconsistency between the expected output and the actual output, and implied that this bug was outside of P4 specifications, and the ACK number of the inner TCP packet header was incorrect. With help from Meissa, our developers finally found that this bug was caused by misuse of pragmas. They carefully adjusted pragmas in the program and fixed this bug.

7 DISCUSSION

This section clarifies Meissa's scope and limitations.

Performance bugs. One limitation of Meissa is that it does not support detecting performance bugs, *i.e.*, it cannot identify packet patterns that incur long latency or low overall bandwidth. This is because the source code of a data plane program only defines its functional behavior, but not execution time. An execution time analysis relies on the generated binary code and a comprehensive model of the underlying hardware, which is not publicly available. There are works that synthesize network performance with verification [22, 23, 39], which are orthogonal to Meissa.

Bug localization. Once a bug is reported, *i.e.*, a pair of buggy concrete input and output packets (or its absence) is found, Meissa symbolically executes this concrete input and generates a trace that

shows all executed actions, hit table rules, branching, and assignment statements, along with the values of corresponding arguments at each statement. It also identifies code bugs by comparing the symbolic output packet with user-defined specifications. Then engineers manually review this trace to identify the root cause of this bug. Usually, for code bugs, such a trace is sufficient for its localization. However, for compiler bugs and compilation misconfigurations, engineers need to inject the same input and collect a physical trace from the programmable device to check where the physical execution diverges from its source code.

Testing stateful programs. Meissa does not test the interactive behaviors of stateful programs. In particular, its high-level intent only allows specifying single-packet processing behaviors, and it treats stateful registers as unbounded stateless variables. The reason is twofold. (*i*) Our production programmable data planes mostly contain stateless processing logic. (*ii*) Interactive behaviors usually allow infinite packet sequences, which is orthogonal to the full path coverage this paper focuses on. Recent work p4wn [43] uses probabilistic methods to test stateful processing behaviors (without guaranteeing full coverage), which is complementary to Meissa.

Testing control plane behaviors. Meissa does not test control plane behaviors such as BGP peering and MAC learning. Meissa is used to test data planes and static table rules. Moreover, testing control plane behaviors is hard. It is very tricky to model these behaviors and control plane logics are much more complex than data plane logics. We leave it a future work to model interactions between control plane modules and generate test cases.

Checking the program, not the compiler/ASIC. The goal of Meissa is to check the correctness of data plane *programs* for program developers and network operators. It detects code bugs in programs, and non-code bugs triggered by those programs; it does not actively search for compile bugs that are not triggered. Compiler bugs that are not manifested by a program are not concerns for developing and deploying this particular program. Comprehensively testing compilers is a non-goal for Meissa. Recent work Gauntlet [68] focuses on testing compilers for compiler developers, which is orthogonal to Meissa.

Potential path explosion. Although evaluation results prove Meissa scales to large and complex programs, path explosion is still theoretically possible. For example, a program with few public pre-conditions weakens inter-pipeline public pre-condition filtering. In practice, we only test one type of packets at a time to avoid path explosion. Besides, when too few public pre-conditions weaken code summary, we group pre-conditions according to packet type, conduct summary seperately and merge them into a full summary.

8 RELATED WORK

Programmable data plane testing. Many efforts have been proposed to test programmable data planes, but none of them managed to test programs with production scales and functionalities. p4pktgen [61] uses symbolic execution to generate test cases for P4 programs. Path explosion makes it impracticable to test large-scaled programs. It also does not test table rules and other production functionalities. PTA [18] translates P4 programs (with assumptions and assertions) into packet sender and checker programs. It requires engineers to handwrite unit tests in programs, while providing unit tests for complex data plane programs is probably costly and incomprehensive. Gauntlet [68] focuses on testing compilers with small input programs. Its model-based testing mode is too rudimentary to test production-scale programs. Compared to them, Meissa supports commonly used functionalities, scales to large programs and achieves full path coverage. p4wn [43] profiles stateful programs with a probabilistic method to perform adversarial testing with no guarantee of full coverage, which is complementary to Meissa (§7). Yardstick [86] defines and computes network-wide coverage metrics, while Meissa focuses on testing individual data plane programs.

Network verification. Many verification techniques have been proposed for data plane programs [27, 28, 31, 56, 60, 76, 79]. Aquila [79] focuses on scalability and specification complexity for production-level data plane programs. p4v [56] formally check P4 program correctness with a classic verification approach. Vera [76] leverages symbolic execution for scalable, exhaustive verification of P4 program snapshots. bf4 [27] searches possible bugs, and avoids them in runtime by adding table rules. Besides, there is a lot of work on verifying network protocols [9, 11, 29, 30, 69, 73, 77, 85] and configurations [13, 14, 16, 21, 32, 34, 37, 38, 41, 42, 44, 45, 49, 58, 63, 64, 66, 71, 74, 78, 80, 82, 88–90, 93, 94, 96–98].

Applications of programmable data planes. Many network applications are implemented on programmable data planes. Some of them improve network performance or fault tolerance [48, 55, 91, 92]. Many efforts are proposed to offload functionalities to the data plane for better performance [40, 47, 52, 54, 57, 59, 62, 70, 95, 102]. Network telemetry leverages the flexibility of programmable data planes to monitor network status [15, 25, 35, 36, 99–101]. Meissa helps developers test data planes to ensure the correctness of these applications.

9 CONCLUSION

We present Meissa, a scalable network testing system for programmable data planes. Meissa leverages a domain-specific *code summary* technique to guarantee full path coverage. Meissa is able to test large data plane programs that cannot be supported by state-of-the-art solutions, and identify both existing and new bugs. We have deployed Meissa in production, which serves as a vital tool in the development and operation of our data plane products.

This work does not raise any ethical issues.

Acknowledgments. This work was supported by the National Key Research and Development Program of China under the grant number 2020YFB2104100 and the National Natural Science Foundation of China under the grant number 62172008. We thank the anonymous reviewers for their valuable feedback on this paper. We thank Vladimir Gurevich, Jeongkeun Lee, and Yiqun Cai for confirming the bugs and providing comments on this the paper. Xin Jin (xinjinpku@pku.edu.cn) is the corresponding author. Naiqian Zheng, Xuanzhe Liu, and Xin Jin are also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

REFERENCES

[1] 2020. bf-p4c backend bug 1 detected by Gauntlet. https://github.com/p4gauntlet/gauntlet/blob/master/bugs/tofino/semantic/semantic_bug1.p4.
[2] 2020. bf-p4c backend bug 3 detected by Gauntlet. https://github.com/p4gauntlet/gauntlet/blob/master/bugs/tofino/semantic/semantic_bug3.p4.
[3] 2020. bf-p4c backend bug 6 detected by Gauntlet. https://github.com/p4gauntlet/gauntlet/blob/master/bugs/tofino/semantic/semantic_bug6.p4.
[4] 2020. P4c issue 2147. https://github.com/p4lang/p4c/issues/2147.
[5] 2020. P4c issue 2343. https://github.com/p4lang/p4c/issues/2343.
[6] 2021. Data Plane Development Kit. https://www.dpdk.org.
[7] 2021. Open Tofino. https://github.com/barefootnetworks/Open-Tofino.
[8] 2021. PTF: Packet testing framework. https://github.com/p4lang/ptf.
[9] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast multilayer network verification. In *USENIX NSDI*.
[10] Paul Ammann and Jeff Offutt. 2016. *Introduction to software testing*. Cambridge University Press.
[11] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward formally verifying congestion control behavior. In *ACM SIGCOMM*.
[12] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. Cvc4. In *International Conference on Computer Aided Verification*.
[13] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *ACM SIGCOMM*.
[14] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *ACM SIGCOMM*.
[15] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. Pint: Probabilistic in-band network telemetry. In *ACM SIGCOMM*.
[16] Rudiger Birkner, Tobias Brodmann, Petar Tsankov, Laurent Vanbever, and Martin T Vechev. 2021. Metha: Network Verifiers Need To Be Correct Too!. In *USENIX NSDI*.
[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* (2014).
[18] Pietro Bressana, Noa Zilberman, and Robert Soule. 2020. Finding hard-to-find data plane bugs with a PTA. In *ACM CoNEXT*.
[19] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX OSDI*.
[20] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* (2013).
[21] Eric Hayden Campbell, William T Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soule, and Nate Foster. 2021. Avenir: Managing Data Plane Diversity with Control Plane Synthesis. In *USENIX NSDI*.
[22] Yiyang Chang, Chuan Jiang, Ashish Chandra, Sanjay Rao, and Mohit Tawarmalani. 2019. Lancet: Better network resilience by designing for pruned failure sets. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2019).
[23] Yiyang Chang, Sanjay Rao, and Mohit Tawarmalani. 2017. Robust validation of network designs under uncertain demands and failures. In *USENIX NSDI*.
[24] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *International Conference on Software Engineering*.
[25] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. 2020. Beaucoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*.
[26] Leonardo De Moura and Nikolaj Bjorner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.
[27] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. 2020. bf4: towards bug-free P4 programs. In *ACM SIGCOMM*.
[28] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2019. Dataplane equivalence and its applications. In *USENIX NSDI*.
[29] Tomer Eliyahu, Yafim Kazak, Guy Katz, and Michael Schapira. 2021. Verifying learning-augmented systems. In *ACM SIGCOMM*.
[30] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. 2021. Prognosis: closed-box analysis of network protocol implementations. In *ACM SIGCOMM*.
[31] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Uncovering bugs in p4 programs with assertion-based verification. In *Symposium on SDN Research*.
[32] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*.
[33] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. *SIGPLAN Not.* (2007).
[34] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time network verification using atoms. In *USENIX NSDI*.
[35] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. 2021. Toward nearly-zero-error sketching via compressive sensing. In *USENIX NSDI*.
[36] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. 2020. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *ACM SIGCOMM*.
[37] Karthick Jayaraman, Nikolaj Bjorner, Geoff Outhred, and Charlie Kaufman. 2014. Automated analysis and debugging of network connectivity policies. *Microsoft Research* (2014).
[38] Karthick Jayaraman, Nikolaj Bjorner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. 2019. Validating datacenters at scale. In *ACM SIGCOMM*.
[39] Chuan Jiang, Sanjay Rao, and Mohit Tawarmalani. 2020. PCF: provably resilient flexible routing. In *ACM SIGCOMM*.
[40] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soule, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*.
[41] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. 2020. GRooT: Proactive Verification of DNS Configurations. In *ACM SIGCOMM*.
[42] Siva Kesava Reddy Kakarla, Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd Millstein, Yuval Tamir, and George Varghese. 2020. Finding network misconfigurations by automatic template inference. In *USENIX NSDI*.
[43] Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. 2021. Probabilistic profiling of stateful data planes for adversarial testing. In *ACM ASPLOS*.
[44] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header space analysis: Static checking for networks. In *USENIX NSDI*.
[45] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. Veriflow: Verifying network-wide invariants in real time. In *USENIX NSDI*.
[46] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*.
[47] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*.
[48] Daehyeok Kim, Jacob Nelson, Dan RK Ports, Vyas Sekar, and Srinivasan Seshan. 2021. Redplane: Enabling fault-tolerant stateful in-switch applications. In *ACM SIGCOMM*.
[49] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable dynamic network control. In *USENIX NSDI*.
[50] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* (1976).
[51] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *ACM SIGPLAN Notices* (2012).
[52] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *USENIX NSDI*. 741–761.
[53] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* (2015).
[54] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan RK Ports. 2020. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *USENIX OSDI*.
[55] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. 2019. HPCC: High precision congestion control. In *ACM SIGCOMM*.
[56] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soule, Han Wang, Cualin Cacscaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical verification for programmable data planes. In *ACM SIGCOMM*.
[57] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *USENIX Security Symposium*.
[58] Nuno P Lopes, Nikolaj Bjorner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking beliefs in dynamic networks. In *USENIX NSDI*.
[59] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching

asics. In *ACM SIGCOMM*.
[60] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Verification of p4 programs in feasible time using assertions. In *ACM CoNEXT*.
[61] Andres Notzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. P4pktgen: Automated test case generation for p4 programs. In *Symposium on SDN Research*.
[62] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. 2021. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *ACM SIGCOMM*.
[63] Aurojit Panda, Katerina Argyraki, Mooly Sagiv, Michael Schapira, and Scott Shenker. 2015. New directions for network verification. In *Summit on Advances in Programming Languages*.
[64] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying reachability in networks with mutable datapaths. In *USENIX NSDI*.
[65] Van-Thuan Pham, Marcel Bohme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *International Conference on Software Testing, Validation and Verification*.
[66] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*.
[67] Jose Miguel Rojas and Corina S Pasareanu. 2013. Compositional symbolic execution through program specialization. *BYTECODE'13 (ETAPS)* (2013).
[68] Fabian Ruffy, Tao Wang, and Anirudh Sivaraman. 2020. Gauntlet: Finding bugs in compilers for programmable packet processing. In *USENIX OSDI*.
[69] Leonid Ryzhyk, Nikolaj Bjorner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B Terry, and George Varghese. 2017. Correct by construction networks using stepwise refinement. In *USENIX NSDI*.
[70] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtarik. 2019. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701* (2019).
[71] Tibor Schneider, Rudiger Birkner, and Laurent Vanbever. 2021. Snowcap: synthesizing network-wide configuration updates. In *ACM SIGCOMM*.
[72] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* (2005).
[73] Jinghao Shi, Shuvendu K Lahiri, Ranveer Chandra, and Geoffrey Challen. 2018. Wireless protocol validation under uncertainty. *Formal methods in system design* (2018).
[74] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. Probabilistic verification of network configurations. In *ACM SIGCOMM*.
[75] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *ISOC Network and Distributed System Security Symposium*.
[76] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 programs with Vera. In *ACM SIGCOMM*.
[77] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM*.
[78] Alan Tang, Siva Kesava Reddy Kakarla, Ryan Beckett, Ennan Zhai, Matt Brown, Todd Millstein, Yuval Tamir, and George Varghese. 2021. Campion: debugging router configuration differences. In *ACM SIGCOMM*.
[79] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. 2021. Aquila: a practically usable verification system for production-scale programmable data planes. In *ACM SIGCOMM*.
[80] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. 2019. Safely and automatically updating in-network acl configurations with intent language. In *ACM SIGCOMM*.
[81] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–white box test generation for. net. In *International Conference on Tests and Proofs*.
[82] Yaron Velner, Kalev Alpernas, Aurojit Panda, Alexander Rabinovich, Mooly Sagiv, Scott Shenker, and Sharon Shoham. 2016. Some complexity results for stateful network verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.
[83] Xusheng Xiao, Sihan Li, Tao Xie, and Nikolai Tillmann. 2013. Characteristic studies of loop problems for structural test generation via symbolic execution. In *International Conference on Automated Software Engineering*.
[84] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *IEEE/IFIP International Conference on Dependable Systems and Networks*.
[85] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing Safe and Efficient Kernel Extensions for Packet Processing. In *ACM SIGCOMM*.
[86] Xieyang Xu, Ryan Beckett, Karthick Jayaraman, Ratul Mahajan, and David Walker. 2021. Test coverage metrics for the network. In *ACM SIGCOMM*.
[87] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*.
[88] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. 2020. Aragog: Scalable Runtime Verification of Shardable Networked Systems. In *USENIX OSDI*.
[89] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, Duncheng She, Qing Ma, Biao Cheng, Hui Xu, Ming Zhang, Zhiliang Wang, and Rodrigo Fonseca. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *ACM SIGCOMM*.
[90] Farnaz Yousefi, Anubhavnidhi Abhashkumar, Kausik Subramanian, Kartik Hans, Soudeh Ghorbani, and Aditya Akella. 2020. Liveness verification of stateful network functions. In *USENIX NSDI*.
[91] Liangcheng Yu, John Sonchack, and Vincent Liu. 2020. Mantis: Reactive programmable switches. In *ACM SIGCOMM*.
[92] Zhuolong Yu, Chuheng Hu, Jingfeng Wu, Xiao Sun, Vladimir Braverman, Mosharaf Chowdhury, Zhenhua Liu, and Xin Jin. 2021. Programmable packet scheduling with a single queue. In *ACM SIGCOMM*.
[93] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. 2020. NetSMC: A custom symbolic model checker for stateful network verification. In *USENIX NSDI*.
[94] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *ACM SOSP*.
[95] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. 2022. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In *USENIX NSDI*.
[96] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. 2014. Libra: Divide and conquer to verify forwarding tables in huge networks. In *USENIX NSDI*.
[97] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. 2020. Check before you change: Preventing correlated failures in service updates. In *USENIX NSDI*.
[98] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. 2020. Automated verification of customizable middlebox properties with gravel. In *USENIX NSDI*.
[99] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. 2021. CocoSketch: high-performance sketch-based measurement over arbitrary partial key query. In *ACM SIGCOMM*.
[100] Yikai Zhao, Kaicheng Yang, Zirui Liu, Tong Yang, Li Chen, Shiyi Liu, Naiqian Zheng, Ruixin Wang, Hanbo Wu, Yi Wang, and Nicholas Zhang. 2021. LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets. In *USENIX NSDI*.
[101] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. 2020. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*.
[102] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. Racksched: A microsecond-scale scheduler for rack-scale computers. In *USENIX OSDI*.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A TIME COMPLEXITY ANALYSIS OF CODE SUMMARY

Without loss of generality, we assume a data plane program consists of k sequentially concatenated pipelines, and each pipeline contains m valid paths out of n possible paths.

The time complexity of the basic test case generation framework (§3.2) is straightforward. It is proportional to the total number of possible paths in the entire CFG, that is, $O(n^k)$.

On the other hand, the code summary technique (§3.3) does not need to traverse all possible paths in the original CFG. It iterates

over each pipeline, computes public pre-conditions for it, performs symbolic execution within this pipeline, and computes its summary. During each iteration, the computation of public pre-conditions for a pipeline traverses $O(k)$ summarized pipelines with $O(m)$ paths each, resulting in a time complexity of $O(m^k)$. Then, symbolic execution within a pipeline has a time complexity of $O(n)$, while computing its summary costs $O(m)$ time. In total, iterating over all pipelines has a time complexity of $O(k \cdot m^k + k \cdot n + k \cdot m)$. The summarized CFG contains k pipelines with $O(m)$ paths each. Generating test case templates on the summarized CFG has a time complexity of $O(m^k)$. Therefore, the total time complexity is $O(k \cdot m^k + k \cdot n + k \cdot m + m^k)$. In practice, we have $k \ll m \ll m^k \ll n$. Thus, the total time complexity for the code summary technique is $O(k \cdot n)$.

B THE BASIC FRAMEWORK ACHIEVES FULL PATH COVERAGE

This section formally proves that the basic symbolic execution framework (Algorithm 1) achieves 100% path coverage (Definition 3).

Theorem 4 (The basic algorithm achieves full path coverage)**.** Given a CFG G, for any of its valid path π, it must be discovered by Algorithm 1, and the reported path condition C satisfies the following

$$\forall s, C(s) \implies (\exists s', \langle \pi; s \rangle \rightarrow s')$$

Proof. By Definition 2, any valid path π must also be a possible path, which is discovered by DFS. Thus, the proof obligation is to show that Algorithm 1 indeed symbolically executes along π (*i.e.*, it does not early terminate) and accumulates the correct path constraint.

Assume that path π is of length l_π. We generalize the above theorem to discuss the partial evaluation along its prefix of length l.

Assume that at length l, the current symbolic state V_l, path condition C_l and the partial trace $\pi[0:l]$ satisfies the following

$$\forall s, C_l(s) \implies \langle \pi[0:l]; s \rangle \rightarrow [\![V_l]\!]s$$

At length $l+1$, the current vertex corresponds to either an action statement or a predicate statement.

(1) The current vertex is of $id_1 := aexp_1$. In this case, $C_{l+1} = C_l$, while V_{l+1} equals $V_l[id_1 \leftarrow [\![V_l]\!]aexp_1]$. By the assumption, any initial state s satisfying $C_{l+1}(s)$ must execute along $\pi[0:l]$, and further $\pi[0:l+1]$, since the current vertex is not a predicate. In addition, the resulting symbolic state also equals $[\![V_{l+1}]\!]s$.
(2) The current vertex is of *assume* $bexp_1$. In this case, $V_{l+1} = V_l$, while $C_{l+1} = C_l \;\&\&\; [\![V_l]\!]bexp_1$. By the assumption, any initial state s satisfying $C_l(s)$ must execute along $\pi[0:l]$ and result in $[\![V_l]\!]s$. Further, since $([\![V_l]\!]bexp_1)s$ must hold, the current predicate $bexp_1([\![V_l]\!]s)$ must evaluate to True, and the initial state s executes along $\pi[0:l+1]$.

Thus, the induction hypothesis holds.

Applying this hypothesis with $l := l_\pi$ proves the theorem. □