Aquila: A Practically Usable Verification System for Production-Scale Programmable Data Planes

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[★], Minlan Yu[§]

[†]*Alibaba Group* [§]*Harvard University* [★]*Nanjing University*

ABSTRACT

This paper presents Aquila, the first practically usable verification system for Alibaba's production-scale programmable data planes. Aquila addresses four challenges in building a practically usable verification: (1) specification complexity; (2) verification scalability; (3) bug localization; and (4) verifier self validation. Specifically, first, Aquila proposes a high-level language that facilitates easy expression of specifications, reducing lines of specification codes by tenfold compared to the state-of-the-art. Second, Aquila constructs a sequential encoding algorithm to circumvent the exponential growth of states associated with the upscaling of data plane programs to production level. Third, Aquila adopts an automatic and accurate bug localization approach that can narrow down suspects based on reported violations and pinpoint the culprit by simulating a fix for each suspect. Fourth and finally, Aquila can perform self validation based on refinement proof, which involves the construction of an alternative representation and subsequent equivalence checking. To this date, Aquila has been used in the verification of our production-scale programmable edge networks for over half a year, and it has successfully prevented many potential failures resulting from data plane bugs.

CCS CONCEPTS

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

KEYWORDS

Formal Methods; Programmable Switches; P4 Verification

ACM Reference Format:
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 2021 Conference (SIGCOMM '21), August 23–27, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3452296.3472937

[△]Both authors contributed equally to the paper.

1 INTRODUCTION

As a prominent online service provider, Alibaba operates a global network infrastructure serving over one billion customers, offering diverse services including cloud, e-commerce, and video. In order to provide end users with faster services (*e.g.*, IoT and CDN) at reasonable costs, Alibaba has built and is operating a large number of edge networks, each consisting of hundreds of light-weight servers with tight space constraints and CPU compute limitations. As services today constantly evolve in size, it is becoming increasingly difficult for these resource-constrained edge networks to handle the ever-growing traffic, a situation that could severely undermine the performance of business services. This prompted us to look into solutions to offload a group of network functions (*e.g.*, load balancing, firewall, and NAT) from the server, with a goal to conserve CPU resources for better performance.

Recent advances in programmable switching ASICs have equipped us with the ability to implement network functions in the data plane using P4 language, opening up new cost-effective solutions to significantly improve the performance of our edge networks, in terms of both functionality and efficiency as well as flexibility. First, we can offload a group of network functions from the edge servers to the switch data plane to significantly save on the constrained server CPU resources. Second, we can implement the network functions on programmable ASICs at Tbps speeds for packets. Finally, we can introduce additional flexibility in network function updates as the business evolves, by directly programming the switch data plane. Given these advantages, we have widely adopted programmable switches in our edge networks.

Nevertheless, the deployment of programmable switches also inevitably introduces new challenges. Given the evolving scale and diversity of our services, the data plane programs in our edge networks are greatly increasing in complexity. The data plane program of each switch consists of thousands lines of P4 code in multiple pipelines, each pipeline holding a number of network functions. Network functions across different pipelines are tangled with various packet paths, resulting in complex function chain logical relations within a single programmable device. Therefore, it has become a great challenge for our network engineers to ensure the correctness of data plane programs of such complexity.

Among methods of checking correctness, testing (*e.g.*, p4pktgen [36]) is the most straightforward. However, it is too expensive and complicated to test programmable devices with diverse packet formats and operations [30, 48]. A number of previous efforts have focused on the rigorous verification of programmable data planes, such as p4v [30], Vera [48], bf4 [11], p4-assert [35], and p4-NOD [32]. While these state-of-the-art efforts work well in principle, in reality in our situation, they fail to address a number of technical obstacles and limitations that affect the practical usage experience of our engineers. Under the existing methods, our engineers have encountered drawbacks in both specification expression and verification efficiency, as well as violation localization and verifier self-validation.

Alibaba therefore decided to build a verification system that not only provides a *rigorous* guarantee on the correctness of our production-scale programmable data planes, but also meets the practical usage requirements of our network engineers. A *practically usable* verification system that satisfies our purpose should holistically and simultaneously achieve the following features: first, it should enable our network engineers to express their correctness specifications with ease; second, it should be able to efficiently verify the production-scale data plane programs within few minutes; third, upon detection of a violation, it should be able to automatically and accurately localize the bugs in the data planes and the table entries; and fourth, the system should have the ability to self-validate its own implementation correctness, thus delivering confidence in its verification results.

This paper shares our real-life experience with building a **practically usable** verification system, Aquila, as shown in Figure 1, for Alibaba's production-scale programmable data planes. To satisfy the aforementioned feature requirements, we have specifically addressed the following challenges.

Challenge 1: Specification complexity. Properties in our production context are complex and related to specific service purposes, such as "for each packet with headers 'eth, optional vlan, ipv4 (or ipv6), tcp', the tcp header remains unchanged after passing through the switch" as shown in Figure 3. This requires encapsulations of commonly-used property assertions and comprehensive property supports, enabling our engineers to describe their intent with ease. However, prior work does not meet the above requirements. First, the state of the art employed low-level specification languages, *e.g.*, first-order logic, and both Vera and p4v, for example, need 20+ lines of specifications to express just a single (above-mentioned) property. While a recent work bf4 [11] has focused on automatically inferring annotations for undefined behaviors, *e.g.*, invalid header and out-of-register checking, it is unable to infer properties related to specific services (called *service-specific properties*), such as "the DSCP value of each packet destinating 10/8 should be changed to three". In our network, service-specific properties account for 90% of our specifications. Second, existing tools fail to support important properties (*e.g.*, multi-pipeline control, recirculation, deparsing, and checksum), which are widely used in our production.

We, therefore, propose a new specification language, LPI, that encapsulates commonly-used properties with declarative grammar and supports comprehensive properties for parser, MAUs (match-action units) and switch architecture, such as multi-pipeline, header parsing/deparsing, and recirculation. LPI reduces lines of specifications by tenfold compared to previous low-level languages.

Challenge 2: Verification scalability. Our production-scale P4 programs typically contain (1) many network functions, each with thousands lines of P4 code, across multiple pipelines, and (2) complex parser programs with thousands of dependencies across states. Such a complexity results in the exponential growth of states and program branches to be verified, outgrowing a solver's compute capability. For example, when our engineers used p4v and Vera to check an INT-enabled `switch.p4`, which is just a part of our CDN P4 program (in Figure 2), p4v crashed due to the state explosion of encoded formula, and Vera triggered a timeout, let alone checking the entire program. To address the scalability challenge, we propose a novel sequential encoding algorithm to circumvent the exponential growth of states associated with the upscaling of data plane programs to production level. Our experiments show our approach verifies our production-scale programs in an efficient way (see Table 3 and Figure 11).

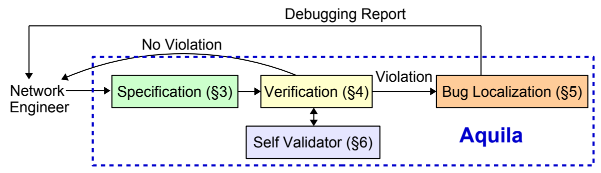


Figure 1: Aquila's practically usable workflow.

Challenge 3: Bug localization. While verification can tell specifications are violated, it is non-trivial to automatically, accurately find the root cause in the production P4 program, because a violation might be caused by diverse root causes, especially in P4 context, such as incorrect table entries and table action missing. Aquila proposes a novel algorithm that narrows down suspect variables and actions based on reported violations and pinpoints the culprit by simulating a fix for each suspect. Our experiments show that this approach can accurately find out the root causes in real-world buggy P4 programs, saving a lot of debugging time.

Challenge 4: Verifier self validation. Verifer implementation errors are headache in practice, since they badly affect the accuracy of verification results. We did incur bugs during Aquila development. To identify Aquila implementation errors, we build a self validator based on refinement proof [33, 39], which checks the GCL (guarded command language) semantic equivalence between Aquila and alternative. We successfully identified errors in the Aquila development such as incorrect encoding and language misunderstanding.

Real-world evaluation. Aquila has been used to verify our programmable edge networks for half a year, and successfully detected many bugs, preventing service downtime. After Aquila is used, no failures resulting from data plane bugs occurred so far. §7 shares real cases we met in practice. §8 compares Aquila with the state-of-the-art verification tools, *e.g.*, p4v and Vera, by verifying open-source and production P4 programs. Aquila outperforms these tools in verification scalability and specification expressing.

2 OVERVIEW

Alibaba has a global network infrastructure to support its worldwide online services, including cloud, e-commerce, and video, which have more than one billion users. By Jan. 2021, we have built many edge networks—consisting of O(100) PoP (point of presence) nodes and O(1000) edge sites in total—to ensure to offer end users fast, high-quality services. To offer high throughput (Tbps speeds) and save CPU resources, our edge networks have widely deployed programmable switches to offload a group of network functions (*e.g.*, load balancing, firewall and DDoS defense) from software to programmable switching ASIC hardware.

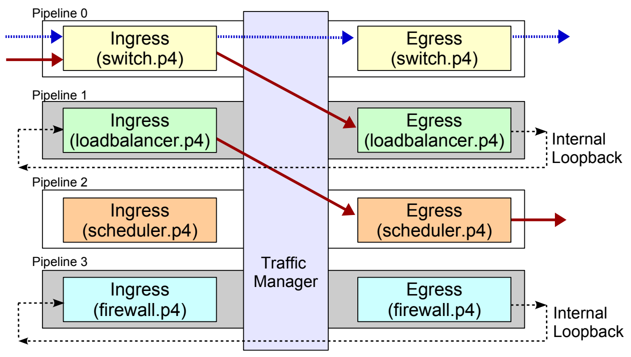
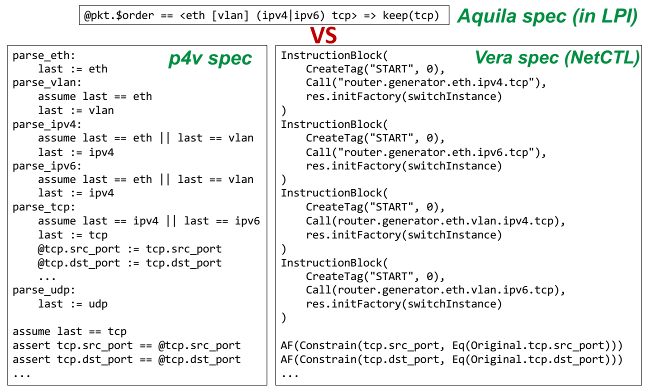


Figure 2: An example hyper-converged data plane within a single edge switch for Alibaba's CDN service.


Aquila spec (in LPI)
VS
p4v spec
Vera spec (NetCTL)

```
@pkt.$order == <eth [vlan] (ipv4|ipv6) tcp> => keep(tcp)
```

```
parse_eth:
    last := eth
parse_vlan:
    assume last == eth
    last := vlan
parse_ipv4:
    assume last == eth || last == vlan
    last := ipv4
parse_ipv6:
    assume last == eth || last == vlan
    last := ipv4
parse_tcp:
    assume last == ipv4 || last == ipv6
    last := tcp
    @tcp.src_port := tcp.src_port
    @tcp.dst_port := tcp.dst_port
    ...
parse_udp:
    last := udp

assume last == tcp
assert tcp.src_port == @tcp.src_port
assert tcp.dst_port == @tcp.dst_port
...
```

```
InstructionBlock(
    CreateTag("START", 0),
    Call("router.generator.eth.ipv4.tcp"),
    res.initFactory(switchInstance)
)
InstructionBlock(
    CreateTag("START", 0),
    Call("router.generator.eth.ipv6.tcp"),
    res.initFactory(switchInstance)
)
InstructionBlock(
    CreateTag("START", 0),
    Call(router.generator.eth.vlan.ipv4.tcp),
    res.initFactory(switchInstance)
)
InstructionBlock(
    CreateTag("START", 0),
    Call(router.generator.eth.vlan.ipv6.tcp),
    res.initFactory(switchInstance)
)

AF(Constrain(tcp.src_port, Eq(Original.tcp.src_port)))
AF(Constrain(tcp.dst_port, Eq(Original.tcp.dst_port)))
...
```

Figure 3: Specification comparison with prior work. This example describes a real property: for each packet with headers 'eth, optional vlan, ipv4 (or ipv6), tcp', the tcp header remains unchanged after passing through the switch.

```
action a1() { ig_md.ttl = ipv4.ttl; }
action a2() { ig_md.drop = 1; }
action a3() { ipv4.ttl = ig_md.ttl; }
apply {
    a1();
    if(ig_md.ttl == 0) { a2(); }
    // ig_md.ttl = ig_md.ttl - 1; //bug: statement missing
    ...
    a3();
} // Specification: assert ipv4.ttl == @ipv4.ttl - 1
```

Figure 4: A bug example in switch.p4 (pipeline 0).

Figure 2 shows a real example of a single programmable switch in our edge network. We call such an architecture involving multiple P4 functions in a single programmable switch as *hyper-converged data plane*. In this example, this switch uses $P4_{16}$ to implement four functions (switch, load balancer, scheduler and firewall) across its four pipelines, respectively. Pipeline 1 and 3 employ pipeline recirculation to allow packets to go through the ingress and egress programs multiple times. Note that we can also implement multiple functions within the same pipeline, which depends on business needs and resource constraints in practice. According to the operation policies, traffic from the Internet may need to travel different function-chain paths, such as switch ingress → load balancer egress → load balancer ingress → scheduler egress (*i.e.*, the red arrows shown in Figure 2). Our network engineers control different traffic paths (*i.e.*, routes forwarding) by installing and distributing different table entries to the corresponding switches. See Appendix A for more details about hyper-converged data plane.

The majority of our edge switches employ the above hyper-converged architecture. Such a single switch's data plane typically consists of thousands lines of $P4_{16}$ code, hundreds of tables, each containing hundreds of entries, and a big parser program with thousands of dependencies across states. Our production networks use $P4_{16}$ (rather than $P4_{14}$) because $P4_{16}$ supports many important features such as multi-pipeline control, reordering header via deparsing, and checksum.

Reliability of edge networks in Alibaba is always extremely important. Given the fact that testing coverage is limited, Alibaba decided to build a verification system to provide a rigorous guarantee on the correctness of programmable data planes. Furthermore, our network engineers propose their practical usage requirements for such a verification system: *a practically usable verification system should simultaneously achieve four features (1) expressing specification with ease, (2) scalable verification, (3) automatic and accurate violation localization, and (4) self validation; otherwise, the system would be hard to use in production.*

In the rest of this section, we first use Figure 2 example to illustrate why the state of the art does not meet the above features, and then present Aquila, the first practically usable verification system.

(1) Specification expressing. Our engineers need to specify diverse properties to express the correctness of the hyper-converged data plane, shown in Figure 2, such as (*i*) is a specified packet processed according to the sequence shown in Figure 2, and (*ii*) whether table *a* in the egress scheduler is only hit by a specified packet. Existing tools (*e.g.*, Vera [48] and p4v [30]) only provide low-level languages, which are hard to express our intent. Figure 3 shows an example snippet that expresses a real property: for each packet with headers 'eth, optional vlan, ipv4 (or ipv6), tcp', the tcp header remains unchanged after passing through the switch. We can observe that p4v and Vera describe such a simple property with many lines of specifications.

(2) Verification. Our production P4 programs, *e.g.*, Figure 2, are large and complex. Existing verification approaches are not scalable to check them. For example, when we used p4v and Vera to check `switch.p4` in Figure 2, they triggered out-of-memory and timeout, respectively, let alone the entire hyper-converged program.

(3) Bug localization. Manually localizing bugs in the production program, as shown in Figure 2, is time-consuming, even though we know the specification is violated. Many bug localization tools for general-purpose languages like C and Haskell [7, 27, 40, 44, 53] have been proposed. These tools, in principle, extract a counterexample when violation occurs and then find root causes iteratively negating counterexamples. However, these efforts do not work in P4 debugging context due to two limitations. *(i)* Bugs resulting from statement missing (*e.g.*, the bug in Figure 4), commonly existed in our hyper-converged programs, cannot be localized by prior work. *(ii)* These general-purpose language debugging tools cannot localize the bugs in P4-specific semantics. For example, P4 parser supports

Table 1: Required properties of production data plane program verification. Half tick means partial support.

Parts	Properties	Meaning	Aquila	p4v [30]	Vera [48]
Parser	Header order	Whether header order (*e.g.*, ethernet → ipv4 → tcp) is expected?	✓	✗	✓✗
	Header parsing	Whether packet headers are parsed correctly?	✓	✗	✗
Match-Action Units (MAUs)	Header validity	Whether invalid header (*e.g.*, ipv6 header should not exist) can be detected?	✓	✓	✓
	Field correctness	Whether header field value (*e.g.*, destination IP) is as expected?	✓	✓	✓
	Payload correctness	Whether header payload is as expected?	✓	✗	✗
	Expected table access	Whether a table or an action is hit by a specified packet?	✓	✓	✓
	Table entry validity	Whether a specified individual table entry is handled as expected?	✓	✗	✓
	Wildcard table entries	Whether a property always holds for any possible table entry?	✓	✓	✓✗
	Deparser	Is output packet (*e.g.*, header reordering and checksum) correct?	✓	✗	✗
Switch	Multi-pipeline	Are pipelines executed in the expected order (*e.g.*, red arrow in Figure 2)?	✓	✗	✗
	ASIC behaviors	Do recirculation, resubmit or mirror run as expected?	✓	✗	✓✗
	Register	Are behaviors, values and states of registers expected?	✓	✓	✓

lookahead operation, which parses packet header based on the unparsed portion. Existing general-purpose language debuggers are hard to be adapted to localize the bugs related to the lookahead operation.

(4) Verifier self validation. A verification system implementation might be buggy, significantly affecting the confidence of the network engineers. No prior work can help.

Aquila: A practically usable verification system. We build Aquila capable of simultaneously addressing the above problems. As shown in Figure 1, a network engineer expresses her specification via our proposed language (§3), saving 10× lines of specifications (see Figure 3). Aquila then efficiently checks if the program meets the specification (§4). For violated properties, Aquila calls the bug localization module (§5) to report where are the bugs. Aquila can detect bugs in data plane programs, and also find incorrect table entries. Aquila introduces a self-validator to tell potential Aquila implementation issues (§6).

Aquila can verify two cases: (1) a specific data plane snapshot (*i.e.*, the P4 code along with a snapshot of deployed table entries), and (2) data plane correctness under any possible table entries, *i.e.*, the P4 code without specifying any concrete table entries. For the second case, the network engineers want to check whether the target P4 program always meets the specification for any table entries to be installed. For the table entries potentially triggering bugs, the second case enables us to record these entries in a blocklist ahead of time, preventing them in runtime.

3 SPECIFICATION LANGUAGE

Building an effective verification system for production data plane programs requires us to comprehensively reason about the important correctness properties our network engineers face in real world. By surveying our engineers, Table 1 lists key properties that a practical system should verify, and also shows the comparison with state-of-the-art tools.

Aquila provides a declarative specification language, LPI (Language for Programmable network Intent). In principle, LPI is better for our engineers to use than p4v and Vera due to two reasons. (1) LPI is able to express more properties, widely-needed by our scenarios, which the state of the art failed to consider (see Table 1). (2) LPI encapsulates common property assertions (including table, match and modified), making it easier to express specifications (10× fewer lines of specifications than prior work). Figure 3 shows a more concrete example comparing LPI with the state of the art.

```
spec     ::=  decl*                              Aquila specification
decl     ::=  assump                             Precondition
          |   assert                             Postcondition
          |   prog                               Main body
          |   ...
assump   ::=  assumption '{' block* '}'
assert   ::=  assertion '{' block* '}'
block    ::=  blk_id '=' '{' stmt* '}'
prog     ::=  program '{' stmt* '}'
stmt     ::=  if '(' exp ')' '{' exp* '}'        If condition
          |   exp ';'                            Simple stmt
          |   ...
exp      ::=  '<' hdr* '>'                       Header sequence
          |   '#' string                         Variable declaration
          |   '@' field_id                       Get init value
          |   exp op exp                         C-like operators
          |   assume '(' blk_id ')'              Insert assumption
          |   assert '(' blk_id ')'              Exec assertion
          |   keep '(' field_id ')'              Field keeps
          |   match '(' tbl_id, act_id ')'       Hit table, action
          |   modified '(' field_id ')'          Field modified
          |   ...
hdr      ::=  string                             Simple header name
          |   '[' hdr* ']'                       Optional header
          |   '(' hdr* ('|' hdr*)* ')'           Header branch
...           ...
```

Figure 5: Aquila's specification language grammar

An LPI specification requires three parts: (1) precondition definition (including input packet and switch initial state), **assumption**, (2) expected behaviors of a single pipeline, **assertion**, and (3) expected behaviors of entire device, **program**. Figure 5 presents the grammar of LPI. Figure 6 shows an example specification that checks a P4 program `forward.p4`, which changes TCP and UDP packets whose destination IP is 10.0.0.1 to 10.0.0.2. The rest of this section illustrates the LPI usage based on this example.

Assumption. LPI allows us to define three types of preconditions: (1) the initial state of switch (*e.g.*, the values in the register), (2) the input packet's headers (*e.g.*, the header field value and header order), and (3) metadata (*e.g.*, mirror state and input port). Specifying (1) and (3) is straightforward via operators like '=='. For example in Figure 6, line 4 is a metadata specification that specifies the packet comes from even port number of the switch. To specify (2), *i.e.*, packet headers, we introduce keywords '< >' for the network engineer to describe the expected header order. Line 5 in Figure 6 specifies expected order for TCP and UDP headers.

Assertion. In the assertion part, LPI enables us to specify the correctness of a single pipeline. LPI allows checking the value of an input packet in any condition at any position in the pipeline: besides

```
1   config {path = ./forward.p4;}
2   assumption {
3     init {
4       ig_md.ingress_port & 0x1 == 0;          // Even port#
5       pkt.$order == <ethernet ipv4 (tcp|udp)> // TCP or UDP header
6       pkt.ipv4.dst_ip == 10.0.0.1;            // Dst. IP
7   }}
8   assertion {
9     pipe_in = {
10      if (@pkt.ipv4.protocol == 6)     // TCP header
11        pkt.ipv4.dst_ip == 10.0.0.2;   // Send to 10.0.0.2
12      if (match(fwd,send))        // Match table fwd, action send
13        modified(pkt.ipv4.dst_ip);     // Dst. IP is modified
14    }
15    pipe_out = {...}
16  }
17  program {
18    assume(init);                    // Add assumption
19    call(ingress_pipeline);          // Execute ingress program
20    assert(pipe_in);                 // Check pipe_in assertion
21    #quit = (ig_md.drop == 0)        // Skip the egress pipeline
22      || (ig_md.to_cpu == 0);        // if dropped or sent to CPU
23    if (!#quit) {
24      call(egress_pipeline);
25      assert(pipe_out);
26  }}
```

Figure 6: Example specification program

regular value checking via '==', LPI allows us to use logical connectives to construct more complex conditional checking. Line 10-11 in Figure 6 show such an example that checks the destination IP when the input packet has a TCP header. The '@' symbol allows us to get the value before the packet entering the switch.

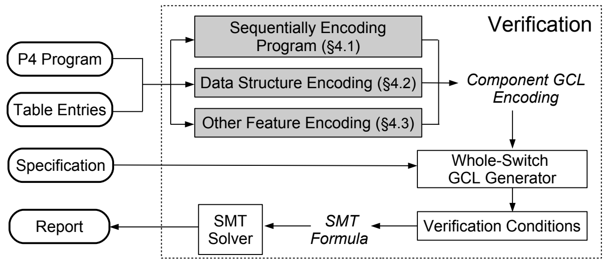
LPI can also specify operation properties, such as whether some value is changed successfully before a table is applied, and whether an action in a table is hit by a specific packet. We use 'match' to specify whether a specific action in a table is hit and 'modified' to check whether a specific header or metadata is modified, no matter what the modified value is. Engineers can express different specification for different actions of the same time. In Figure 6 example, line 12-13 indicate the input packet hits the `fwd` table, `send` action and the destination IP is modified.

Program. LPI uses `program` to connect assumptions and assertions to their corresponding data plane modules, forming the entire specification. For example, line 18 and 20 surround the ingress pipeline encoding (line 19) with **assumption** 'init' and **assertion** 'pipe_in' before and after its execution, respectively. LPI also allows us to specify customized connections between pipelines. For example, line 21-22 define a ghost variable *quit* and skips the egress pipeline when *quit* holds. Aquila also supports bounded recirculation by defining a `recirc` with the maximum allowed recirculations.

During execution, Aquila first parses the specification program, rejects it if it is either syntactically or semantically incorrect. Then, it reads P4 program provided in the `config` section. Finally, Aquila verifies whether the input program meets the specification.

4 VERIFICATION APPROACH

Aquila's verification employs Dijkstra's classic methodology to check a program based on the predicate transformer semantics [10]: given a specification, a P4 program, and table entries, we first encode individual P4 components into Guarded Command Language (GCL); then, we follow the specification program to compose component GCLs into a whole-switch GCL and insert assumptions



Verification
P4 Program
Table Entries
Sequentially Encoding Program (§4.1)
Data Structure Encoding (§4.2)
Other Feature Encoding (§4.3)
Component GCL Encoding
Specification
Whole-Switch GCL Generator
Report
SMT Solver
SMT Formula
Verification Conditions

Figure 7: Verification process of Aquila. Gray components are our contributions.

and assertions at designated places; next, the entire GCL is translated into verification conditions; finally, we invoke an SMT solver (*e.g.*, Z3 [9]) to check whether the specification holds. Note that such a verification methodology is widely used in the verification community—for example, p4v [30] also adopts the similar principle; thus, we make no claim it is novel by itself.

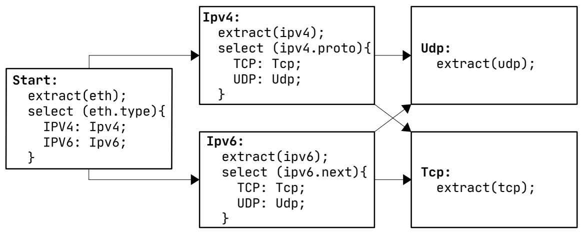
Nevertheless, we find that directly applying the existing approaches to verify our hyper-converged programs will result in state explosion for the following reasons. First, the parser in our production program is a big state machine, and previous approaches would further expand it into thousands of states and branches. And second, our production programs have hundreds of tables, each containing hundreds of entries, and the match-action dependencies across these tables would give rise to complex logical relations. Given these situations, the formula size generated by existing approaches can easily outgrow a solver's compute capability.

To address the above challenges, our solution intuition is to circumvent the exponential growth of states associated with the upscaling of dataplane programs to production level. Therefore, we propose a new encoding approach (*i.e.*, the gray components in Figure 7) that can generate a "compacted" GCL representation with a smaller number of variables and lower complexity, thus ensuring the generated formula simple enough to solve. The entire GCL composition and verification condition translation parts straightforwardly follow Dijkstra's classic way [10, 30], so we skip the details. This section first describes our core sequential encoding approach for P4 program (§4.1), then shows data structures encoding, *e.g.*, packet and table (§4.2), and new features encoding such as inter-pipeline packet passing (§4.3).

4.1 Sequential Encoding

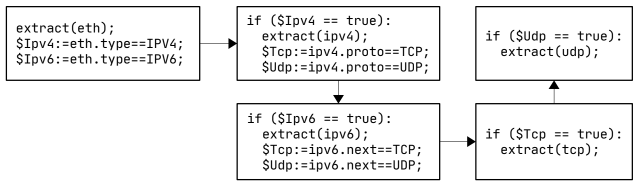
We propose a sequential encoding approach to encode the main body of a data plane component (including parser, MAU, and deparser) into a compact GCL representation, *i.e.*, Component GCL Encoding in Figure 7. In §4.1, we use parser as an example (shown in Figure 8) to first illustrate the key reason for state explosion and then present how our sequential encoding works. This approach also applies to encoding match-action dependencies across tables in MAUs.

The state explosion problem. The parser in P4 uses a state machine model. As shown in Figure 8(a), a parser parses a TCP/UDP packet with either an IPv4 or IPv6 header, and it has five states. If we naively encode this state machine into if-else GCL statements, the result would involve seven states, *i.e.*, a tree structure with



```
Start:
  extract(eth);
  select (eth.type){
    IPV4: Ipv4;
    IPV6: Ipv6;
  }

Ipv4:
  extract(ipv4);
  select (ipv4.proto){
    TCP: Tcp;
    UDP: Udp;
  }

Ipv6:
  extract(ipv6);
  select (ipv6.next){
    TCP: Tcp;
    UDP: Udp;
  }

Udp:
  extract(udp);

Tcp:
  extract(tcp);
```

(a) An example parser state machine



```
extract(eth);
$Ipv4:=eth.type==IPV4;
$Ipv6:=eth.type==IPV6;

if ($Ipv4 == true):
  extract(ipv4);
  $Tcp:=ipv4.proto==TCP;
  $Udp:=ipv4.proto==UDP;

if ($Ipv6 == true):
  extract(ipv6);
  $Tcp:=ipv6.next==TCP;
  $Udp:=ipv6.next==UDP;

if ($Tcp == true):
  extract(tcp);

if ($Udp == true):
  extract(udp);
```

(b) Sequential encoding result

Figure 8: Example for sequential encoding.

seven nodes. It is easy to learn such an encoding structure grows exponentially as the number of states increases. The key problem is that a tree-like GCL, unlike a state machine, has to explicitly embed all possible execution paths, leading to inevitable state duplication. In the worst case, a DAG with $O(n)$ nodes can be expanded into a tree with $O(2^n)$ nodes. In one of our production programs, for example, the parser has only 30 states; however, due to the IP option header, the expansion results in 1174 states and there are hundreds of complex table dependencies and actions corresponding to each of these states, potentially leading to out of memory.

Our algorithm. Our sequential encoding addresses this scalability challenge. We observe that each state (*e.g.*, rectangles in Figure 8(a)) can be visited at most once. Thus, we perform topological sorting on the DAG, introduce ghost variables[1] to indicate the activation of states, and encode the graph as a straight-line program (*e.g.*, Figure 8(b)). This encoding has only $O(n)$ complexity. The algorithm works as follows:

- (1) Given a state machine DAG, S, we order all its states by topological sorting [6] and put the result in a vector E.
- (2) For each state $e_i \in E$, we look for all statements that will result in state transitions (*e.g.*, statements in `select`). We gather its path condition and translate the transition into an assignment to a ghost variable as below:
  `select` (C) `{... V: L}` $\Rightarrow$ `$L :=` $C == V$
  For example in Figure 8(b), the state transition statement `select(eth.type){IPV4:Ipv4}`, is replaced by an assignment expression `$Ipv4:=eth.type==IPV4`.
- (3) Finally, except the 'Start' state, we enclose each state in an if-statement, using its label as the entry condition. In Figure 8(b) example, an `if($Ipv4==true)` is added to guard the second state. In the end, E contains the straight-line program that encodes the original DAG S.

[1]Ghost variables are assignable variables that appear in program encoding but do not correspond to actual variables in the header or program, such as `$Ipv4` in Figure 8(b)

By far, we sequentially encode a parser program. Given a state machine in Figure 8(a), the sequential encoding algorithm results in Figure 8(b). The chain of states represents the union of all possible traces in the DAG, some states can be skipped based on the input packet's value. For example, if an IPv4 packet with TCP header comes in, then `$Tcp` is true and `$Udp` is false, the UDP state is skipped after the TCP state is visited. Compared with prior work, the sequential encoding algorithm achieves $O(n)$ complexity and significantly speeds up the verification later. In the same IP option parser example, the new encoding method reduces number of encoded states from 1174 to 30.

Handling loops. The above algorithm has one assumption that the state machine's dependency graph is a DAG. However, loop may exist when parsing complicated headers like TCP options. Aquila introduces another approach to encode the entire loop into one state and the rest of the graph is encoded via the sequential encoding approach. In short, we first break the loop by removing the transition to the root state (a state that has incoming edges from outside the loop), then encode the states via sequential encoding, and finally surround the encoded statement with a `while` loop in the format of GCL. Similar to the sequential encoding algorithm, the loop exits when the value monitored by the `while` loop is set to false, and the followup state is applied based on other labels in the loop. Due to limited space, see Appendix B.1 for details.

4.2 Data Structure Encoding

Besides sequential encoding, we also encode two important data structures, table and packet, to further compress the resulting GCL.

Table encoding. In a P4 program, a table with multiple entries can be treated as an ordered switch-statement where each entry represents a case branch: a table entry's match filed (*i.e.*, condition) and its corresponding action (*i.e.*, branch body). In our production program, tables may contain thousands of entries, encoding table along with its entries naively with if-statements would easily result in memory explosion. Thus, a memory-efficient table encoding is important. More specifically, we encode the table via a two-step approach: (1) use Action BitVector (ABV) to encode each entry in the table to avoid branches and (2) use a tree like lookup algorithm to encode the search operation among the ABVs.

Action BitVector (ABV). The ABV decouples the condition and branch body by encoding the action index, action parameter into a fixed-length bitvector. The entries of a table are encoded as a list of match conditions and the corresponding ABVs. The table apply operation is separated into (1) looking up the match condition according to the matched field to get the first matched ABV, and (2) applying the corresponding action based on the action and parameter field of the ABV. This encoding decouples the match and action operation, significantly reduces the memory footprint. More details about the format and implementation of ABV can be found in Appendix B.3.

ABV lookup. The naive solution of looking up the ABVs is to use the ITE (If-Then-Else) statement and simulate the one-by-one matching. to simulate the one-by-one matching. It generates deep AND-OR nested expression, which is hard to verify. We instead encode ABVs into a balanced lookup tree via the following formula:

$$ABV_{l,r} = \textbf{ite}\left(Match_{l,\frac{l+r}{2}}, ABV_{l,\frac{l+r}{2}}, ABV_{\frac{l+r}{2},r}\right),$$
$$Match_{l,r} = Match_{l,\frac{l+r}{2}} \vee Match_{\frac{l+r}{2},r},$$

where **ite** is the If-Then-Else operator. This optimization reduces the lookup complexity from $O(n)$ to $O(log(n))$.

Packet encoding. Previous approaches (*e.g.*, p4pktgen [36] and p4v [30]) model the input packet as a huge bitvector with a maximum length. Such an encoding method adds significant overhead to the SMT solver. Because in the deparser, the header fields had to be assigned back to the bitvector. Each assignment creates a new copy of the bitvector in the final logic expression, quickly expanding the memory cost. Aquila introduces a simple, key-value based structure: encoding packet header operations as (key-value) assignments, which gets rid of the huge vector copying issue. For example, the packet operation `extract(eth)` is encoded as `eth:=pkt.eth`. Key-value based encoding requires us to additionally maintain the header order, which can be encoded as a sequence.

Packet encoding makes lookahead no longer trivial. Constraints about the bits looked ahead in previous states should be considered. We move this detail to Appendix B.2.

4.3 Encoding Other Features

Aquila can encode new features from P4$_{16}$ (*e.g.*, inter-pipeline packet passing and recirculation), and other data structures (*e.g.*, hash). Please see Appendix B.4 for details.

5 AUTOMATIC BUG LOCALIZATION

Understanding the violated properties is not enough, since localizing bugs is tedious and time-consuming for our engineers. Prior efforts were extensively proposed to localize bugs in C and Haskell programs [7, 27, 40, 44, 53]; however, they do not fit in P4 programs for the following reasons. First, existing work is unable to localize bugs resulting from statement missing (*e.g.*, Figure 4), which commonly existed in our development. Second, the general-purpose language debugging tools cannot localize the bugs in P4-specific semantics such as bugs in parser and lookahead.

The intuition of our solution is to first narrow down suspects based on reported violations (§5.1), and then pinpoint the buggy code snippets by simulating a fix for each suspect (§5.2). Our bug localization approach not only localizes bugs for statement missing, but also speeds up debugging process by leveraging table action as granularity.

5.1 Finding Violated Assertion

When a violation is reported, the first step of bug localization is to find all the violated assertions. Specifically, we first let the solver only return the first violated assertion. Then, we remove it from the specification and iterate m times to find all violated assertions. For each assertion $assert_i$, we add labels $before_i$ and $after_i$ before and after the assertions, respectively. So the specification is finally encoded as: $before_i \wedge (\neg assert_i \vee (assert_i \wedge after_i \wedge (before_{i+1} \cdots)))$. Such an expression guarantees that, if $assert_i$ is violated, the expression can be simplified into $before_i \wedge true$. Thus, the satisfiability of the above expression is only determined by $before_i$. All the following variables (*e.g.*, $after_i$) are irrelevant to satisfiability and do not appear in the counterexample found by the solver, narrowing down our search space later.

```
control BugExample(md) {
  action a1() { md.ttl = ipv4.ttl; }
  action a2() { ipv4.ttl = md.ttl - 1; }
  table t1() {
    key = { ipv4.dst_ip : exact; } // table entry bug
    actions = { a2; }
  }
  apply { a1(); t1.apply(); }
} // assert ipv4.ttl == @ipv4.ttl - 1
```

Figure 9: Localizing table entry bug.

5.2 Bug Localization

We now localize the bug. Bug localization requires Aquila to point out the potential locations that could potentially fix the violation [44, 53], such as replacing table entries and changing a statement in the **action granularity**. A fix should correspond to a potential bug location. Because one bug may have multiple fixes, our goal is to report the minimal scope of program snippet that may trigger the violation.

Preparation. Before localizing the bug, we first use SMT solver to return a counterexample, which exhibits concrete values for all the variables to trigger the violation. We assign these values to the target program in order to "freeze" the input. Also, we record the actions that the counterexample triggers. The preparation can reduce the search space. Next, we run the following algorithms to locate the bugs.

Table entry bug localization. A table entry bug means the table does not behave as expected, such as incorrect table entry, and a table entry missing. For example in Figure 9, table `t1` has no entry, and action `a1` is never executed, and the assertion is violated. To localize this bug, our table entry bug localization algorithm first finds the correct entries to rectify the table's behavior. According to §4.2, we can encode the table's lookup result into the ABV and execute corresponding actions. Thus, Aquila tries to find such entries by replacing all the tables with variables and let the SMT solver search a valid entry for each table. In Figure 9, for example, `t1` is replaced with a variable `v1`. If the solver can find such valid entries, then we continue the next step to report the minimal set of potential locations; otherwise, the bug should be in data plane. In this case, the SMT solver finds a solution that matches the input packet's destination IP and action `a2`, which means the table entry is buggy.

To minimize the potential locations, for each table, we use an indicator variable rep_i to represent whether a table should be replaced with a function variable fv_i. As a result, the table is encoded as $t_i = \textbf{ite}(rep_i, fv_i, e_i)$, where e_i is the encoded table entries (ABV). Now, we solve the same problem with an optimization goal $\text{MAXSAT}_i \neg rep_i$, obtaining a minimal subset of table entries potentially triggering the bug.

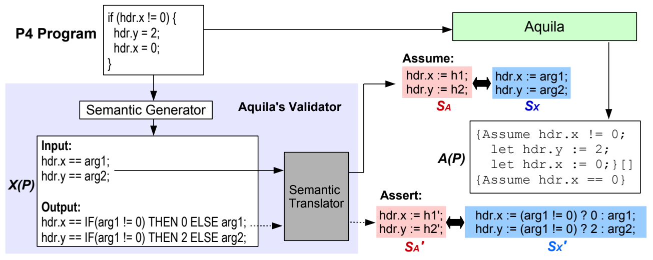
Localizing bugs in P4 program. Intuitively, we simulate a fix by overwriting variables in the suspect actions and checking whether the violation could be fixed. We use SMT solver to find a valid fix. The algorithm works as follows:

- (1) Given the counterexample in the preparation stage and violated assertion v (§5.1), we employ a reverse taint-analysis approach that checks each action in the program backward, and identify and put the action that potentially causes v in a list A. In Figure 4, because `ipv4.ttl` assertion is violated, we use a trace list to record correlated variables and actions, such as `ipv4.ttl` and `ig_md.ttl`.

- (2) We continue filtering the actions in A by checking the causality between each action a_i in A and the violated assertion v. For each a_i, we use the SMT solver to check whether v implies the execution of a_i; if not, we remove a_i from A. Finally, we get a new list A'. This step mainly aims at reducing false positives in our debugging result.
- (3) We maintain a variable list R, which contains all variables that appeared in v. We now check each action a'_i in A' backward from the end of the program. For each a'_i, we put all of its variables in R. Then, for each $r_i \in R$, we create a new statement s that overwrites r_i with an arbitrary "havoc" value, and then insert s below a'_i. We use SMT solver to check whether the new program (*i.e.*, with the newly added statement s) fixes the violated assertion v. If so, this means the action is a candidate root cause for the violated assertion. Such a design enables us to find the location of missing statements. To minimize the location scope, we can again use the minimal satisfiability optimization of SMT solver to get a minimal set.
- (4) Until all actions in A' are checked. We have multiple potential bug root causes for the violation.

An example for localizing statement missing bug. We now use Figure 4 to illustrate how do we localize the statement missing bug. Suppose we meet a violation and get a counterexample: when `ipv4.ttl==1`, the TTL computation is wrong. In the preparation phase, we record actions `a1` and `a3`, because `a2` is removed because it does not trigger the counterexample. In the bug localization phase, (1) we check each action backward in Figure 4 program. Because actions `a1` or `a3` may result in the violated assertion `ipv4.ttl==ipv4.ttl-1`, our list $A = \{a1, a3\}$. (2) $A = A'$ in this example, because the violated assertion implies both the actions. (3) For each action in A', we simulate the "fix" for the violated assertion. Due to limited space, we skip previous fixing steps and go directly to line 5. In line 5, for variable `ig_md.ttl`, we insert one new statement `ig_md.ttl=havoc_i` below `a1`, which overwrites `ig_md.ttl` in `a1`. The new program does not have the violation, which means line 5 or line 6-8 are potential locations for the bug, because line 9 is another action. By optimization mentioned earlier, we can narrow down the scope from line 6-8 to line 7-8. This debugging result means we can fix the bug by two ways: (1) adding `ig_md.ttl=ig_md.ttl-1` in line 7 or 8; and (2) changing the statement in `a1` into `ig_md.ttl=ipv4.ttl-1`.

The "accuracy" of bug localization. In practice, the bug localization of Aquila may not always be just one line in data plane code or one table in the control plane; the result may contain a block of data plane program or multiple tables for complex switch functions. Thus, the bug localization of Aquila aims to narrow down the scope of potential bugs in the "best effort" way. For a data plane program violating specification, it may have multiple potential buggy sources, and fixing one of them would enable the program to meet the specification; thus, our approach can tell the programmer a small enough scope for the potential bug.



```
if (hdr.x != 0) {
  hdr.y = 2;
  hdr.x = 0;
}
```

```
Input:
hdr.x == arg1;
hdr.y == arg2;

Output:
hdr.x == IF(arg1 != 0) THEN 0 ELSE arg1;
hdr.y == IF(arg1 != 0) THEN 2 ELSE arg2;
```

```
hdr.x := h1;      hdr.x := arg1;
hdr.y := h2;      hdr.y := arg2;
```

```
{Assume hdr.x != 0;
  let hdr.y := 2;
  let hdr.x := 0;}[]
{Assume hdr.x == 0}
```

```
hdr.x := h1';     hdr.x := (arg1 != 0) ? 0 : arg1;
hdr.y := h2';     hdr.y := (arg1 != 0) ? 2 : arg2;
```

Figure 10: Self validation of Aquila. We put Assume, A(P), and Assert in a solver to verify equivalence.

6 SELF VALIDATION OF AQUILA

Bugs in Aquila are headaches in reality, as they significantly affect the confidence of our network engineers in the verification result. To make Aquila bug-free, the most rigorous way is to build Aquila by a fully verified toolchain and formally verifying the implementation logic of Aquila, which however are not amenable for an industry-level system. Instead, we take a more practical perspective to ensure the correctness of Aquila implementation. Given the fact that GCL and its verification via SMT solvers were well studied with various tools available for use in the past ten years, we have confidence in including them in our trusted computing base; on the contrary, the component encoding part in Aquila is error-prone based on our experience. Thus, in Aquila implementation, we trust the whole-switch GCL composition, verification condition generation and SMT solver (white boxes in Figure 7), and mainly check implementation bugs in the component encoding (gray boxes in Figure 7).

To validate the implementation of Aquila, we take a translation validation [34, 41] approach: we employ alternative tools to generate semantics for the P4 program, and compare it with the GCL extracted by Aquila. If the two are equivalent, we have confidence that Aquila is implemented correctly; otherwise, we examine the discrepancy and identify bugs based on returned counterexample.

Refinement proof. We employ refinement proof [33, 39] (see figure below) to construct our translation validation.

$$\begin{tikzcd}
s_A \arrow[r, "A(P)"] \arrow[d, dash, "R"'] & s'_A \arrow[d, dash, dashed, "R"] \\
s_X \arrow[r, dashed, "X(P)"] & s'_X
\end{tikzcd}$$

Given a P4 program P is encoded in two different formal languages. We use $A(P)$ to denote the one produced by Aquila and $X(P)$ to denote an alternative representation. Their equivalence means the observable effects of $A(P)$ and $X(P)$, *i.e.*, its processing on packet states, should be the same. In particular, it relies on a relation R that connects equivalent states that describe the same packet but in different formal languages. Assume that $A(P)$ transitions from an initial state s_A to a final state s'_A. Starting from an equivalent initial state s_X such that $R(s_A, s_X)$ holds, $X(P)$ must transition to a final state s'_X where $R(s'_A, s'_X)$ also holds. Such proof validates that $A(P)$ and $X(P)$ has equivalent effects on all packets.

How to construct a refinement proof in our Hoare-style framework? By looking A as the representation generated by Aquila, we can define pre and post-conditions for a given program, and it checks whether the following holds:

$$\forall s_A, s'_A, pre(s_A) \wedge (s_A \xrightarrow{A(P)} s'_A) \Rightarrow post(s'_A). \tag{1}$$

We thus need to seek another representation for P, *i.e.*, $X(P)$, to validate the trustworthiness of $A(P)$, by defining $pre(s_A)$ and $post(s'_A)$ as the following way:

$$\begin{aligned} pre(s_A) &\equiv R(s_A, s_X), \\ post(s'_A) &\equiv R(s'_A, s'_X) \wedge (s_X \xrightarrow{X(P)} s'_X). \end{aligned} \tag{2}$$

Self validator of Aquila. The key challenge in building a self validator is semantic translator (gray box in Figure 10), rather than alternative representation selection or semantic generator. In our implementation, we chose a recent effort, Gauntlet [43], to provide alternative representation, because it defines a big-step semantics for each individual component (parser, match-action unit, deparser). This enables us to implement semantic translator easier.

As shown in Figure 10, we use Gauntlet as the semantic generator to translate a given P4 program P into a representation $X(P)$—it computes symbolically the output value of every header field. This leads to a straightforward refinement relation between s_A and s_X: we simply require that every header field in s_A is identical to its counterpart in s_X.

Building semantic translator is non-trivial. It is used to translate the representation $X(P)$ into the precondition and postcondition for $A(P)$ in the format of guarded command language, *i.e.*, Assume and Assert in Figure 10. Intuitively, we construct an expression representing equation (1), and thus we can check the validation by using SMT solver to check it. Due to limited space, see Appendix C for more details.

Identifying bugs. Aquila's self validator cannot directly pinpoint the implementation bugs; thus, we get a counterexample when inequivalence occurs, and then analyze where are bugs in a semi-automatic way.

7 DEPLOYMENT EXPERIENCE

Aquila has been used by Alibaba's network engineers to verify the correctness of data plane programs in the edge networks for half a year. It has successfully guided our network engineers in avoiding many potential critical failures. After Aquila is used online, no service failure resulting from data plane bugs occurred so far.

This section first presents representative scenarios and bugs in our experience with Aquila (§7.1). Then, we show example bugs detected by Aquila's self validation (§7.2).

7.1 Scenarios and Bugs

Bugs detected by Aquila in the past months mainly include: (1) unexpected program behaviors (*e.g.*, invalid header and out-of-register), (2) incorrect table entries, (3) incorrect service-specific properties (*e.g.*, buggy actions, and incorrect packet processing logic), and (4) wrong call sequence of multi-pipeline.

This section selects three representative scenarios, including a single P4 gateway program, a hyer-converged data plane (*i.e.*, Figure 2), and data plane update, to illustrate the practicality of Aquila. Table 2 compares Aquila's specification complexity with two existing tools p4v and Vera in the verification of these three scenarios. We reproduced p4v system [30], and used Vera open-source prototype [1, 2]. Note that in following scenarios, both p4v and Vera ran out of memory.

Table 2: Comparing lines of specifications.

	Scenario 1	Scenario 2	Scenario 3
Aquila	O(10)	O(100)	O(100)
p4v	O(100)	O(1000)	O(1000)
Vera [1, 2]	Vera's APIs are not flexibly to express our specifications		

Scenario 1: Traffic statistics for monitoring. Our monitoring system for edge networks relies on statisticizing the incoming business traffic. For a given business traffic, it should be first forwarded by a metropolitan router to a VXLAN gateway. Then, this VXLAN gateway copies these packets and sends the original packets back to that metropolitan router. For the copied packets, the VXLAN gateway encapsulates and sends them to a collection of servers for traffic statistics. These servers have been deployed programs responsible for classifying the received packets into different groups based on their business. Because the rapid traffic growth brought big pressure to these servers, we implemented the traffic statistic logic in P4 switches to replace those servers.

We used Aquila to verify this P4 program, which is important. A small error would mess up our monitoring system. The specification includes: (1) when a known type of packet p comes, whether the number and states of p is statisticized correctly, (2) when a new (unknown type of) traffic packet q comes, whether a correct metadata is successfully added to q's header, and (3) whether q's fields are evaluated correctly. For example, whether each packet with destination IP address 10/8 is successfully added a queue length metadata information in its header and whether the DSCP value of each packet destinating 20/8 is changed to be three.

Aquila detected two data plane bugs in one second. The first bug was detected in the old traffic handling component of VXLAN gateway P4 program. As mentioned above, this component should send the original packets p back to the metropolitan router, enabling the backend servers to handle these packets normally; however, in this bug case, the program incorrectly sets the metadata of the original traffic flow packet, *i.e.*, p, to zero, causing the backend servers to read the incorrect state of p. The second bug is caused by a "copy-and-paste" error: when our engineers directly copied and pasted the register value assignment in our P4 program responsible for statisticizing the incoming traffic flow, they forgot to change some of the pasted P4 code.

Scenario 2: Hyper-converged P4 CDN. In our edge networks, a CDN PoP includes three components: the edge servers providing content delivery service, the middle-boxes providing network functions such as load balancer and DDoS defense, and L3 switches connecting the CDN to ISP network. As motivated in §2, our operators put the functions of middle-boxes (including scheduler, load balancer, firewall, and DDoS defense) and L3 switch into a single programmable switch, as shown in Figure 2, where the most packets are processed by programmable ASICs, and the rest (*e.g.*, cache-missed HTTP requests) are forwarded to switch CPUs.

We used Aquila to verify the CDN's hyper-converged P4 program with a set of correctness specifications, including (1) each function's correctness, (2) undefined behavior checking, (3) the correctness of values passed among different pipelines, and (4) whether recirculation number is correctly bounded. A critical undefined behavior bug was detected. As shown below, for an input packet that does not have an ipv4 header and does not have ipv6 header either, *e.g.*, an ARP packet, the packet should apply the table `egress_ipv4`, if this packet sets `mac_config_on=false`.

```
if (ipv6.isValid()) {
  egress_ipv6.apply(ipv6, eg_state);
} else if (!mac_config_on || ipv4.isValid()) {
  egress_ipv4.apply(ipv4, eg_state);
}
```

An undefined behavior would occur because this packet does not have an ipv4 header. Aquila detected the violation within 40.1 seconds, and localized this bug in one minute.

Another bug we detected in this scenario existed in deparser program. The engineer wanted to reassemble the packet via a predefined `struct` that contains the necessary headers. It was intended to be used elsewhere so the header order is not carefully designed to align with the order in the packet. The engineer failed to realize this issue and unsurprisingly, the returned packet is wrong.

Scenario 3: Checking bugs before updates. Due to our business diversity, the network engineers are frequently required to update data plane programs to meet the service needs. However, the program update is one of the major root causes of significant service disruptions; thus, we use Aquila to verify our updates. For the update scenarios, we typically use the original specification, because we want to ensure the entire program behaves the same before and after updates.

In an important update event, we needed to exchange the pipelines of load balancer and switch, which means we move switch.p4 to Pipeline 1 and loadbalancer.p4 to Pipeline 0 in Figure 2, respectively. This update requirement comes from the following reason. The load balancer pipeline contains packet processing functions that involve virtual network encapsulation/decapsulation, such as network address translation (or NAT). They may change the forwarding source or destination address of a packet, which the switch pipeline relies on. For example, NAT translates the destination address from a public IP to an internal IP, which the switch pipeline uses to forward the packet to the correct server. Thus, our engineers were required to place loadbalancer.p4 in front of switch.p4; otherwise, intricate metadata has to be used to make sure the switch forwarding behavior is correct.

Such an update task required us to modify many parts of P4 programs, such as the input packet format of load balancer program, and adding recirculation to switch. It is prone to causing functional inconsistency before and after the update. Aquila detected a critical bug. We have two tables, an ACL table *a*, which accepts IP address 10.0.1/24 but drops IP address 20.0.1/24, and a forwarding table *b* which changes the IP address in a packet header from 10.0.1/24 to 20.0.1/24.[2] Before the update, a packet sent to IP address 10.0.1/24 can be successfully transferred into a packet with destination IP 20.0.1/24 due to table *b*; however, this update moved *b* in front of *a*, so that *b* first changes the packet's IP from 10.0.1/24 to 20.0.1/24, and then the packet is dropped by table *a*'s ACL rules. Once such an update is committed online, all traffic destinating 10.0.1/24 would be blocked. It is hard to manually detect such a bug due to complex logic across pipelines.

[2]The IP addresses have been anonymized for confidentiality reasons.

7.2 Self Validation Experience

The majority of bugs in Aquila were detected in the early stage of Aquila development. The self-validator helped us identify tens of Aquila implementation bugs in total, which were caused by reasons such as language misunderstanding, incorrect function implementation, and chip-specific feature misunderstanding. We also met bugs in the alternative representation implementation. Because Aquila's verification results do not contain any false positive in recent months, we believe the self validator successfully assisted us to tune the correctness of Aquila implementation. We pick two bug examples to explain below.

Language misunderstanding. The developers of Aquila may misunderstand or ignore some features of P4 language, implicitly injecting bugs in Aquila. For example, P4 allows using annotation `@defaultonly` to restrict an action. The initial version of Aquila ignored this feature, which caused a correct program to violate its specification.

Function implemented incorrectly. Function-level implementation bugs accounted for the majority of Aquila's bugs. Some of the functional bugs were tricky to be found. For example, in our initial attempt, our encoding module failed to well handle empty states, implicitly returning headers that are already extracted. In other words, this bug treated an empty state as the 'accept' state, causing the parser encoding to accept more packets than its actual code. The self validator revealed this subtle inequivalence in the parser, which is very unlikely if we rely on human code review.

8 PERFORMANCE EVALUATION

All of our experiments were conducted in a container with 32GB RAM and one 2.5GHz CPU core, with Z3 4.8.5 installed as an SMT solver. We used the end-to-end verification time to evaluate our performance, which includes the parsing, encoding and verification process.

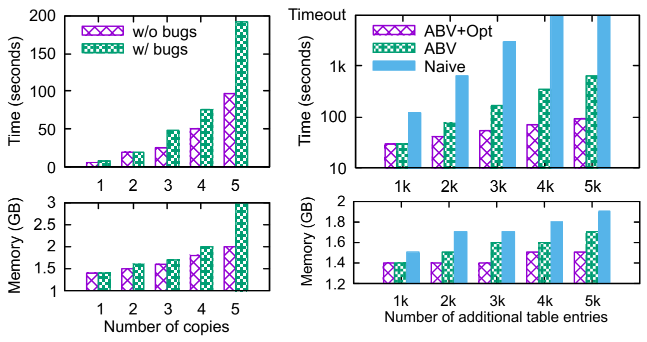
8.1 Benchmark

We chose twelve P4 programs, 8 open-sourced and 4 private ones, to compare the performance of Aquila with Vera and p4v. We asked them to check the invalid header access bugs because it is a good benchmarking property [30]. The chosen programs had at least one such bug. We recorded the time to find the first bug (by checking all assertions together) and all bugs (by checking each assertion one by one) and set 2 hours as the timeout threshold. Note that in all experiments, we make no assumption about the table entries, registers and input packet. Also in [30], p4v only recorded the execution time of finding the first bug in the Switch BMv2 program with INT module disabled, we modified our own p4v implementation to find all bugs. As shown in Table 3, Aquila could report bugs within one second for small programs, and within one minute for large programs. Even for the largest program, Aquila only needed 4.8 Gigabytes of memory. As a comparison, Vera took a significantly longer time to verify the small programs, and p4v run out of memory for large programs due to lack of scalable encoding approaches. We also observed a higher memory footprint when finding the first bug, because Aquila encoded all the assertions at once and maintained more states.

Table 3: Comparing verification time and memory consumption (OOT - Out of Time, OOM - Out of Memory).

Program	LoC	Pipes	Parser States	Tables	Time (s) (Finding first/all bugs)			Memory(GB) (Finding first/all bugs)		
					Aquila	p4v	Vera	Aquila	p4v	Vera
Simple Router	131	1	3	4	0.01 / 0.02	0.01 / 0.02	0.42 / 0.45	0.1 / 0.2	0.1 / 0.2	0.1 / 0.1
NetPaxos Acceptor [8]	185	1	5	4	0.01 / 0.09	0.01 / 0.17	9.71 / 9.71	0.1 / 0.2	0.1 / 0.2	0.2 / 0.3
NetPaxos Coordinator	148	1	5	2	0.01 / 0.03	0.01 / 0.06	4.44 / 6.70	0.1 / 0.2	0.1 / 0.2	0.3 / 0.8
NDP [22]	224	1	4	7	0.01 / 0.04	0.01 / 0.06	0.50 / 0.51	0.1 / 0.2	0.1 / 0.2	0.1 / 0.1
Flowlet Switching	237	1	4	6	0.01 / 0.04	0.02 / 0.06	2.75 / 2.94	0.1 / 0.2	0.1 / 0.2	0.3 / 0.3
NetCache [26]	538	1	17	96	0.17 / 9.56	0.22 / 16.1	0.90 / 241	0.1 / 0.4	0.1 / 0.5	0.1 / 5.4
Switch BMv2 w/o INT	5036	1	59	104	1.26 / 290	197 / OOT	13.5 / OOT	0.4 / 0.7	10.5 / OOT	2.5 / OOT
Switch BMv2	5599	1	64	120	1.41 / 347	OOM / OOM	226 / OOT	0.5 / 0.8	OOM / OOM	12.7 / OOT
Switch from vendor	5453	2	30	141	20.1 / 1286	OOM / OOM	Error / Error[1]	2.5 / 1.6	OOM / OOM	Error / Error
Production Program 1	> 6000	4	41	> 150	23.7 / 558	OOM / OOM	Error / Error	2.5 / 2.6	OOM / OOM	Error / Error
Production Program 2	> 6000	4	47	> 150	25.2 / 733	OOM / OOM	Error / Error	2.9 / 2.6	OOM / OOM	Error / Error
Production Program 3	> 2000	6	114	> 120	41.3 / 3574	OOM / OOM	Error / Error	4.8 / 2.7	OOM / OOM	Error / Error

[2] Vera [1, 2] only supports $P4_{14}$ while the programs are written in $P4_{16}$. Even if it supports $P4_{16}$, we infer it would be OOT from Switch BMv2 results.



(a) Program complexity **(b) Table entries.**

Figure 11: Scalability evaluation.

We used two Switch BMv2 programs, one with the INT module enable and one without. The INT module contains a complex parser module and the additional complexity caused p4v to run out of memory. Vera had a similar pattern that the added INT module increases the verification time by 16× and memory by 5×. On the contrary, due to the sequential encoding, this additional complexity only adds 10% overhead in time and 25% in memory to Aquila. While Vera cannot verify our production programs, since it only supports $P4_{14}$, we infer it would be OOT according to Switch BMv2's result, whose $P4_{16}$ version is a part of our production programs.

8.2 Scalability

The performance of Aquila is directly affected by the complexity of the encoded GCL representation, which is affected by the complexity of the input program and the size of the data structures, *e.g.*, the number of table entries. To evaluate Aquila's scalability, we conducted two experiments based on a vendor-provided `switch.p4` program (called `switch-T`) with thousands lines of code.

The first experiment evaluated the performance under different sizes of the program. Due to the complexity of the mapping between the P4 program and encoded GCL, it is hard to evaluate the scalability by comparing different programs. Instead, we constructed a huge P4 pipeline by connecting k `switch-T` programs, where k ranges from 1 to 5. For each k, we checked two versions, one without bug and one with bugs, and reported the verification time and memory usage. Figure 11a shows even for the largest case (five `switch-T` connected), Aquila finished verification within 200 seconds, and consumed 3GB memory.

Table 4: Bug localization time and precision.

Scale	Wrong Entry		Code Missing		Code Error	
	Time (s)	Prec.	Time (s)	Prec.	Time (s)	Prec.
Large	31.2	100%	164	96.0%	3.01	100%
Medium	21.2	100%	83.2	95.7%	2.92	100%
Small	17.1	100%	68.6	94.8%	1.47	100%

The second experiment focused on the number of entries in the tables and influence of the ABV encoding. We installed different number of entries in the tables and compared the performance against two inferior approaches: the naive encoding that checks entries one by one through if-else branching and the ABV without lookup optimization. As shown in Figure 11b, as the number of entries increases, Aquila's verification time grows logarithmically, with the $O(\log n)$ lookup optimization complexity. The naive approach triggers time out after only 4k entries. The lookup optimization further improved the verification time by up to 6×. Also, across all three approaches, memory only grew mildly, suggesting that the table lookup is a compute intensive job.

8.3 Bug Localization

Finally, we evaluated the bug localization. Based on `switch-T` program, we built three versions: `Large`, which is the original `switch-T` program; `Medium`, with DTEL and sFlow function disabled; `Small`, with QoS, mirroring, L2 forwarding and IPv6 disabled additionally. We injected three bugs, one with a wrong entry installed, one with a statement missing, and one with a wrong statement. To evaluate the performance of the localization algorithm, we introduced a metric called **precision**. The precision denotes the percentage of potential locations that Aquila can filter out. This value directly reflects the amount of effort we can save. A 100% precision means Aquila can accurately locate the bug with no false positive. Table 4 shows Aquila located wrong entry and code error bugs with 100% precision in a few seconds. For the code missing bugs, because there might be multiple potential bug locations, Aquila took longer time, but is still able to maintain the precision around 95%.

9 LESSONS AND DISCUSSIONS

In this section, we share our lessons in designing and deploying Aquila, and also discuss limitations of Aquila and open questions.

Specifying service-specific properties. Most (about 90%) of our specifications are service-specific properties. For example, (1) any packet sending to 10.0.1/24 should not have VXLAN filed in its header, (2) all packets from 20.1/16 should remain the DSCP values after leaving the current programmable switch, and (3) table *arp_t* should not be hit by packets destinating 12.0.2/24. Our experience shows specifying the above properties is much more challenging than undefined P4 behaviors such as invalid header and out-of-register checking. This is because our engineers should well first understand these service-specific properties and then express them in a correct and complete way. While using LPI saves our engineers' property-specifying time, an automatic specification inferring approach based on example-based synthesis [3, 21] might be a potential direction for building more efficient way to express service-specific properties.

Verification or testing? Verification can offer rigorous correctness checking by modeling program of interest and using some prover to prove whether the target program is bug-free; on the contrary, testing techniques provide input-output checking based on the limited test case coverage. In Alibaba network, we need both of above two techniques. Specifically, we are using verification techniques (*i.e.*, Aquila) to ensure the correctness of P4 program in the perspective of packet processing logic; however, Aquila is not able to detect bugs such as compiler and ASIC-specific bugs. Thus, how to build a full-coverage testing technique capable of automatically detecting compiler, hardware and performance bugs in P4 programs is highly needed and is still an open question.

Distributed data plane verification. In Alibaba's edge networks, each programmable switch's data plane program is individual; thus, Aquila is only focused on verifying data plane program on a single switch. As increasingly more new data plane applications are launched in the future, we believe *distributed data plane programs* may play important roles. A distributed data plane program means multiple programmable switches, where switches' data plane programs are different, coordinately work together to process packets for some purpose. For example, a super big table that may not be put in a single switch's ASIC can be split across multiple programmable switches [16]. Verifying such a distributed data plane program—even with heterogeneous underlying ASICs and programming languages—would need to address more challenges.

Automatically bug repairing. While Aquila can detect the potential violations and automatically localize bugs, Aquila is not able to fix the detected bugs. Automatically repairing bugs in data plane programs is very challenging, because (1) the data plane program may have very complex functional logic, (2) a buggy program may have many different fixing solutions, and (3) it is also non-trivial to avoid side effects when producing a potential fixing plan. Thus, we leave how to automatically repair bugs in the future work.

Aquila's usage phase. In Alibaba network, we are mainly using Aquila in two phases: (1) checking data planes during service runtime and (2) checking new data plane programs before updating them in the network. In the service runtime, we aim to use Aquila to check newly-required properties or check data planes if any new table entries are installed. In the update phase, we use Aquila to ensure that the updated data planes still meet our required properties ahead of time.

10 RELATED WORK

Programmable data planes checking. There have been several verification techniques proposed to check correctness of programmable data planes. For example, p4v [30] employs a classic verification approach to formally check P4 program correctness in terms of any table entry. Vera [48] leverages symbolic execution to ensure the correctness of a network snapshot (including both P4 programs and table entries). bf4 [11] targets undefined P4 behavior prevention in runtime by automatically inferring assertions and preventing table rules from potentially triggering bugs. bf4 is a good complementary to Aquila, since Aquila does not explicitly prevent bugs in runtime. p4-assert [35] and p4-NOD [32] translate P4 code into other language models (*e.g.*, C and NoD) which are verifiable by the existing frameworks. In addition, p4pktgen [36] and Netdiff [12] use testing techniques to check programmable data planes.

Network configuration verification. Many efforts were proposed to verify the correctness of network configuration—*i.e.*, whether a given network configuration meets the operation specification. These efforts can be classified into two groups: control plane verification [4, 13, 15, 19, 20, 38, 42, 47, 51, 52, 55, 56], and data-plane forwarding verification [5, 23–25, 28, 29, 31, 37, 49, 50]. These systems focused on checking the properties such as routing equivalence and packet reachability, which are different from Aquila's goal.

Data plane program synthesis. The state-of-the-art efforts in synthesizing data plane program for programmable switching ASICs are mainly focused on building compilers for chip-specific code generation. For example, Lyra [16] and μP4 [46] propose high-level languages for engineers to easily express their programs and the compilers generate chip-specific code such as P4 and NPL. Domino [45] was built to support stateful processing, enabling a wide class of data plane algorithms; as an enhancement effort, Chipmunk [17, 18] leverages the synthesis technique to generate more cases than Domino, and the generated code needs fewer resources than Domino. Dejavu [54] targets compiling a hyper-converged function-chain into a single ASIC.

11 CONCLUSION

This paper presents Aquila, the first-ever reported practically usable verification system for programmable data planes in Alibaba. To achieve the practical usage goal, Aquila makes four contributions: (1) a language for engineers to express intent with ease; (2) a new encoding approach making verification scalable to production-scale data planes; (3) a bug localization approach capable of accurately finding bugs resulting in violation; and (4) a self-validator for identifying issues in Aquila. Aquila has been used to verify Alibaba's programmable edge networks for half a year, and it successfully prevented many potential failures resulting from data plane bugs.

This work does not raise any ethical issues.

ACKNOWLEDGMENTS

We thank our shepherd, Stefano Vissicchio, and SIGCOMM reviewers for their insightful comments. Bingchuan Tian and Chen Tian are supported in part by the National Natural Science Foundation of China under Grant Numbers 61772265 and 62072228. Jiaqi Gao and Minlan Yu are supported in part by NSF grant CNS-1834263.

REFERENCES

[1] Vera old version: P4 program verification. https://github.com/dragosdmtrsc/oldvera.

[2] Vera2: P4 program verification. https://github.com/dragosdmtrsc/vera2.

[3] An, S., Singh, R., Misailovic, S., and Samanta, R. Augmented example-based synthesis using relational perturbation properties. *Proc. ACM Program. Lang. 4*, POPL (2020), 56:1–56:24.

[4] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. A general approach to network configuration verification. In *ACM SIGCOMM (SIGCOMM)* (2017).

[5] Beckett, R., Mahajan, R., Milstein, T. D., Padhye, J., and Walker, D. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *ACM SIGCOMM (SIGCOMM)* (2016).

[6] B.Kahn, A. Topological sorting of large networks. *Commun. ACM 5*, 11 (1962), 558–562.

[7] Christakis, M., Heizmann, M., Mansur, M. N., Schilling, C., and Wüstholz, V. Semantic fault localization and suspiciousness ranking. In *25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2019).

[8] Dang, H. T., Sciascia, D., Canini, M., Pedone, F., and Soulé, R. Netpaxos: Consensus at network speed. In *ACM SIGCOMM Symposium on SDN Research (SOSR)* (2015).

[9] de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008).

[10] Dijkstra, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM 18*, 8 (1975), 453–457.

[11] Dumitrescu, D., Stoenescu, R., Negreanu, L., and Raiciu, C. bf4: towards bug-free P4 programs. In *ACM SIGCOMM (SIGCOMM)* (2020).

[12] Dumitrescu, D., Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. Dataplane equivalence and its applications. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2019).

[13] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., and Varghese, G. Efficient network reachability analysis using a succinct control plane representation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[14] Flanagan, C., and Saxe, J. B. Avoiding exponential explosion: Generating compact verification conditions. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2001).

[15] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[16] Gao, J., Zhai, E., Liu, H. H., Miao, R., Zhou, Y., Tian, B., Sun, C., Cai, D., Zhang, M., and Yu, M. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs. In *ACM SIGCOMM (SIGCOMM)* (2020).

[17] Gao, X., Kim, T., Varma, A. K., Sivaraman, A., and Narayana, S. Autogenerating fast packet-processing code using program synthesis. In *18th ACM Workshop on Hot Topics in Networks (HotNets)* (2019).

[18] Gao, X., Kim, T., Wong, M. D., Raghunathan, D., Varma, A. K., Kannan, P. G., Sivaraman, A., Narayana, S., and Gupta, A. Switch code generation using program synthesis. In *ACM SIGCOMM (SIGCOMM)* (2020).

[19] Gember-Jacobson, A., Viswanathan, R., Akella, A., and Mahajan, R. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM (SIGCOMM)* (2016).

[20] Giannarakis, N., Beckett, R., Mahajan, R., and Walker, D. Efficient verification of network fault tolerance via counterexample-guided refinement. In *International Conference on Computer Aided Verification* (2019), Springer.

[21] Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2011).

[22] Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., and Wójcik, M. Re-architecting datacenter networks and stacks for low latency and high performance. In *ACM SIGCOMM (SIGCOMM)* (2017).

[23] Horn, A., Kheradmand, A., and Prasad, M. R. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[24] Jayaraman, K., Bjørner, N., Outhred, G., and Kaufman, C. Automated analysis and debugging of network connectivity policies. In *Technical Report MSR-TR-2014-102* (2014).

[25] Jayaraman, K., Bjørner, N., Padhye, J., Agrawal, A., Bhargava, A., Bissonnette, P. C., Foster, S., Helwer, A., Kasten, M., Lee, I., Namdhari, A., Niaz, H., Parkhi, A., Pinnamraju, H., Power, A., Raje, N. M., and Sharma, P. Validating datacenters at scale. In *ACM SIGCOMM (SIGCOMM)* (2019).

[26] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., and Stoica, I. Netcache: Balancing key-value stores with fast in-network caching. In *26th Symposium on Operating Systems Principles (SOSP)* (2017).

[27] Jose, M., and Majumdar, R. Cause clue clauses: error localization using maximum satisfiability. In *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2011).

[28] Kazemian, P., Varghese, G., and McKeown, N. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012).

[29] Khurshid, A., Zhou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying network-wide invariants in real time. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[30] Liu, J., Hallahan, W. T., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Cascaval, C., McKeown, N., and Foster, N. p4v: Practical verification for programmable data planes. In *ACM SIGCOMM (SIGCOMM)* (2018).

[31] Lopes, N. P., Bjørner, N., Godefroid, P., Jayaraman, K., and Varghese, G. Checking beliefs in dynamic networks. In *12th USENIX Symposium on Networked System Design and Implementation (NSDI)* (2015).

[32] McKeown, N., Talayco, D., Varghese, G., Lopes, N., Bjørner, N., and Rybalchenko, A. Automatically verifying reachability and well-formedness in P4 networks. Tech. rep., 2016.

[33] Milner, R. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence* (San Francisco, CA, USA, 1971), IJCAI'71, Morgan Kaufmann Publishers Inc., p. 481–489.

[34] Necula, G. C. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2000), PLDI '00, Association for Computing Machinery, p. 83–94.

[35] Neves, M. C., Freire, L., Filho, A. E. S., and Barcellos, M. P. Verification of P4 programs in feasible time using assertions. In *14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2018).

[36] Nötzli, A., Khan, J., Fingerhut, A., Barrett, C. W., and Athanas, P. p4pktgen: Automated test case generation for P4 programs. In *Symposium on SDN Research (SOSR)* (2018).

[37] Panda, A., Argyraki, K., Sagiv, M., Schapira, M., and Shenker, S. New directions for network verification. In *LIPIcs-Leibniz International Proceedings in Informatics* (2015), vol. 32.

[38] Panda, A., Lahav, O., Argyraki, K. J., Sagiv, M., and Shenker, S. Verifying reachability in networks with mutable datapaths. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2017).

[39] Park, D. Concurrency and automata on infinite sequences. In *Theoretical Computer Science* (Berlin, Heidelberg, 1981), P. Deussen, Ed., Springer Berlin Heidelberg, pp. 167–183.

[40] Pavlinovic, Z., King, T., and Wies, T. Finding minimum type error sources. In *Proceedings of ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)* (2014).

[41] Pnueli, A., Siegel, M., and Singerman, E. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 1998), B. Steffen, Ed., Springer Berlin Heidelberg, pp. 151–166.

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

[43] Ruffy, F., Wang, T., and Sivaraman, A. Gauntlet: Finding bugs in compilers for programmable packet processing. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2020).

[44] Sahoo, S. K., Criswell, J., Geigle, C., and Adve, V. S. Using likely invariants for automated software fault localization. In *Architectural Support for Programming Languages and Operating Systems, (ASPLOS)* (2013).

[45] Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., and Licking, S. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM (SIGCOMM)* (2016).

[46] Soni, H., Rifai, M., Kumar, P., Doenges, R., and Foster, N. Composing dataplane programs with μp4. In *ACM SIGCOMM (SIGCOMM)* (2020).

[47] Steffen, S., Gehr, T., Tsankov, P., Vanbever, L., and Vechev, M. Probabilistic verification of network configurations. In *ACM SIGCOMM (SIGCOMM)* (2020).

[48] Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L., and Raiciu, C. Debugging P4 programs with Vera. In *ACM SIGCOMM (SIGCOMM)* (2018).

[49] Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM (SIGCOMM)* (2016).

[50] Tian, B., Zhang, X., Zhai, E., Liu, H. H., Ye, Q., Wang, C., Wu, X., Ji, Z., Sang, Y., Zhang, M., Yu, D., Tian, C., Zheng, H., and Zhao, B. Y. Safely and automatically updating in-network ACL configurations with intent language. In *ACM SIGCOMM (SIGCOMM)* (2019).

[51] Velner, Y., Alpernas, K., Panda, A., Rabinovich, A., Sagiv, M., Shenker, S., and Shoham, S. Some complexity results for stateful network verification. In *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2016).

[52] Wang, A., Jia, L., Zhou, W., Ren, Y., Loo, B. T., Rexford, J., Nigam, V., Scedrov, A., and Talcott, C. L. FSR: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Network (ToN) 20*, 6 (2012), 1814–1827.

[53] Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. A survey on software fault localization. *IEEE Trans. Software Eng. 42*, 8 (2016), 707–740.

[54] Wu, D., Chen, A., Ng, T. S. E., Wang, G., and Wang, H. Accelerated service chaining on a single switch ASIC. In *18th ACM Workshop on Hot Topics in Networks (HotNets)* (2019).

[55] Ye, F., Yu, D., Zhai, E., Liu, H. H., Tian, B., Ye, Q., Wang, C., Wu, X., Guo, T., Jin, C., She, D., Ma, Q., Cheng, B., Xu, H., Zhang, M., Wang, Z., and Fonseca, R. Accuracy, scalability, coverage - A practical configuration verifier on a global WAN. In *ACM SIGCOMM (SIGCOMM)* (2020).

[56] Zhai, E., Chen, A., Piskac, R., Balakrishnan, M., Tian, B., Song, B., and Zhang, H. Check before you change: Preventing correlated failures in service updates. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2020).

APPENDIX

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

A HYPER-CONVERGED DATA PLANE EXAMPLE

In Figure 2, we showed an example of our hyper-converged data plane that contains four different network functions. Different from software-based network functions, the hardware implementation is constrained by the hardware limitation. First of all, the switch uses a pipeline architecture, the packet can only be processed in one direction. This limits the total length of the installed program. Secondly, one switch have multiple pipelines, there are four pipelines in Figure 2. To fit all programs in the switch, we have to program each pipeline individually. Thirdly, due to the pipeline architecture, there are only two ways to redirect the packets: (1) the traffic manager connecting the ingress and egress of all pipelines and (2) the internal loopback that sends the packet from one egress pipeline to its ingress. As a result, the engineers have to carefully allocate the program in each pipeline to make sure the program can fit into the switch without sacrificing performance. For example, given that the packet has to go through the `switch` pipeline first, we have to allocate the first half of the load balancer program to the egress pipeline and second half on the ingress pipeline. Because we have no way to force one packet traverse two ingress pipelines sequentially without touching one egress pipeline. If we install the first half load balancer program in the egress pipeline, the packet has to be recirculated once more, which has additional processing throughput overhead.

In deployment, we rely on the port loopback configuration and table entries to allow different packets traverse different paths. For example, in Figure 2, the ingress of `switch` checks whether the packet should go through the load balancer and sends it to the corresponding egress pipeline. The port loopback function is enabled to fit the complex load balancer program. The `switch` program can also redirect the packet to the scheduler and firewall program based on the table entry installed by the engineers.

B VERIFICATION

B.1 Sequential Encoding with Loops

The sequential encoding algorithm introduced in §4.1 relies on an assumption that the state machine's dependency graph is a DAG, because it is designed in principle based on topological sorting [6]. However, loops may exist in the dependency graph. For example, some headers (*e.g.*, MPLS and VLAN) may contain header stacks and require a transition to itself. Other headers may formulate a complicated dependency graphs with multiple loops—for example,

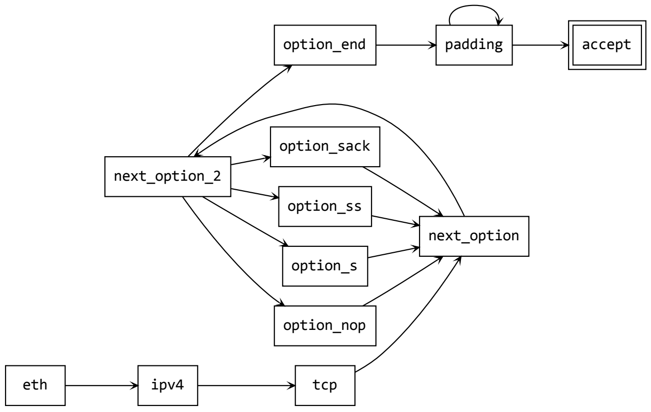


Figure 12: TCP options parsing graph

the TCP option header has four loops, shown in Figure 12. Naively, we can unroll the loops into a DAG because the packet length has an upper bound, but this method introduces too many branches and causes the state explosion problem.

Instead, we have three observations: for one header whose parsing graph has loops, (1) the loops form a strongly connected component (SCC). For example in Figure 12, the four loops form an SCC. (2) The SCC has only one input transition. For example in Figure 12, the only input transition is from state `tcp` to state `next_option`. We name the transition's end state the **root state**. So the `next_option` state is the root state. (3) Each loop has one transition to the root state, in Figure 12, all four loops have one transition to the `next_option` state.

Based on the above observations, we propose the following algorithm to encode the loops into one state. (1) In the SCC, remove all the transitions to the root state, then the SCC is simplified into a DAG. (2) Use sequential encoding to encode the DAG. (3) Enclose the encoded expression with a `while` loop, monitoring the root state's label. (4) Create the outgoing transitions of the encoded state based on the outgoing transitions of the SCC.

For example, in Figure 12, there is one SCC with four loops visiting states `option_nop`, `option_ss`, `option_s`, and `option_sack` respectively. The `next_option` state is the root state. In the first step, the four transitions to the root state is removed. Secondly, the six states in the SCC are encoded into the right half of Figure 13. Next, the encoded statement is enclosed by a `while` loop monitoring variable `Next_option`. Finally, the only outgoing state `option_end`

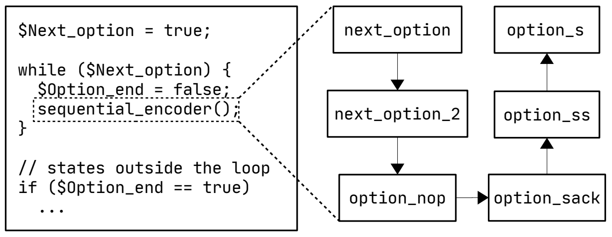

```
$Next_option = true;

while ($Next_option) {
  $Option_end = false;
  sequential_encoder();
}

// states outside the loop
if ($Option_end == true)
  ...
```

Figure 13: Merged state of TCP options

```
state next_option_2 {
  transition select(
     pkt.lookahead<bit<8>>())
  ){
    0: option_end;
    1: option_nop;
    2: option_ss;
    3: option_s;
    4: option_sack;
  }
}
```

```
tmp := tcp$s0e7;
if (tmp == end.kind)
  $option_end := true;
if (tmp == nop.kind)
  $option_nop := true;
...
if ($option_end == true){
  Assume(tcp$s0e7 == end.kind);
  end.$valid := 1;
}
...
```

Figure 14: The lookahead operation in TCP options

is appended to the tail of the merged state. The encoded result is shown in Figure 13.

B.2 Lookahead

Traditionally, the transition from the current parsing state to the next state depends on the value of the parsed field. For example, in Figure 8, the transitions from `Eth` to `IPv4` and `IPv6` depend on the value of `eth.type`, which has already been extracted into the `eth` header. However, the lookahead operation allows the programmer to first check the unparsed bits and then decide which header these bits belong to. For example, in Figure 12, the transition from state `next_option_2` to the next five states depends on the first 8 unparsed bits of the packet, like the left part of Figure 14.

In §4.2, we have introduced the key-value based packet encoding in parsers and deparsers. While it reduces the verification complexity, it also makes lookahead no longer trivial: constraints about the bits looked ahead in previous states should be considered.

Aquila solves this problem by introducing a placeholder with unassigned "havoc" value, like the right part of Figure 14. First, we use the placeholder `tcp$s0e7` to represent the first 8 bits looked ahead. Next, encode all transitions through the placeholder. And finally, encode all following states sequentially. In each following state, an "assume" statement is added to guarantee that the bits looked ahead before (*i.e.*, the placeholder) is equal to the bits extracted here.

For more complex scenarios where a state can be transited from multiple states via individual lookaheads, Aquila introduces a variable recording the previous state, then distinguishes these placeholders via the variable and then generates constraints accordingly.

B.3 Action BitVector (ABV)

We use the following ACL table as an example to introduce the format of ABV:

```
action accept() { ... }
action deny(bit<8> reason) { ... }
action redirect(bit<16> port) { ... }
table ACL {
  key = { ... }
  actions = { accept; deny; redirect; }
  default_action = accept;
}
```

The ABVs of above three actions are represented as:

```
0.1............8...............16..............24
+++++++++++++++++++++++++++++++++++++++++++++++++
|D|...LAID=0....|............PADDING............| accept
+++++++++++++++++++++++++++++++++++++++++++++++++
|D|...LAID=1....|....REASON.....|....PADDING....| deny
+++++++++++++++++++++++++++++++++++++++++++++++++
|D|...LAID=2....|.............PORT..............| redirect
+++++++++++++++++++++++++++++++++++++++++++++++++
```

Each ABV has a fixed `D` field indicating if the current entry is the default entry, and a fixed `LAID` field representing the local action ID. Then the values of action parameters are appended, and a padding field is added to align all ABVs of the table.

Aquila calculates a constant ABV value for each table entry. With ABV, Aquila can represent table matching as:

```
// Without ABV: Thousands of branches and inlining
if (match_0) { accept(); }
else if(match_1) { deny(0); }
else if(match_2) { accept(); }
else if(match_3) { deny(1); }
else if(match_4) { redirect(4); }
... // omit 1k else-if statements
else { accept(); }

// With ABV: Several branches and inlining
abv = ite(match_0, abv_0,
    ite(match_1, abv_1,
    ite(..., // omit 1k nested ite operators
    abv_default)));
if(abv[7:1] == 0) { accept(); }
else if(abv[7:1] == 1) { deny(abv[15:8]); }
else { redirect(abv[23:8]); }
```

Aquila uses ABV to prevent any action from being inlined more than once. Note that the final expression size grows quadratically as the number of branches growing [14]. Aquila in addition connects ABVs by nested **ite** operators, so that branches can be represented in an indirect way, and finally, expression size explosion is avoided. Recall that we further provide an optimized ABV encoding method in Section 4.2.

Another use of ABV is to encode match-related conditions. Aquila uses the `D` field to encode the if-statement such as `if(acl.apply().hit){..}` or `if(acl.apply().miss){..}`, and uses the `LAID` field to encode the switch-statement such as `switch(acl.apply().action_run){..}`.

B.4 More Features

This section details the data structure and new feature encoding and optimization mentioned in §4.3.

Inter-pipeline packet passing. Similar to the input packet encoding, the inter-pipeline packet passing is encoded as a sequence of headers and their values. There are two differences: (1) the header order is generated by the deparser of the previous pipeline rather than given by the engineer: each `emit` statement pushes corresponding headers into the header sequence if they are valid; (2) the unparsed header in the previous pipeline (*e.g.*, when its parser consumes up to layer 3, leaving a packet's tcp header unparsed) is merged with the reassembled header, since the next pipeline may parse deeper than the previous one. The output packet should be encoded as:

$$\texttt{pkt.\$order} := \underbrace{\langle \texttt{eth ipv4 ...} \rangle}_{\text{deparsed headers}} + \text{ unparsed headers}$$

Pipeline behaviors encoding. Pipeline behaviors include mirroring, resubmission, recirculation, *etc.* These behaviors allow packets to traverse through the pipelines multiple times, which easily results in the state explosion. Due to the aforementioned encoding, Aquila can stitch multiple pipelines together, making GCL's complexity grow linearly.

Hash algorithm. Hash algorithm is widely used in data plane programs such as ECMP, bloom filter, *etc.* The non-linear relationship introduced by hash between the input and output slows down the verification performance. Aquila solves this issue by removing this relationship and assuming the output as an independent variable, whose value is "havoc" and bounded by the range of the hash algorithm. This assumption can introduce false positives in rare cases. However, we find it never happened in practice, because our engineers rarely care about the concrete hash value by simply assuming that it is evenly distributed.

Stateful memories. Stateful memories such as registers, meters, and counters are organized as arrays by P4. However, arrays can make verification condition undecidable. Thanks to the inherent constraints of staged-based pipelines, we can safely treat an array as a normal field and ignore the index. Unless the initial values of stateful memories are specified, Aquila leaves them unassigned and considers all kinds of possibilities.

Field groups and quantifiers. Field groups are supported by LPI to describe similar properties in batch over a group of fields, *e.g.*, the TCP 5-tuple. Two quantifiers, *i.e.*, `forall` and `exists`, can be applied to either a fields group defined in LPI or a header defined in P4 code. Aquila translates quantifiers over finite sets into propositional logic to reduce the complexity.

C VALIDATOR IMPLEMENTATION

Building the semantic translator for our validator is non-trivial in reality, since alternative representation, *i.e.*, $X(P)$, may present a totally different semantics from Aquila's GCL representation, *i.e.*, $A(P)$. Based on our implementation experience, the majority of these semantic difference is caused by dynamic values, *e.g.*, header validity and table entries, which are decided and changed in runtime rather than re-deployment stage. We use header validity ambiguity as an example to explain this implementation challenge.

Because the header validity semantics are hard to translate, different representations handle their outputs in quite different ways. For example, Aquila explicitly uses a symbolic a boolean variable, *e.g.*, `$valid`, to denote whether a header is valid; alternatively, Gauntlet [43] embeds the validity in the header's field value directly

We use an example below to illustrate different approaches to represent header validity. Note that we focus on the output state when `ipv4` is not valid.

```
// P4 program
if (ipv4.isValid()) {
  ipv4.ttl = ipv4.ttl - 1;
}

// Gauntlet representation
ipv4.ttl == invalid;

// Aquila representation
ipv4.ttl == @ipv4.ttl;
ipv4.$valid == 0;
```

Given a P4 program, if we only focus on the case that `ipv4` is not valid, as shown in the example above, Gauntlet represents the case as invalid, and Aquila models `ipv4.ttl` with `ipv4.$valid==0` to indicate that all fields in `ipv4` should be ignored.

To match the two semantic representations of the final state, we distinguish whether a header is valid or not. If it is valid, we require that the field computed by Gauntlet and Aquila must be identical to each other. Otherwise, the value computed by Gauntlet must be `invalid` and the `$valid` flag in Aquila must be set to 0. We thus match it with the Aquila semantics as the following post-condition:

$$\texttt{ipv4.\$valid} \Rightarrow \texttt{g == ipv4.ttl},$$
$$\neg\texttt{ipv4.\$valid} \Rightarrow \texttt{g == invalid},$$

where `g` equals

$$\textbf{ite}(\texttt{@ipv4.\$valid, @ipv4.ttl - 1, invalid}).$$