Hydra: Effective Runtime Network Verification

Sundararajan Renganathan
Stanford University
rsundar@stanford.edu

Benny Rubin
Cornell University
bcr57@cornell.edu

Hyojoon Kim
Princeton University
hyojoonk@cs.princeton.edu

Pier Luigi Ventre
Intel
pier.ventre@intel.com

Carmelo Cascone
Intel
carmelo.cascone@intel.com

Daniele Moro
Intel
daniele.moro@intel.com

Charles Chan
Intel
charles.chan@intel.com

Nick McKeown
Stanford University & Intel
nickm@stanford.edu

Nate Foster
Cornell University
jnfoster@cs.cornell.edu

ABSTRACT

It is notoriously difficult to verify that a network is behaving as intended, especially at scale. This paper presents Hydra, a system that uses ideas from runtime verification to check that every packet is correctly processed with respect to a specification in real time. We propose a domain-specific language for writing properties, called Indus, and we develop a compiler that turns properties thus specified into executable P4 code that runs alongside the forwarding code at line rate. To evaluate our approach, we used Indus to model a range of properties, showing that it is expressive enough to capture examples studied in prior work. We also deployed Hydra checkers for validating paths in source routing and for enforcing slice isolation in Aether, an open-source cellular platform. We confirmed a subtle bug in Aether's 5G mobile core that would have been hard to detect using static techniques. We also evaluated the overheads of Hydra on hardware, finding that it does not significantly increase latency and often does not require additional pipeline stages.

CCS CONCEPTS

• **Networks** → **Programmable networks**; **In-network processing**; **Network properties**; • **Software and its engineering** → **Domain specific languages**;

KEYWORDS

Programmable networks, runtime verification, P4.

ACM Reference Format:
Sundararajan Renganathan, Benny Rubin, Hyojoon Kim, Pier Luigi Ventre, Carmelo Cascone, Daniele Moro, Charles Chan, Nick McKeown, and Nate Foster. 2023. Hydra: Effective Runtime Network Verification. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3603269.3604856

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604856

1 INTRODUCTION

At first glance, most packet-switched networks appear simple. Each device implements straightforward tasks like looking up headers in routing tables, filtering packets using access control lists, and adding or removing tunneling headers. But operating a network correctly is a real challenge, especially at scale. Faults, outages, performance degradation, and security breaches occur often in practice, for reasons ranging from simple misconfigurations to pernicious hardware and software bugs. Misconfigurations and bugs can appear anywhere—the control plane or the data plane, fixed-function switches or programmable switches, conventional NICs or smart-NICs, the end host networking stack, and so on.

Prior work has proposed methods and tools to check if a network correctly forwards traffic according to a formal specification. For example, static checkers take snapshots of the network forwarding state (e.g., device configurations or forwarding rules) to build mathematical models of network behavior. These models can be used to verify cloud contracts [9], to answer "what if" questions about router configurations [7, 17], and to verify network-wide properties like connectivity, waypointing, and freedom from loops [4, 27, 28, 30, 38, 51]. Despite enjoying great success, they have well-known limitations regarding scalability [26], the complexity of collecting data plane snapshots [55], and the restriction to stable configurations [7]. Moreover, there is a growing sense (e.g., at Google [2], Facebook [39], and Microsoft [49]) that the success of static checking has shifted the goalposts—the most important failures now often relate to switch hardware and software bugs rather that simple misconfigurations.

Perhaps the most important limitation of static checkers, however, is that they rely on an accurate model of the network. Hence, static checkers can make mistakes if the abstract models they rely upon do not reflect the "ground truth" experienced by packets traveling through the data plane. For example, a static checker might deduce that an end-to-end path exists (based on its model of the forwarding state), but due to bugs in the end host networking stack or some other part of the network that the static checker does not model (e.g., the low-level driver code for the switches) the packet might actually follow a different path. In this situation, ironically, rudimentary tools like `ping` or `traceroute` can successfully detect a bug that the static checker cannot! This modelling limitation exists irrespective of how the static checker builds its model—if any

aspect of the network's behavior is not reflected in the model, then some bugs may go undetected.

In contrast, runtime verification systems can verify the behavior of the network in real time, directly in the data plane. One approach is to send special probe packets and check them against a model [10, 40, 43, 44, 54]. However, this technique only works if the probe packets test all the paths (in the topology and in the code). A second approach is to attach additional information or telemetry data to real data packets, which are collected and analyzed offline at a centralized server [24, 29, 46, 47, 57]. This technique is hard to scale for large or fast networks, because the centralized server quickly becomes the bottleneck.

This paper sets out to answer the following question:

> *Can a network check that every packet is correctly processed, in real-time, against a specification?*

Our Approach. We present *Hydra*, a system that uses ideas from the field of runtime verification [6] and applies them to networking. Rather than analyzing idealized models or performing post-hoc analysis of telemetry, Hydra allows an operator to verify that each packet traversing the network is processed according to a formal specification. Properties are specified in *Indus*, a domain-specific language (DSL) we designed.

Indus is designed to require little to no understanding of the forwarding specification, and operates at a higher level of abstraction. In fact, it reads like typical imperative programming languages that operators are already familiar with. A key distinguishing feature of Indus is that it models network-wide, stateful properties using *telemetry* (comprising packet state, switch-local state, and control-plane state) and *checkers*, which are predicates over telemetry that determine whether a packet should be forwarded, rejected, or reported to the control plane. Indus operates at a higher layer of abstraction than existing DSLs (e.g., P4, eBPF, and DPDK), enabling operators to focus on higher-level behaviors, without concern for how and where they are implemented, or what devices they are compiled to.

Hydra verifies every packet by collecting telemetry data, adding it to packets as they make their way through the network. Indus only requires programmers to specify what telemetry should be collected at each hop and what the predicate on that telemetry should be. By checking each packet, on switch, without the need for a central server, Hydra is inherently scalable and can enforce properties in real time.

Contributions. Our contributions are as follows:

- We present Hydra, the first practical system for checking network-wide properties in real time at line rate (see Section 2).
- We design Indus, a DSL that allows an operator to specify runtime verification policies concisely (see Section 3)
- We develop a compiler for Indus that generates switch-specific checking code that executes independent of the forwarding code (see Section 4).
- We demonstrate that Hydra can find bugs in real-world networks by building a working prototype and using it to implement a form of path validation for source routing, and to detect a subtle bug in Aether [19] (see Section 5).

```
/* Variable declarations */
control dict<bit<8>,bit<8>> tenants;
tele bit<8> tenant;
header bit<8> in_port;
header bit<8> eg_port;

/* Code blocks */
init { /* Executes at first hop */
  tenant = tenants[in_port];
}
telemetry { /* Executes at every hop */ }
checker { /* Executes at the last hop */
  if (tenant != tenants[eg_port]) { reject; }
}
```

Figure 1: Indus program for bare-metal multi-tenancy.

- We assess the expressiveness of Indus from the theoretical and practical perspectives. We show that Indus can express all properties that can be encoded using Linear Temporal Logic over finite traces (LTL_f) (see Section 3). We also develop Indus programs for a range of properties studied previously in the network verification literature (see Section 6).
- We evaluate the overheads of Hydra on Tofino switches [25], finding that the costs of implementing Indus checkers are modest, whether measured in terms of pipeline resources, packet-processing latency, or throughput (see Section 6).

2 HYDRA BY EXAMPLE

In this section, we present a series of examples based on real-world scenarios where there is a need for verification. These examples showcase how Hydra takes runtime verification (RV) ideas and applies them to networking, a hitherto underexplored avenue for said ideas. Each example first describes the real-world scenario, then gives an intuitive description of the property being verified, and then presents a program that expresses the property in Indus.

Bare-metal multi-tenancy. In bare-metal cloud services, tenants have full control over physical servers, including the NIC and host networking stack. To ensure tenant isolation, the Top-of-Rack (ToR) switch is typically programmed with functions such as Virtual Routing and Forwarding (VRF) tables and VXLAN encapsulation [5]. In this setup, all traffic sent and received through a given port connected to a physical server is expected to belong to the same tenant. If any packet crosses between supposedly isolated tenants, the cloud provider risks losing business and trust. The Indus program in Figure 1 enforces network-wide, per-port traffic isolation.

There are two important things to note about this program. First, while Indus is a kind of specification language, programs looks more like a program in a scripting language than a formula in logic. We chose to design a new DSL, rather than re-using an existing logical framework (e.g., Linear Temporal Logic), to avoid the well-known challenges that arise when programmers are asked to write specifications. Second, unlike existing networking DSLs like P4, which captures the functionality of a single switch, Indus models the end-to-end behavior of the network. Hence, it can be used to express network-wide properties—*e.g.*, here, that each packet enters and exits at ports associated with the same tenant.

```
sensor bit<32> left_load = 0;
sensor bit<32> right_load = 0;
control left_port;
control right_port;
control thresh;
control dict<bit<8>,bool> is_uplink;
tele bit<32>[15] left_loads;
tele bit<32>[15] right_loads;
header bit<8> eg_port;

init { }
telemetry {
  if (is_uplink[eg_port]) {
    if (eg_port == left_port) {
      left_load += packet_length;
    }
    elsif (eg_port == right_port) {
      right_load += packet_length;
    }
  }
  left_loads.push(left_load);
  right_loads.push(right_load);
}
checker {
  for (left_load, right_load in left_loads,
       right_loads) {
    if (abs(left_load - right_load) > thresh) {
      report;
    }
  }
}
```

Figure 2: Indus program for data center load balancing.

More formally, an Indus program comprises three blocks. The `init` block executes when a packet enters the network, at the first hop, before it has undergone any other processing. The `telemetry` block executes at every hop, including the first and last hops.[1] The `checker` block executes only at the *last* hop, before the packet exits the network (*e.g.*, in the egress pipeline of the last switch). It executes a predicate on the collected telemetry, which can either come from the `init` or `telemetry` block, to determine whether the packet should be halted ("`reject`"), allowed to proceed, or allowed to proceed but with a report generated ("`report`").

Indus supports several different kinds of variables, each related to how they are used: `tele` variables are carried in the packet, while `control` variables are switch-local state that are managed by the control plane. In this example, the `tenants` control variable is realized as a table that associates switch ports to tenants. The `tenant` telemetry variable records the tenant associated with the original ingress port in the packet. At the last hop, the `checker` block verifies that the ingress and egress ports were associated with the same tenant, and rejects the packet if not.

Load Balancing. For the next example, consider a tiered data center network with servers connected to ToR switches. Data center operators typically spread traffic across multiple paths (e.g., at the granularity of flows [1], flowlets [3] or even individual packets) to balance the load, which reduces congestion. In our example, we will check that the actual usage of the uplink switch ports is approximately balanced, within a given threshold.

[1]In this example the telemetry block is empty but it will be used in other examples.

```
control dict<(<bit<32>,bit<32>),bool> allowed;
tele bool violated = false;

header bit<32> ipv4_src;
header bit<32> ipv4_dst;

init { /* Checks if packet is allowed to enter */
  if (!allowed[(ipv4_src,ipv4_dst)]) {
    violated = true;
  }
}
telemetry { /* Checks if packet on reverse
     direction has been seen */
  if (last_hop && !allowed[(ipv4_dst, ipv4_src)]) {
    report((ipv4_dst,ipv4_src));
  }
}
checker {
  if (violated) { reject; }
}
```

Figure 3: Indus program for stateful firewall.

Figure 2 shows how we can specify load-balancing in Indus in an intuitive manner. To keep the example simple, we focus on load balancing across just two ports (`left_port` and `right_port`), but the program generalizes to any number of ports in a straightforward manner. Note that load balancing is verified on a *per-packet* basis, even if the implementation of load balancing is performed at per-flow granularity. This approach is more scalable than polling each switch for per-port utilization information and then checking whether the load is imbalanced. To implement the desired functionality, the Indus program uses `sensor` variables, which aggregate telemetry data across multiple packets using switch-local state, and a non-trivial `telemetry` block, which records the total amount of data transmitted on each port in `tele` variables. The `checker` block iterates over the telemetry and flags a `report` if it detects an imbalance above a fixed threshold. It is worth noting that the left and right port numbers, as well as the load imbalance threshold are `control` variables. Hence, these values can be changed on the fly, without having to recompile the Indus program.

As shown in this example, we collect telemetry as the packet makes its way through the network in the form of a list, and only perform the check at the last hop. This provides a nice abstraction, similar to that of classical runtime verification, where the Indus program only needs to specify a read-only trace that is collected as the packet flows through the network (telemetry block) and a predicate on that trace (checker block). Enforcing the check at the last hop has the nice property of moving programmability from the core to the edge of the network, where the functionality could be implemented on a smartNIC or even in the kernel. We elaborate on this design decision in Section 4.

Stateful Firewall. Figure 3 is a program to enforce the property that packet flows can only enter the network if a device inside the network initiated the communication. To accomplish this, the control plane installs rules in the reverse direction when it sees a packet in the forward direction. As described earlier, Indus programs are coupled to the network topology, which might mandate that packets enter and leave through a designated choke point.

However, this Indus program is generic enough to check this property in other topologies. For example, every edge switch could be a firewall, instead of all packets going through a choke point. Following standard techniques for ensuring control plane consistency, the control plane could add firewall rules to all edge switches in response to a single report [41].

We use the input packet's contents to check if it is allowed to reach the destination, and we carry this flag in the packet. At each hop, we check if a packet along the reverse direction has been seen (in the `telemetry` block), and if not, we generate a report containing the IP addresses so that the control plane can install the corresponding rules in the `allowed` dictionary. Dictionaries are implemented using P4 tables, as we discuss in Section 4.

In the `init` block, which executes when a packet enters the network at the first hop, if the source and destination IP address tuple is not in the allowed dictionary (added by the control plane), then the packet is marked as violating the firewall rule and will be rejected by the checker. When a packet reaches its last hop, if the source and destination IP addresses are not in the allowed dictionary, Hydra sends a report to the control plane to add it. `last_hop` is a built-in keyword that evaluates true if and only if a packet is at its last hop before it egresses the network.

The three Hydra programs presented in this section are examples of properties that could not be fully verified by a static checker. It is possible to imagine a checker that enforces that the proper tenant isolation rules are installed, or a model checker that ensures that a switch complies with the firewall rule. But this doesn't guarantee correct runtime behavior: For example, a bug in the control-plane might install an incorrect filtering rule; or a bug in the compiler or data plane might not process a packet in the way the static checker assumed. Similarly, hardware faults (memory errors, bit flips on signals, failing connectors) would be undetectable by a static checker. Of course, Hydra programs can have bugs too. But it is less likely that the same bug would appear in the forwarding *and* the checker. This independence between forwarding and checking is key to the value of runtime verification.

3 THE INDUS LANGUAGE

Having introduced some of the main features of Hydra by example, we now give a more precise definition of Indus, a domain-specific language we use to specify network-wide properties. To a first approximation, an Indus program can be thought of as a classical runtime monitor that is attached to each packet traversing the network. The monitor runs alongside the forwarding code in the data plane at line rate. It can observe the behavior of each switch on the network-wide path, maintain state in telemetry variables that are carried along with the packet, and aggregate information across multiple packets using sensors.

3.1 Language Design

Before delving into the details of Indus, it is worth asking: why design a new language? Generally speaking, prior work on runtime verification has followed one of two approaches. The first uses formal logic to specify correctness properties. For example, to stipulate that a packet must not visit switch A twice, we could use the following formula, $\Box\ \neg(A \wedge \bigcirc(\Diamond\ A))$, which is written in Linear Temporal Logic over Finite Traces (LTL_f) [13]. Formally, it says that globally ($\Box$), it is not the case that some event satisfying A (i.e., the packet being at switch A) is followed by ($\bigcirc$) an event where A eventually occurs ($\Diamond$). More intuitively, it says that the packet must not traverse a topological loop involving switch A.

But while formal logic is very well understood, we ultimately elected not to use it as the specification language for Hydra. First, we did not believe that network operators would like or use formal logic. Second, it was not clear how to cleanly accommodate all of the state related to packet-processing—*e.g.*, packet headers and metadata, mutable state on switches, not to mention any new data we might add to support verification [14]. Instead, we followed the second main approach used in runtime verification, relying on a domain-specific instrumentation language (e.g., Eagle [6] or JavaMOP [11]) to specify correctness properties. Here, the programmer writes a program that monitors the execution of the program being verified, using introspection features such as aspect-oriented programming. Ultimately, the program implements a predicate that determines whether the execution should be allowed or not.

The design of Indus is guided by three fundamental principles. First, it provides direct access to all state in the data plane and the control plane that could be relevant to how a packet is processed. To put it another way, the language strives to make it easy to observe network-wide behaviors. Second, the language enforces a strict separation between the variables that track network state, which are read-only, and other variables, which can be read and written. This separation is to ensure that the Indus program does not interfere with the network's forwarding behavior, except at the edge, where it rejects packets that violate the specified property. Third, the language incorporates a number of restrictions to ensure that programs can be compiled to high-speed packet-processing hardware—e.g., all state must be statically allocated and it must be possible to show that all loops terminate.

3.2 Syntax and Semantics

Indus syntax is based on familiar imperative programming constructs (e.g., variables, conditionals, loops, etc.) and it provides a rich set of data types (e.g., bitstrings, booleans, arrays, sets, dictionaries, etc.) and operators (e.g., arithmetic, boolean, and bitwise operations) to express network-wide correctness properties. Figure 4 defines the formal syntax of a core fragment of the language. Our prototype implementation supports a few extensions to this core language, such as `for` loops that iterate over multiple variables, `report` exceptions that carry values, etc. We elide these from our formalization for simplicity. A program p consists of a list of declarations $\overline{d}$ followed by an initialization block, telemetry block, and checking block. Each variable is tagged with a modifier: `tele` variables reside on the packet and aggregate information along the network-wide path; `sensor` variables reside on the switch and aggregate information across multiple packets; `header` variables provide read-only access to data plane variables, such as packet headers and metadata; likewise `control` variables provide read-only access to control-plane state and other configuration information.

The initialization block is executed when the packet first enters the network. Its purpose is to perform computations that cannot

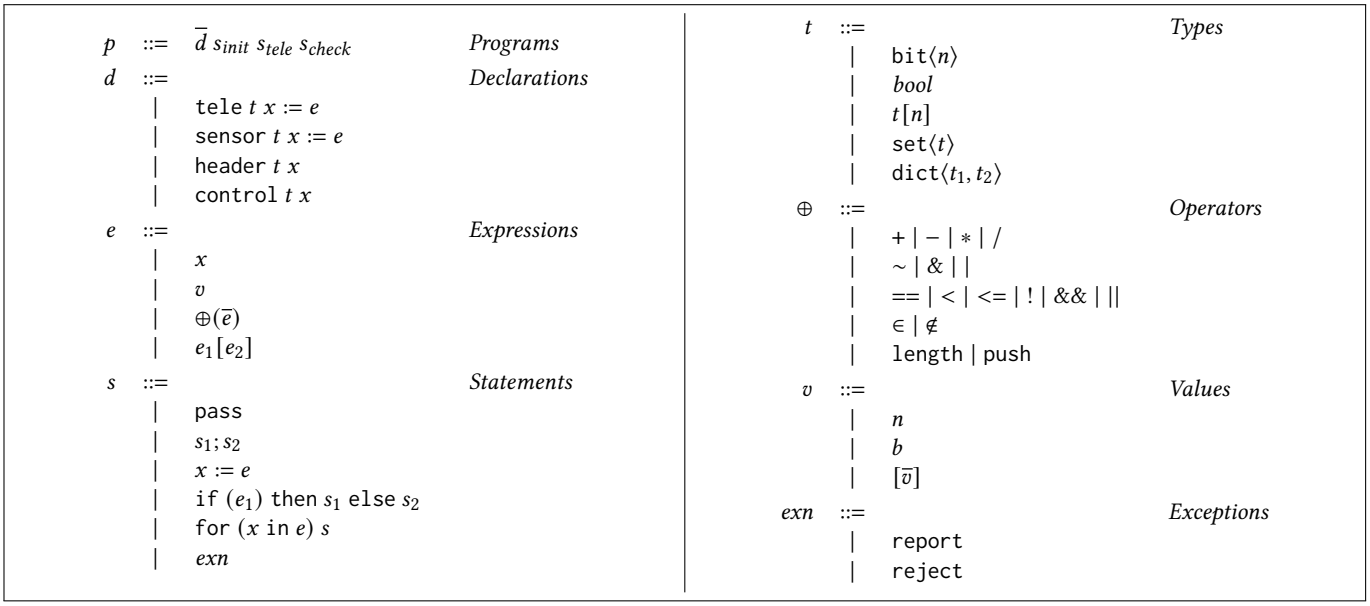


Figure 4: Indus syntax.

be easily encoded using the initializers for variable declarations—e.g., computing a function over multiple control-plane variables. The telemetry block is executed at each hop. Often the telemetry block will push data obtained from header variables into arrays maintained in tele variables, but other approaches are also possible. The telemetry block can also update sensor variables. Finally, the checking block is executed at the last hop. Its main purpose is to decide whether the packet is allowed to exit the network or if it needs to be rejected and/or reported to the management plane.

By design, Indus is strongly typed, which means all operations are checked to ensure that variables are used in ways consistent with their declaration. Types are also important for ensuring termination — e.g., because arrays have a maximum size that is known at compile time, for loops are guaranteed to terminate. As mentioned above, the language also enforces a clear separation between data-plane and control-plane variables, which are read only, and telemetry, sensor, and local variables, which can be read and written. Formally, this ensures that for packets that do not trigger a property violation (i.e., by raising an exception), the final output packet(s) will be identical to the packet(s) that would have been produced had the Indus program not been running at all. To put it another way, Indus does not interfere with the execution of packets that satisfy the property, only those that violate it. Similarly, the telemetry, sensor, and local variables, which are used to implement the checking logic, are kept separate from the other variables. Hence, the network cannot subvert the property being enforced simply by injecting certain packets into the network or issuing control-plane commands. Indus can be used to verify that a network is free of infinite forwarding loops, but the overhead is non-trivial—one must either enforce a maximum length on forwarding paths, or keep track of the packet's path and periodically check for duplicates. Moreover, because loops are almost always undesirable, many networks already offer robust mechanisms for avoiding them—e.g., IPv4's time-to-live (TTL) field.

$$\varphi ::= A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1\ \mathcal{U}\ \varphi_2$$

$$\begin{aligned}
[\![A]\!]_x &\triangleq A(x) \\
[\![\neg\varphi]\!]_x &\triangleq \neg[\![\varphi]\!]_x \\
[\![\varphi_1 \wedge \varphi_2]\!]_x &\triangleq [\![\varphi_1]\!]_x \wedge [\![\varphi_2]\!]_x \\
[\![\bigcirc\varphi]\!]_x &\triangleq \exists y.\, succ(x, y) \wedge [\![\varphi]\!]_y \\
[\![\varphi_1\ \mathcal{U}\ \varphi_2]\!]_x &\triangleq \exists y.\, x \le y \wedge y \le last \wedge [\![\varphi_2]\!]_y \wedge \\
&\qquad \forall z.\, x \le z \wedge z < y \Rightarrow [\![\varphi_1]\!]_z
\end{aligned}$$

Figure 5: LTL$_f$ syntax (top) and encoding into first-order logic (bottom) [13].

Hence, in examples, we will often elide the additional logic that would be needed to encode loop freedom in Indus.

3.3 Expressiveness

Having defined Indus, it is natural to wonder about the class of properties that it can capture. Generally speaking, questions about expressiveness are settled by giving translations that map programs from one language into another—e.g., this is how we show that Turing machines and λ-calculus capture the same class of computations. We are not aware of any logic or existing language that precisely captures the set of properties that can be expressed using an Indus program. Among other things, the presence of header and control variables, which operate as a kind of "foreign function interface" to the data plane and the control plane, as well as sensor variables, make the relationship difficult to state. Nevertheless, to establish a lower bound, we prove here that Indus is rich enough to express all LTL$_f$ formulas. Along the way, we also show that it also corresponds to first-order logic formulas over finite traces.

Recall that LTL$_f$ can be understood as defining predicates on traces. Each trace is made up of an ordered sequence of events,

which are assumed to be finite. Figure 5 gives the formal syntax of LTL_f. Formulas A correspond to atomic predicates that either hold or do not hold at a given event. For instance, atomic predicates could keep track of the location of the packet in the network, or the value of the destination address in the IPv4 header. Formulas $\neg\varphi$ and $\varphi_1 \wedge \varphi_2$ correspond to logical negation and conjunction respectively. Formulas $\bigcirc\varphi$ state that φ holds in the *next* event—i.e., the one that follows the current event in the ordered sequence. Finally, formulas $\varphi_1 \ \mathcal{U} \ \varphi_2$ state that φ_1 holds at all events *until* some point at which φ_2 holds. As usual, other formulas can be encoded. For example both $\Box\varphi$, which states that φ always holds, and $\Diamond\varphi$, which states that φ eventually holds, can be encoded using the until operator.

In their original paper on LTL_f, De Giacamo and Vardi proved that formulas can be translated to first-order logic [13] over finite sequences. The bottom half of Figure 5 gives the translation, which is parameterized on a variable x corresponding to an index in the sequence, initially the index of the first element. Hence, to prove that TPC can express the same set of properties as LTL_f, we simply have to show that it can model the semantics of these first-order formulas. Assume that the telemetry block populates an array T with an increasing sequence of integers as well as arrays A corresponding to the atomic predicates occurring in the program. With this encoding, it is straightforward to show that the semantics of every first-order formula used in the translation of LTL_f can be expressed in Indus. For example, existential formulas $\exists x.\, P$ map to a `for` loop:

$$
\begin{array}{l}
\texttt{bool } r_0 := false; \\
\texttt{for } (i \texttt{ in } T)\{ \\
\quad x := i; \\
\quad (\!|P|\!)_{r_0}; \\
\quad r := r \ || \ r_0; \\
\}
\end{array}
$$

Here $(\!|P|\!)_{r0}$ denotes the translation of P using an auxiliary variable r_0 to store the result. For the complete formalization, please see the long version of this paper, which is available online [42].

Theorem 3.1 (Expressiveness). *Let φ be an LTL_f formula, π a trace, $\mathcal{I}$ a corresponding first-order interpretation, and σ the corresponding Indus store. Also let $P = [\![\varphi]\!]_x[x \mapsto 1]$ and $s = (\!|P|\!)_r$. The following are equivalent.*

- $\pi, i \models \varphi$
- $\mathcal{I} \models P$
- $\langle \sigma, s \rangle \Downarrow \langle \sigma', s' \rangle$ *and* $\sigma'(r) = \texttt{true}$.

Proof. The first two cases were given by De Giacamo and Vardi [13]; the third case follows by induction. □

Corollary 3.2. *Every network-wide property that can be expressed in LTL_f can be expressed in Indus.*

Overall, this result shows that Indus is at least as expressive as the specification language used in many other runtime verification systems, modulo the choice of atomic predicates A.

4 THE INDUS COMPILER

This section presents our compiler, which converts Indus programs to P4 code, which can then be linked with the forwarding code. Our compiler is designed to make it easy to ensure that the state and control-flow of the Indus program are not tampered with during this process. Our compiler is written in approximately 2500 lines of OCaml code. Our current prototype only supports P4, but we envision possible extensions that target other DSLs like eBPF and DPDK in the future.

```
// Hydra Headers
struct hydra_header_t {
  eth_type2_t hydra_eth_type;
  bit<8> tenant;
}
struct hydra_metadata_t {
  bit<8> tenant;
  bool reject0;
}
// Generated Init Code
apply {
  ...
  // look up ingress port tenant
  tenants_in_port.apply();
  // initialize tele variable
  hydra_header.tenant = hydra_metadata.tenant;
}
...
// Generated Checker Code
apply {
  // lookup output port
  tenants_eg_port.apply();
  if (hydra_header.tenant!=hydra_metadata.tenant) {
    // reject if ingress and egress disagree
    hydra_metadata.reject0 = true;
  }
  strip_telemetry(); // strip telemetry at last hop
}
```

Figure 6: Generated `tna` code for bare-metal multitenancy.

4.1 Code Generation

The compiler takes as inputs an Indus program and a topology file in which each switch is classified as an edge or non-edge switch. The compiler then generates switch-specific code for each switch in the topology.

The front-end of the compiler first lexes and parses the Indus program into an abstract syntax tree. Next, the type checker ensures programs are well typed and respect constraints such as read-only access to `control` and `header variables`. The type checker also constructs a symbol table for the declarations in the Indus program, which is used in the construction of the P4 headers and parsers. Finally, the compiler generates P4 code for each Indus construct. Many of the abstractions found in Indus can be directly mapped onto analogous constructs in P4—e.g., assignments, conditionals, etc. But for some other abstractions, it is not obvious how to implement them in P4. The following list summarizes the strategies used to generate code in the compiler:

- `header` variables: A header variable declaration requires an annotation (indicated with an `@`) that specifies the corresponding name in the forwarding program that tells the compiler how to translate references to the variable. For example, if an Indus program needs to refer to the source

IP address through the `ip_src` variable, the required annotation is `hdr.ipv4.src_addr`. In examples, we omit these annotations for brevity.

- `tele` variables: A telemetry variable declaration leads to an extra field in a special telemetry header generated by the compiler. The `tele` variables travel with the packet as telemetry and are serialized and deserialized using parsers and deparsers generated by the compiler.
- `sensor` variables: A sensor variable declaration is implemented as a P4 register. Reads and writes to sensor variables are translated, provided the underlying target (e.g., BMv2, Tofino) supports them.
- `control` variables: A control variable declaration is mapped to a match-action table. There are two different types of control variables: a non-dictionary control variable and a dictionary control variable. A non-dictionary control variable is statically defined by the control plane, and can be initialized by a default action in a single match-action table that executes at the start of the pipeline. On the other hand, a dictionary control variable requires more complex lookups. To ensure the lookup operation returns the most up-to-date value for each dictionary control variable, our compiler creates and places a match-action table right before the statement that contains the lookup in the translated P4 code.
- Lists and loop operations: Lists are implemented as header stacks in P4, which have the semantics of a fixed length array. P4 does not support loops. Thus, our compiler *unrolls* Indus's `for` loops into sequential code: the loop body is executed for each list index that is valid. Our compiler also supports the `in` operator, which translates into an expression that tests if the left-hand side is equal to any valid elements of the header stack specified on the right-hand side.

At the final hop, before a packet exists the network, we strip the checking headers produced by the Indus program. This ensures conformance with software running on end hosts that do not recognize the extra headers injected by Indus. To this end, the control plane needs to specify the set of edge ports in the network to the compiler. Then the compiler generates an extra match-action table that matches on the egress port and strips the headers for packets that are sent to these egress ports. A similar process is done for injecting Indus-generated headers to packets at the first hop. In principle, we could delegate these "last-hop" and "first-hop" tasks to the NIC at end hosts. We leave this extension to future work.

4.2 Linking

Figure 6 shows the generated P4 code for the bare-metal multi-tenancy example described in Section 2. The final compilation step is to link the generated headers and parsers blocks as well as the `init`, `telemetry`, and `checker` code blocks with the forwarding code for the switch, which we assume is also written in P4. Specifically, the `init` block must be placed at the beginning of the ingress pipeline on first-hop switches, the `telemetry` block is placed at the egress pipeline on every switch, and the `checker` block is placed at the end of the egress pipeline on last-hop switches. Since networks are bidirectional, the edge switches in the network end up running all three code blocks, while the non-edge switches only run the `telemetry` block. Automatically linking our compiler output blocks with the forwarding P4 program is future work.

```
control bool is_spine_switch;
tele bool visited_spine;
tele bool to_reject;

init {
  visited_spine = false;
  to_reject = false;
}
telemetry {
  if (is_spine_switch) {
    if (visited_spine) {
        to_reject = true;
    }
    visited_spine = true;
  }
}
checker {
  if (to_reject) {
    reject;
  }
}
```

Figure 7: Valley-free routing in Indus.

4.3 Last-Hop vs. Per-Hop Checking

Our current compiler compiles Indus programs to the network so that a switch at every hop collects telemetry but the check only runs at the last hop, or edge, switch. This approach has a number of advantages. First, it saves resource usage on non-edge switches since running a check at a switch requires additional computation. This approach is also more amenable to incremental deployment since Hydra can still run with switches that are not fully programmable but can run telemetry and attach information to packets. Another approach, however, is to execute checks at every hop. The main advantages of this approach are that it often requires less telemetry data, and packets that violate the given property can rejected (or reported) at any switch, not just at the edge. We plan to implement this approach in the future, using our compiler to automatically relocate checks from the edge and into the network core.

5 HYDRA CASE STUDIES

This section presents a pair of case studies that demonstrate the practical utility of using Hydra for enforcing network-wide properties using runtime verification. The first case study develops an application of Hydra to implement path validation in a data center network with source routing, ensuring that packets follow "valley-free" paths. The second case study illustrates a use of Hydra to detect a subtle bug in Aether's implementation of application filtering, which provides a form of slicing.

5.1 Example 1: Valley-Free Source Routing

Recall that in source routing, the sender specifies the path the packet should take through the network. In its purest form, the path is specified as a list of hops, and each switch simply pops

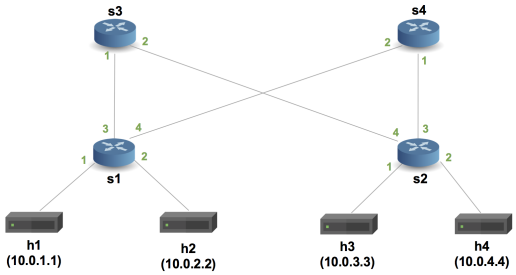


Figure 8: Simple Leaf-Spine Topology

the stack and forwards the packet accordingly. Source routing has many advantages—e.g., it eliminates the need for large routing tables and complex routing protocols, since senders are responsible for computing paths. One downside, however, is that source routing does not offer operators the same degree of control as traditional, destination-based forwarding schemes. With Hydra, operators can specify and enforce policies that restrict the set of legal paths when using source routing; any packet that attempts to follow an illegal path will be automatically dropped. For example, an important property in data center routing is that paths are valley-free, preventing an explosion of suboptimal paths in a fat-tree topology. In particular, packets may not traverse an link that goes "up" in the topology after they have already traversed a "down" link.

Indus checker for source routing. Figure 8 depicts the topology of the simple network we instrumented with Hydra, generalizing code found in the P4 Tutorial [12]. The network contains a leaf-spine topology with four switches. All the switches run the same P4 program, which implements a simple source routing scheme, and we link the program with the valley-free routing checker written with Indus, shown in Figure 7. While it is possible to write a general Indus program to check valley-free routing for any given fat-tree topology, we leverage the fact that Indus is topology-specific to write an efficient program that only requires a single control variable and two bits of telemetry to ensure that a spine switch is visited at most once. This program consists of a simple state machine that checks if the current switch is a spine switch and marks the packet to be dropped if it has already visited a spine switch. Note that the Indus program is independent of the forwarding P4 code: it could operate on any routing protocol. And while the forwarding program operates on egress ports, the Indus program operates at a higher level, using switch-specific control plane state.

Bug caught by Hydra. In this case study, we artificially injected a bug into the script used by the sender to add extra invalid hops to the source route. Using Mininet [34], we generated a number of paths and verified that Hydra allowed all possible valley free paths between hosts and successfully dropped any packets that followed errant paths due to the bug in the sender script.

5.2 Example 2: Application Filtering in Aether

Aether [19] is an open-source edge computing platform that offers private LTE/5G connectivity. Figure 10 shows an Aether edge deployment with three main elements: (1) small cells that provide LTE or 5G access to mobile clients such as cameras, sensors, or phones; (2) servers that run edge-applications exposing low-RTT services to

```
tele bit<32> ue_ipv4_addr;
tele bit<32> app_ipv4_addr;
tele bit<8> app_ip_proto;
tele bit<16> app_l4_port;
tele bit<8> filtering_action = 0; // 1=deny,2=allow
control dict<(bit<32>,bit<8>,bit<32>,bit<16),bit
     <8>> filtering_actions;

header bit<32> inner_ipv4_src;
/* ... Header variable declarations ... */
header bit<16> outer_udp_dport;

init {
  if (inner_ipv4_is_valid) {
  // this is an uplink packet
    ue_ipv4_addr = inner_ipv4_src;
    app_ip_proto = inner_ipv4_proto;
    app_ipv4_addr = inner_ipv4_dst;
    if (inner_tcp_is_valid) {
      app_l4_port = inner_tcp_dport;
    } elsif (inner_udp_is_valid) {
      app_l4_port = inner_udp_dport;
    }
  } elsif (ipv4_is_valid) {
  // this is a downlink packet
    ue_ipv4_addr = outer_ipv4_dst;
    app_ip_proto = outer_ipv4_proto;
    app_ipv4_addr = outer_ipv4_src;
    if (tcp_is_valid) {
      app_l4_port = outer_tcp_sport;
    } elsif (udp_is_valid) {
      app_l4_port = outer_udp_sport;
    }
  }
  filtering_action = filtering_actions[(
       ue_ipv4_addr, app_ip_proto, app_ipv4_addr,
       app_l4_port)];
}
telemetry {}
checker {
  if (filtering_action == 1 && !to_be_dropped) {
    reject; report((ue_ipv4_addr, app_ip_proto,
         app_ipv4_addr, app_l4_port,
         filtering_action));
  }
  if (filtering_action == 2 && to_be_dropped) {
    report((ue_ipv4_addr, app_ip_proto,
         app_ipv4_addr, app_l4_port,
         filtering_action));
  }
}
```

Figure 9: Aether application filtering in Indus.

mobile clients; and (3) an SDN fabric of P4-programmable switches that connects small cells to servers and the Internet [20].

The Aether software stack includes an operator-facing portal and API for system configuration, a 3GPP-compliant dual-mode 4G/5G mobile core, and ONOS, a distributed SDN controller responsible for controlling the fabric switches. The fabric provides L3 connectivity by routing IPv4 packets over the spine switches using Equal Cost Multi-Path (ECMP) forwarding. It supports L2 bridging and VLAN isolation within a rack, and other common features such as rerouting in case of failures, learning/advertising routes via BGP, configuring static-routes, DHCP relay, multicast, and ACLs for filtering. A notable feature in Aether is that the switches help implement the mobile core User Plane Function (UPF) [37] (i.e., with support

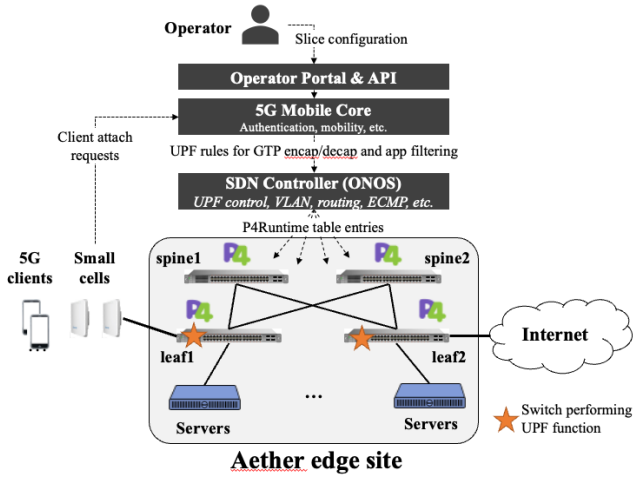


Figure 10: Aether architecture and topology.

for GTP-tunnel encapsulation/decapsulation, downlink buffering, accounting, QoS, application-filtering, and slicing).

Aether application filtering. We implemented a wide range of Hydra checkers for Aether (see Section 6, Table 1), but we focus here on UPF application filtering, which had a subtle bug we detected using Hydra. Application filtering allows operators to create slices that connect an isolated group of clients and give them with bandwidth guarantees. Operators can define filtering rules allowing clients in a slice to access some edge-applications while denying access to others. Internet access is considered an application, and applications can be shared across slices. For example, mobile clients belonging to the camera-slice are allowed to communicate with an edge application that analyzes video, but cannot access the Internet. Mobile clients in the phone-slice have the opposite permissions.

Each slice has a prioritized list of filtering rules of the form:

`priority : ip-prefix : ip-proto : l4-port : action`

where `ip-prefix`, `ip-port`, and `l4-port` identify the application. The `action` can be `allow` or `deny`, and the priority is used to disambiguate in case of overlapping rules. For example, to deny all traffic by default but allow access to applications using UDP port 81, the operator could use the following rules:

- `20 : 0.0.0.0/0 : UDP : 81 : allow`
- `10 : 0.0.0.0/0 : any : any : deny`

Now, to integrate with any 3GPP-compliant mobile core, Aether's ONOS controller uses a standard 3GPP interface named PFCP. This interface does not allow to specify application filtering rules globally for a slice. Instead, rules are sent to ONOS on a per-client basis. This means that ONOS receives the same application filtering rules for every client that connects to the network. Thus, in each slice, there are a set of clients (identified by their IMSI, a unique number associated with a SIM card) and a list of application filtering rules for each client. When a new client connects to Aether, the mobile core looks up the slice configuration for the given IMSI and installs the user plane rules on switches to terminate the GTP tunnels and enforce application filtering. The P4 program running on the switch optimizes ASIC resources by splitting UPF processing across

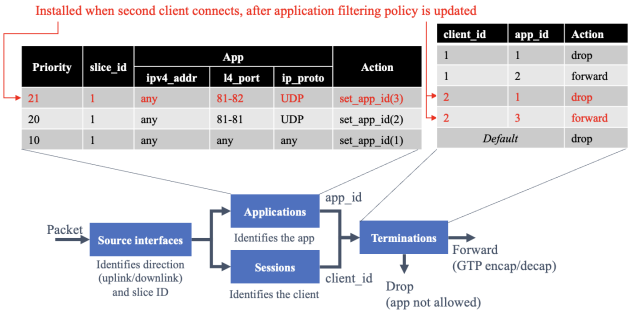


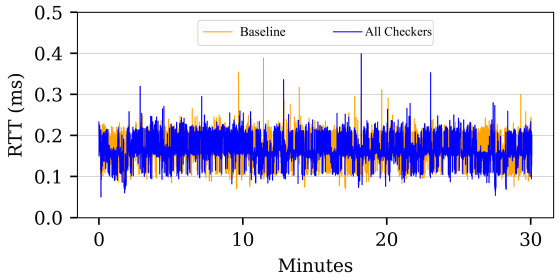
Figure 11: P4 tables demonstrating application filtering bug

different types of tables, and ONOS is responsible for translating UPF rules into multiple table entries and updating the entries in each leaf switch. Hence, while the slice and application filtering configuration is conceptually simple, ensuring the correctness of the filtering depends on the interaction of multiple software and hardware components, each of which could be subject to different bugs. Bugs and errors could result in the installation of erroneous entries, which may cause traffic to violate the intended policy.

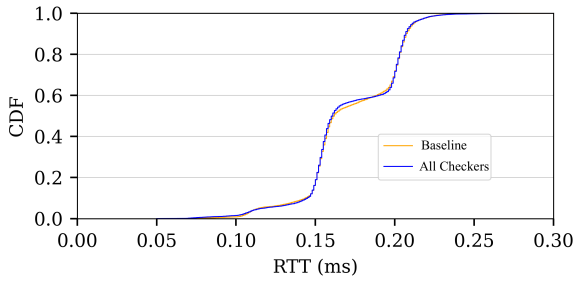
Hydra checker for application filtering. Figure 9 shows the Indus program to verify application filtering. The `init` block first determines the direction of the packet and then fetches the fields of interest into `tele` variables, which it then uses to look up a `control` variable to know the filtering action. The filtering action is carried on the packet (in addition to the packet fields used in the lookup). A simple control plane application that runs atop ONOS as part of the rest of the deployment configures the `control` dictionary variable. At a high level, it receives the application filtering rules from the operator at startup, listens for attach requests from mobile clients, and installs the corresponding entries in the table for the `filtering_actions` variable.

Bug caught by Hydra. We now describe a known bug in Aether that causes traffic to be dropped when updating application filtering. Figure 11 provides a simplified representation of the multiple P4 tables used to realize the UPF function. To reduce memory utilization (in particular, of TCAM), the *Applications* table is designed so that entries can be shared by multiple clients of the same slice. This table determines the application for a packet by matching on the IPv4 and L4 port headers, and sets the appropriate app ID metadata for the packet. The *Terminations* table then uses the app and client IDs together to determine whether to forward or drop the packet. This design requires ONOS to correctly manage shared application entries when clients connect to the network or when rules are updated in the operator portal.

Figure 11 shows a scenario where a specific slice is first configured with filtering rules that deny all traffic by default (app ID 1) but allow traffic for apps on UDP port 81 (app ID 2), which has a higher priority. When client ID 1 connects, two rules are installed: the default drop rule for client ID 1 with app ID 1, and the allow rule for client ID 1 with app ID 2. Thus, client ID 1 can successfully access applications on UDP port 81. Let's say the operator later updates the filtering rules in the portal by expanding the UDP port range to 81-82 and increasing the priority of that rule, and set the



(a) RTT comparison over time



(b) RTT CDF comparison

Figure 12: Performance overhead of Hydra

app ID as 3. When client ID 2 connects, the mobile core installs client-specific rules with this updated policy, thus ONOS installs a new table entry with range 81-82 in the *Applications* table. Due to the new higher-priority entry for app ID 3, packets from client ID 1 with UDP port 81 will now get an app ID 3 assigned by the *Applications* table. As a result, traffic for client ID 1 on port 81 that was previously allowed is now dropped since the client-app ID pair does not exist in the *Terminations* table. This subtle bug is hard to catch and even harder to pinpoint the exact location where the packets are being dropped.

With the checker compiled from Figure 9, Hydra detects that client ID 1's packets with UDP port 81 are actually to be dropped when it should have been allowed. With the `report` action, such behavior is explicitly reported to the control plane by the switch where the inconsistency was detected.

6 EVALUATION

To further evaluate our design, we wrote more checkers for verifying a range of properties in the Aether testbed, including examples studied previously in the network verification literature. We assess the expressiveness of Indus and and overheads of our Hydra system. Table 1 summarizes our results.

6.1 Expressiveness and Conciseness

In Table 1, we show the number of lines of Indus code required to specify each property and the number of lines of P4 code generated by our compiler. Indus enables expressing properties succinctly, typically requiring an order of magnitude less code compared to the direct implementation in P4. We optimized the programs to streamline their compilation to hardware. For example, in the Indus checker for detecting load imbalance, we maintain a boolean variable that records whether an imbalance has been detected on any switch on the network-wide path, which eliminates the need to iterate over multiple arrays in the block. Overall, our evaluation shows that Indus, our domain-specific language, can express a wide range of practical network properties in a concise manner.

6.2 Resource and Performance Overheads

Next, we discuss the overheads associated with deploying Hydra checkers on Intel Tofino switches.

Resource Overhead. The main resources on Tofino switches that are relevant to Hydra are the number of pipeline stages used and the amount of Packet Header Vector (PHV) bits used. Other stage resources (e.g., SRAMs, TCAMs, etc.) are also important, but their contribution is implicitly accounted for in the usage of pipeline stages. We first measure the resource utilization of the baseline forwarding program that runs in the Aether mobile core and then measure the resource utilization for each of the implemented properties when linked with this program.

The baseline program is already at 12 stages. In general, deploying a Hydra checker will require extra resources. However, in this instance, each of the checkers can be executed in parallel alongside the base program and they do not increase the number of stages when linked with the base program. This parallel execution is made possible by the independence between the forwarding and checking code. We can see that the overhead on PHV resources is relatively modest, with higher usage for the programs that collect more telemetry. For instance, the properties that require the most PHV are source routing path validation and application filtering. The former carries a significant chunk of telemetry per hop while the latter collects all its telemetry in the `init` codeblock. The PHV resource usage increases from 44.53% to 52.14% with the application filtering checker on, a 7.6% difference.

Performance Overhead: Setup. Next, we evaluate if Hydra introduces any performance overhead when deployed in practice with our Aether testbed. We confirmed that mobile devices connected to the cellular network had stable Internet connectivity even when Hydra checkers were running. However, although Aether processes real-world traffic, the data rates in our testbed are currently not high enough to fully evaluate Hydra's performance limits. Thus, for this evaluation, we tapped and mirrored network traffic from a production campus network and replayed it towards `leaf1` in Figure 10. As illustrated in Figure 13, we utilize an existing P4Campus infrastructure [31] at Princeton University. The campus network has network Test Access Point (TAP) devices installed at several vantage points in the network, which create a mirror of the traffic they see on links. We tap two /16 campus network subnets at our border routers and send the mirrored traffic to a P4-based packet anonymizer [32], which runs on a programmable switch. This P4 program hashes personally identifiable information like MAC and IP addresses in a prefix-preserving manner at line rate and delivers the anonymized traffic to the cellular network. The resulting anonymized packet trace's load is around 350K packets per second. *Ethical considerations*: All packet traces were inspected and sanitized by a campus network operator. Personal data, like MAC and

Property Name	Description	LoC		Tofino Overhead	
		Indus	P4 Output	Stages	PHV (%)
Baseline	Aether P4 program compiled in `fabric-upf` profile	-	-	12	44.53
Multi-Tenancy	All traffic through a given ToR switch port, facing a bare-metal server should belong to the same tenant	14	102	11	48.44
Datacenter uplink load balance	Uplink ports in data center switches should load balance, to exact equivalence, between specified ports	37	194	12	48.83
Stateful firewall	Flows can only enter the network if a device inside initiated the communication	23	164	12	49.21
Application filtering	Clients should only be able to communicate with designated applications (as identified by layer 4 ports)	64	126	12	52.14
VLAN isolation	Packets should traverse switches in the same VLAN	21	119	11	47.85
Egress port validity	Packets should only egress a switch at allowed ports	18	132	12	46.09
Routing validity	The first and last hop of any packet should be a leaf switch, while the rest of the hops are spine switches	21	122	12	46.09
Loops (4 hops)	Packets should not visit the same switch twice	20	156	12	48.24
Waypointing	All packets should pass through a choke point	22	154	12	47.85
Service chains	Packets from switch s to switch t should pass through switches $(w_1, w_2, ..., w_n)$ in that order on the way	26	121	12	47.26
Source routing with path validation	A packet that is source routed through switches $(s, s_1, ..., t)$ should pass them in order	34	211	12	51.56

Table 1: Hydra properties.

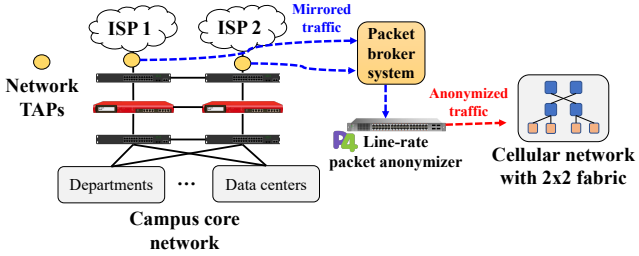


Figure 13: TAP architecture for mirroring campus network traffic to Aether.

IP addresses, were removed or hashed before being accessed by researchers. Addresses were anonymized in a consistent manner using a one-way hash with a salt, and payloads are discarded. Our research was discussed and approved by Princeton's Institutional Review Board (IRB).

Performance Overhead: Result. Next, we evaluate if Hydra adds noticeable performance overhead due to its parsing and checking logic. We perform a microbenchmark with and without Hydra enabled and compare the two. Our throughput comparison with and without Hydra were almost identical with around 20 Gb/s. However, we were not able to push near to the throughput limit of the hardware switches, which is is 6.5 TB/s. Thus, we focused our performance evaluation to measuring Hydra's overhead on packet-processing latency. First, we generate bidirectional UDP traffic at 10 Gb/s on our cellular access network testbed using `iperf3`. The background traffic utilizes all links between the two spine and two leaf switches with ECMP routing. Then, we started a fast ping (every 0.2 s) from one server attached to the `leaf1` switch to another service attached to the `leaf2` switch. Figure 12a shows the round-trip times (RTTs) during the experiment. There appears to be no significant difference between the baseline and with all checkers enabled. To further evaluate the latency overhead statistically, we plotted the cumulative distribution function of our RTT measurements in Figure 12b, and also performed a t-test [23]. Both results confirm that there is no statistical latency difference between the baseline and with all checkers on.

7 RELATED WORK

Our work adds to the growing literature on network verification. Here, we summarize the most relevant pieces of prior work, grouped in several topical areas.

Runtime Verification. In runtime verification (RV), a system is instrumented to send events about its execution to a monitor. While the system executes, the monitor verifies the behavior against a specification. When a behavior violation occurs, the monitor sends feedback to correct the behavior or halt the execution. Over time, researchers have created more expressive languages to specify properties in runtime verification systems. Work such as Eagle [6] helped popularize the use of Linear Temporal Logic to specify properties in the monitor. Eagle's powerful logic can express a diverse set of runtime behaviors, such as requiring each request in an application to have a corresponding response or limiting the size of a queue. Eagle also provides significant flexibility in what it monitors: any system can be instrumented to send a log of events to Eagle as the structure and content of the logs are user defined.

Static Verification for Networks. There is also a large body of work on static verification for networks. Early work by Xie et al. [50] proposed using static techniques to reason about reachability in IP networks. It proposed the now-standard approach of computing the transitive closure of transfer functions that model the behavior of individual devices and links. Header Space Analysis [28], Anteater [38], and Veriflow [30] emerged later, and applied this general approach in the context SDN. To improve the scalability of

their analyses, they developed optimized data structures for transfer functions. Atomic predicates [52] replaces complex classifiers with simple predicates that can be handled efficiently in backend solvers. NetKAT [4] is an algebraic framework based on a sound and complete deductive system and a decision procedure based on automata [18]. Tools like `p4v` [35], Acquila [48], and Network Optimized Datalog (NoD) [36] translate P4 code into representations that can be verified using staic techniques. Vera [45] and P4-Assert [21] address the same problems using symbolic execution, while `bf4` [15] infers control-plane constraints automatically. Gravel [56] formally verifies the software of middle boxes such as NATs and firewalls. A complementary line of work focuses on control plane verification. Batfish [17] and Minesweeper [7] statically analyze configuration files for distributed protocols to verify reachability properties automatically. Recent approaches use abstract interpretation to verify simpler representations of programs [8, 16, 22]. Our work builds on the extensive foundation provided by this prior work, but uses an approach based on runtime rather than static techniques.

Runtime Verification for Networks. Early work on runtime checking for networks focused on generating test or probe packets [10, 40, 54]. P4Consist proposes adding a new module to tag packets with the path and forwarding rules used to process them [44]. However, P4Consist only adds these tags to special probe packets, which are generated using a traffic generator, and verification is performed out-of-band—i.e., the tagged packets are sent to a separate server for analysis. VeriDP follows a similar approach, but focuses on detecting inconsistencies between control-plane and data-plane state [57]. In contrast, Hydra instead uses checkers that execute directly in the data plane, allowing it to detect violations as they occur and halt erroneous packets. Offline analysis approaches also face inherent scalability issues or accuracy tradeoffs due to sampling. Hydra does not rely on sampling—it performs runtime checking on every packet. DBVal verifies assertions at runtime by instrumenting the data plane [33]. However, their checks are tied to how forwarding is implemented. Thus, the verification code and system being instrumented are not independent, which could lead to false negatives if forwarding and checking code have the same bug. Aragog [53] supports defining properties parameterized on location, stateful variables, and temporal predicates. Additionally, it checks every execution trace in the system. Aragog differs from Hydra in that it focuses on distributed network functions, rather than the data plane itself. In that sense, it is a complementary approach to Hydra. Aragog requires making modest modifications to the source code for the network functions, in order to send events of interest to the verifier. Hydra checks every packet and is independent of the forwarding code.

Summary. Overall, Hydra builds on ideas that have been developed for years in the runtime verification and formal methods communities and applies them to the problem of verifying network behavior. It provides an easy-to-use specification language for expressing a rich set of network-wide properties as well as a compiler that translates these programs into executable code that can be deployed on network switches. Our approach is expressive, scalable, and operates in-band, detecting and blocking errant packets at line rate and in real time.

8 CONCLUSION

There is an important difference between catching a packet on the wrong path *immediately* versus catching it *eventually*. If an intruder is exfiltrating confidential data, one packet may be all it takes; if a single packet passes between two "isolated" virtual tenants, trust (and business) is lost. Our approach is to check *every* packet as it flows through the network. While it is perhaps an extreme approach, we think it is essential if we are to automate the closed loop control of networks and minimize human intervention. Hydra programs are easy to read and write for a large set of expressive properties. Our experiences deploying Hydra programs on P4 switches and in the context of an open-source cellular access network with real-world traffic shows they create little overhead, yet can catch real bugs in a live system.

ACKNOWLEDGMENTS

We are grateful to Megan Jung, Bruce Spang, Owolabi Leunsen, and the participants of the Bellairs Workshop on Network Verification for early ideas and helpful feedback. This work is supported by the NSF under grant FMiTF-1918396, the ONR under Contract N68335-22-C-0411, and DARPA under Contract HR001120C0107.

REFERENCES

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, page 19, 2010.

[2] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. SwitchV: automated SDN switch validation with P4 models. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 365–379, 2022.

[3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, page 503–514, 2014.

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 113–126, 2014.

[5] Manikandan Arumugam, Deepak Bansal, Navdeep Bhatia, James Boerner, Simon Capper, Changhoon Kim, Sarah McClure, Neeraj Motwani, Ranga Narasimhan, Urvish Panchal, Tommaso Pimpo, Ariff Premji, Pranjal Shrivastava, and Rishabh Tewari. Bluebird: High-performance SDN for bare-metal cloud services. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 355–370, 2022.

[6] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 44–57, 2004.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 155–168, 2017.

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.

[9] Nikolaj Bjørner and Karthick Jayaraman. Checking cloud contracts in Microsoft Azure. In *International Conference on Distributed Computing and Internet Technology (ICDCIT)*, pages 21–32, 2015.

[10] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 127–140, 2012.

[11] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, page 546–550, 2005.

[12] P4 Language Consortium. P4Lang Tutorials, Fetched July 15th, 2023. https://github.com/p4lang/tutorials.

[13] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *International Joint Conference on Artificial*

Intelligence (IJCAI), page 854–860, 2013.

[14] Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic (TOCL)*, apr 2009.

[15] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: Towards bug-free P4 programs. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 571–585, 2020.

[16] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 217–232, 2016.

[17] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, page 469–483, 2015.

[18] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 343–355, 2015.

[19] Open Networking Foundation. Aether: An open source 5G connected edge platform, 2022. https://opennetworking.org/aether/.

[20] Open Networking Foundation. Software Defined Fabric (SD-Fabric) Release 1.2, Fetched February 15th, 2023. https://docs.sd-fabric.org/master/release/1.2.0.html.

[21] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in P4 programs with assertion-based verification. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.

[22] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, page 300–313, 2016.

[23] William Sealey Gosset. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[24] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 71–85, 2014.

[25] Intel. Intel® Tofino™., Fetched February 10th, 2023. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[26] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 200–213, 2019.

[27] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, page 99–112, 2013.

[28] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 113–126, 2012.

[29] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 421–436, 2019.

[30] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–27, 2013.

[31] Hyojoon Kim, Xiaoqi Chen, Jack Brassil, and Jennifer Rexford. Experience-driven research on programmable networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 51(1):10–17, January 2021.

[32] Hyojoon Kim and Arpit Gupta. Ontas: Flexible and scalable online network traffic anonymization system. In *ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, pages 15–21, 2019.

[33] K Shiv Kumar, K Ranjitha, PS Prashanth, Mina Tahmasbi Arashloo, U Venkanna, and Praveen Tammana. DBVal: Validating P4 data plane runtime behavior. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2021.

[34] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for Software-Defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[35] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 490–503, 2018.

[36] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, page 499–512, 2015.

[37] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay ThakuR, Larry Peterson, Jennifer Rexford, and Oguz Sunay. A P4-based 5G user plane function. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, page 162–168, 2021.

[38] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 290–301, 2011.

[39] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *ACM SIGCOMM Internet Measurement Conference (IMC)*, page 393–407, 2018.

[40] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. P4Pktgen: Automated test case generation for P4 programs. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.

[41] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, page 323–334, 2012.

[42] Sundararajan Renganathan, Benny Rubin, Hyojoon Kim, Pier Luigi Ventre, Carmelo Cascone, Daniele Moro, Charles Chan, Nick McKeown, and Nate Foster. Hydra: Effective network verification (full version), September 2023. Available at https://www.cs.cornell.edu/~jnfoster/papers/hydra-tr.pdf.

[43] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtěch Havel, Hanneli Tavante, Rob Sherwood, Vlad Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. P4Testgen: An extensible test oracle for P4. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, 2023. To appear.

[44] Apoorv Shukla, Seifeddine Fathalli, Thomas Zinner, Artur Hecker, and Stefan Schmid. P4consist: Toward consistent P4 SDNs. *IEEE Journal on Selected Areas in Communications (JSAC)*, 38(7):1293–1307, 2020.

[45] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with Vera. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 518–532, 2018.

[46] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with Pathdump. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 233–248, 2016.

[47] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with SwitchPointer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 453–456, 2018.

[48] 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. Aquila: A practically usable verification system for production-scale programmable data planes. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, page 17–32, 2021.

[49] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. *ACM SIGCOMM Computer Communication Review (CCR)*, 42(4):419–430, August 2012.

[50] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On static reachability analysis of IP networks. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 2170–2183, 2005.

[51] Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2015.

[52] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, April 2016.

[53] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 701–718, 2020.

[54] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Emerging Networking EXperiments and Technologies (CoNEXT)*, page 241–252, 2012.

[55] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, page 87–99, 2014.

[56] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 221–239, 2020.

[57] Peng Zhang, Hao Li, Chengchen Hu, Liujia Hu, Lei Xiong, Ruilong Wang, and Yuemei Zhang. Mind the gap: Monitoring the control-data plane consistency in software defined networks. In *Emerging Networking EXperiments and Technologies (CoNEXT)*, page 19–33, 2016.