Liveness Verification of Stateful Network Functions

Farnaz Yousefi, *Johns Hopkins University;* Anubhavnidhi Abhashkumar and Kausik Subramanian, *University of Wisconsin-Madison;* Kartik Hans, *IIT Delhi;* Soudeh Ghorbani, *Johns Hopkins University;* Aditya Akella, *University of Wisconsin-Madison*

https://www.usenix.org/conference/nsdi20/presentation/yousefi

This paper is included in the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)

February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Open access to the Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20) is sponsored by

Liveness Verification of Stateful Network Functions

Farnaz Yousefi
Johns Hopkins University

Anubhavnidhi Abhashkumar
University of Wisconsin-Madison

Kausik Subramanian
University of Wisconsin-Madison

Kartik Hans
*IIT Delhi**

Soudeh Ghorbani
Johns Hopkins University

Aditya Akella
University of Wisconsin-Madison

Abstract

Network verification tools focus almost exclusively on various safety properties such as "reachability" invariants, *e.g.,* is there a path from host *A* to host *B*? Thus, they are inapplicable to providing strong correctness guarantees for modern programmable networks that increasingly rely on stateful *network functions*. Correct operations of such networks depend on the validity of a larger set of properties, in particular *liveness* properties. For instance, a stateful firewall that only allows solicited external traffic works correctly if it *eventually* detects and blocks malicious connections, *e.g.,* if it eventually blocks an external host *E* that tries to reach the internal host *I* before receiving a request from *I*.

Alas, verifying liveness properties is computationally expensive and, in some cases, undecidable. Existing verification techniques do not scale to verify such properties. In this work, we provide a compositional programming abstraction, model the programs expressed in this abstraction using compact Boolean formulas, and show that verification of complex properties is fast on these formulas, *e.g.,* for a 100-host network, these formulas result in 8× speedup in the verification of key properties of a UDP flood mitigation function compared to a naive baseline. We also provide a compiler that translates the programs written using our abstraction to P4 programs.

1 Introduction

In recent years, network verification has emerged as a crucial framework to check if networks satisfy important properties. While there are a variety of tools that differ in their focus (*e.g.,* verifying current data plane snapshot vs. verifying a network's control plane under failures), they all share a common attribute: they focus mainly on verifying various flavors of reachability invariants: Is a point in the network reachable from another point [33, 34, 42, 43]? Is there a loop-free path between them [33, 34, 42, 43]? Is the path congested [27,29,38]? Does it traverse a waypoint [41,58]? Is the reachability preserved under link failures or external messages [10, 26]? Are all datacenter shortest paths available for routing [30]? *etc.* Crucially, these tools leave out a richer set of properties that depend on networks guaranteeing *liveness*.

Networks today increasingly deploy complex and stateful network functions such as intrusion detection/prevention systems (IDPS) that monitor traffic for malicious activity and policy violations, and they prevent such activities. To rely on such networks, operators need to verify if "something good (a desired property) eventually happens" [47]—i.e., liveness. As a concrete example, consider a stateful firewall with the policy that a host *E* external to an enterprise is only allowed to send traffic to an internal host *I* if *I* sends a request to *E* first. The liveness property here is "will a host *E* that should be allowed to send traffic to *I* (*i.e., E* was first contacted by *I*) eventually get whitelisted?", or more precisely that "the event of *I* sending traffic to *E* leads to the firewall's whitelisting of *E*" (§3). Existing reachability-centric tools cannot verify this property: the existence of paths from *I* to *E* does not show whether any packet has actually traversed that path; similarly, the reachability of *I* from *E* does not give any guarantees whether it was established before or after *I* sent traffic to *E*.

Recent work has shown that reachability verification can be made efficient by operating on *Equivalence Classes* (ECs), *i.e.,* groups of packets that experience the same forwarding behavior [33, 34, 43] on a *static* snapshot of the network state. However, verifying liveness is not amenable to such techniques as liveness properties reason about progress and concern the succession of events in dynamic systems. They cannot be verified on a static snapshot of the system.

A dynamic network, in which the state changes, can be modeled as a state machine and, conceptually, existing static verification approaches [35] could be extended to reason about properties of the states and transitions of this machine (§3). However, this naive approach results in state explosion as the network size increases and is therefore impractical (§3, §4).

In this paper, we argue that the goal of verifying liveness properties is achievable using a top-down *function-oriented* strategy that rethinks network abstractions with the efficiency

*Work done during an internship at Johns Hopkins University.

of verification in mind. To realize this vision: (a) we provide network programmers with a simple, familiar *abstraction* of one big switch to express their intent. This abstraction enforces the logical separation of different network functions (§2). (b) We then *model* the program as a compact "packet-less" data structure that, unlike the common approach of modeling the forwarding behavior for classes of packets [33,34,42,43], abstracts away the explicit notion of packets and focuses instead on the entities responsible for implementing functions: packet handling rules. (c) We build and evaluate a prototype of our system.

Abstraction: We provide a unified abstraction of one big switch for both control and data planes that conceptually handles all packets. This abstraction closely resembles the simple, familiar data plane abstraction of one big switch directly connecting all hosts [36, 49]. In contrast to the data plane one-big-switch, our abstraction does not require a separate control plane to program it. To reduce verification time, we enforce a *functional decomposition* by requiring the user to implement each function using a separate logical flow table.

Modeling and verification: For verifying properties, similar to prior work [28], we focus exclusively on network behaviors *visible* to users. This enables us to model the system behavior using a compact "packet-less" data structure that abstracts away any details invisible to users, such as the hop-by-hop journey of the packet inside the network [33,34,42,43,52] or even the explicit notion of packets or classes of packets (§3). The packet-less structure models changes in the forwarding behavior as Boolean transitions. We demonstrate the verification efficiency of the packet-less model experimentally (§4), *e.g.,* for a UDP flood mitigation function, within a 1,000*sec.* time-bound, it enables verifying a key liveness property (*host A is eventually blocked*) for networks that are 3.5× larger than those verifiable with the aforementioned naive approach (extending static verification to deal with network states and dynamic transitions).

Implementation and evaluation: We develop a prototype of our design that exposes two interfaces (to express functions on the one-big-switch abstraction and specify verification properties) and a P4 compiler that converts such functions to programs executable on today's programmable devices. Our evaluations show the expressiveness of our interfaces, their low overhead, and the efficiency of our verification design, *e.g.,* for a 100-host network, the packet-less model verifies key liveness properties of a UDP flood mitigation function 8× faster than a packet-based baseline—a gap that increases with the network scale (§4).

2 A Unified Switch Abstraction

To simplify programming and relieve network programmers of the burden of writing distributed, multi-tier programs, we provide the abstraction of a single, centralized switch that conceptually handles *every* packet. This approach for simplifying programming by having a single unified abstraction for both the control plane and data plane is inspired by Maple [59] and deployed in Flowlog [46]. In contrast to Maple that allows programmers to use standard programming languages and Flowlog's SQL-like language, we start with the most basic and familiar data plane abstraction in networks: one switch that directly connects all hosts. We then augment this abstraction to make it programmable. Similar to Maple and Flowlog, we proactively compile the programs written on our abstraction to control and data plane programs executable in today's networks (§4). We describe our abstraction, its distinction from the common one-big-switch abstraction, and how several canonical network functions can be programmed on it, in turn.

One-big-switch: A common data plane abstraction in networking is that of one logical switch with multiple flow tables, each containing a set of rules, that directly connects all users' hosts together [5, 32, 36, 44]. A rule generally includes: (a) a match field to match against packet headers and ingress ports, (b) priority to determine the matching precedence of rules, (c) counters that are updated when packets are matched, (d) timers that show the time that the rule will be expired and removed from the switch, and (e) actions that are executed when a packet matches the match field of an unexpired rule that has the highest priority among all unexpired rules. These actions could result in changes in the packet, dropping it, or forwarding it. We use R.`match`, R.`priority`, R.`counter`, R.`timer`, and R.`action` to denote the match, priority, counter, timer, and action of rule R.

Our goal is to provide a similar abstraction to this familiar abstraction while making it programmable and amenable to fast verification. In particular, rather than a static, stateless switch, the programmer should be able to implement *dynamic* functions whose behaviors change over time based on traffic. Plus, she should be able to focus solely on the high-level *functionality* that she wishes her switch to provide (*e.g.,* the firewall policies), and the provider of the abstraction is responsible for handling the low-level, distributed implementation of the functionality including reachability (*e.g.,* ensuring that all required packets are correctly forwarded through the firewall). To further assist the user in developing her desired functions, an ideal framework should also provide modular programming and separate the functionality of a program into independent modules. Towards these goals, we make the following alteration in the one-big-switch abstraction [5, 36]:

Add/delete actions: In addition to the standard SDN actions (forward, drop, *etc.*), we allow the execution of a rule to add or remove rules from the switch. As an example, to program a stateful firewall that allows an external host E to talk to an internal host I only after I sends E a request, the switch needs a rule that, upon receiving a request from I to E, alters its state to allow E to I communication.

Actions `add(`R`)` and `delete(`R`)` show that the execution of the rule results in, respectively, adding and deleting rule

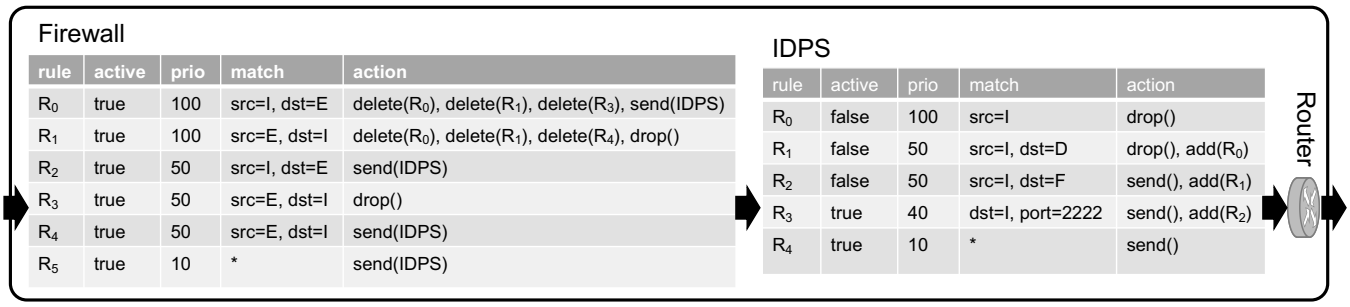


Figure 1: One-big-switch implements firewall and IDPS.

R. To each rule R, we add a boolean variable R.`active` to show if the rule is currently active on the switch, *i.e.,* packets are matched against it. As an example, the initial state R_0.`active=true`; R_1.`active=false`; R_0.`action=add`(R_1) shows that, unlike R_0, rule R_1 is not initially active and will be activated as a result of R_0's execution.

Actionable measurements: For a rule R_i, we allow users to define match fields as predicates not just on packet headers (*e.g., src*=*A*) but also as conditions in the forms of $l_i \leq c_j < u_i$ on counters, where c_j is the j^{th} counter and l_i and u_i are constants. This condition expresses that for the rule to match a packet, the value of counter c_j should be in the $[l_i, u_i)$ interval.[1] We assume that counters are bounded in the range $[0, m)$. Counters enable the network programmer to easily write critical network functions that depend on traffic statistics such as security applications that detect SYN flood, port scanning, DNS amplification, *etc.* [7, 16, 23] and campus network monitors [35] that block users once their usage exceeds the quota specified by the campus policy. Whenever a rule with a counter is executed on a packet, the counter value is incremented by one. We allow multiple rules to share a counter.

Other optimizations: To improve programmability and verification speed, we also perform the following optimizations: (a) *Functional decomposition:* the user is provided separate tables for each of her network functions, *e.g.,* one table for her firewall, one table for her load balancer, *etc.* If a packet matching a rule R in function F_1 needs to be sent to another function F_2, *e.g.,* an IDPS rule needs to send a packet to the traffic scrubber, this is expressed as R.`action=send`(F_2) in table F_1. (b) *Declarative routing:* Our abstraction provides routing and forwarding as a service, that we call the *router*, to its users. This liberates the user from computing paths and updating them after infrastructure changes, such as changes in policy, topology, and addressing. The user only declares the goal that a packet should reach a destination and delegates the task of figuring out *how* this is done to the provider. Action rules `send`(A) and `send`() simply express the intent of the user to forward a matching packet to the endpoint with ID A and to the packet's destination, respectively. In our prototype, for implementing our declarative routing service, we deploy a basic shortest path forwarding function [1]. (c) *Symbolic rule representation:* the original one-big-switch rules can express a restricted subset of predicates on explicit packet headers, *e.g.,* `src-IP=10.0.0.1 ∧ dst-IP=10.0.0.2`. Declarative routing enables us to provide a higher-level abstraction to express any general predicate on sets of endpoint identities and header fields, *e.g.,* the programmer can declare a forwarding policy for packets sent to or from a set of hosts called T via a single rule R: R.`match=(src=`T`∨dst=`T`)`, R`.action=send()`, without managing low-level details such as the physical locations and IP addresses of T's hosts.

The changes above turn the data plane one-big-switch abstraction into a unified abstraction that can be efficiently verified: add/delete actions and actionable measurements make the abstraction programmable, and other optimizations accelerate the verification process by reducing the program size (§4). To further assist with efficient verification, our abstraction is designed to have less expressive power than Turing-complete control plane programming languages [46] such as Floodlight [4] and Pyretic [44]. We find experimentally that despite being computationally universal, in practice, control planes perform only a limited set of operations, *e.g.,* adding and deleting rules based on traffic patterns. Our one-big-switch abstraction is designed to be capable of performing similar computations and is therefore expressive enough to program a wide range of network functions. Table 1 lists the functions of a few common control planes and recent network abstractions that we re-wrote on top of our abstraction (implementations in §8). To support the functions that cannot be expressed on top of our abstraction, such as content-based security policies, our framework allows the use of external libraries in conjunction with our abstraction. However, we can only verify the programs expressed on our abstraction.

Examples. Figure 1 shows an illustrative example where the user deploys two tables to implement a chain of two canonical functions: a stateful firewall followed by an IDPS. The first table implements a stateful firewall at the periphery of an enterprise that allows endpoint E (*e.g.,* as an external host) to talk to I (*e.g.,* as an internal host) only if I sends a request to E first. Initially, the table has high priority rules that "watch for" traffic between I and E (R_0 and R_1). If traffic from E to I

[1] If the rule only increases the value of c_j when executed without defining a condition on c_j, we assume l_i=0, and u_i= m, where m is the maximum value supported for counters.

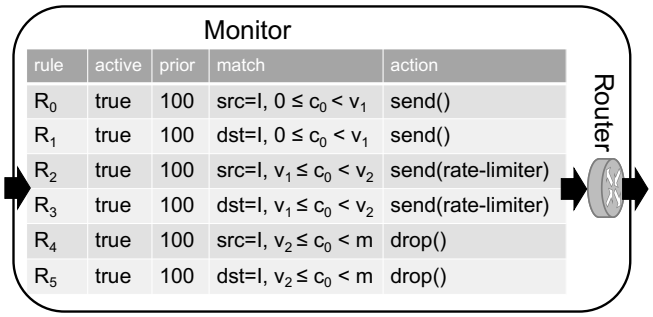


Figure 2: One-big-switch implements a monitor.

is observed first, the execution of rule R_1 drops the traffic and changes the state to block E from reaching I. Receiving traffic from I to E first, however, triggers the execution of R_0 which in turn allows bidirectional traffic between I and E to pass through the firewall. Packets not discarded by the firewall are always sent to the next function, the IDPS explained below, for further monitoring. Note that when defining rules in the one-big-switch abstraction, the match fields of the rules must be defined in a way that at least one rule matches each received packet.

The second table implements an IDPS that detects and blocks Trojans. The IDPS determines if the internal host I is infected and needs to be blocked based on a recognizable fingerprint of a backdoor application [16] with the following sequence of operations: (i) I receives a connection on port 2222, (ii) I connects to an FTP server F, and (iii) I tries to connect to the database server D.

Initially, the table contains two active rules: R_3 that matches traffic destined to I on port 2222, and a lower priority rule R_4 that matches all other traffic. Both rules forward the traffic to its destination. The execution of R_3, however, corresponds with the (i) operation above. Once triggered, it activates R_2 that is executed once I tries to reach F (operation (ii)). R_2's execution, in turn, activates R_1, and once I tries to send traffic to D (operation (iii)), it gets blocked (R_1 activates R_0). Once the traffic goes through this pipeline of tables, it is handed to the router to be delivered to its destination.

Figure 2 shows another example for an application that implements a simple campus policy in which an inside-campus host I is allowed to send and receive data before hitting a utilization cap v_1. Once its usage exceeds v_1, but before it reaches v_2, its traffic is routed through a rate limiter. After its usage passes v_2, its access is blocked.[2] A survey of campus network policies shows that universities commonly deploy such usage-based rate-limiting [35]. While for simplicity, we only provide examples of linear chaining of functions in this section (*e.g.,* IDPS after firewall), our abstraction is general enough to express arbitrary dependencies between functions. An example is provided in §8.

[2]In practice, such policies are usually enforced periodically, *e.g.,* the rules (along with their counters set to 0) are re-installed daily.

3 Function Verification

A standard approach for verifying liveness properties of dynamic systems is modeling the behavior of the system as a transition system and expressing its desired properties using temporal logics. The complexity and scalability of this approach depend on the size of the state machine. In this section, we show how we can model stateful dynamic network functions as compact, Boolean "packet-less" transition systems that can be verified efficiently. In §4, we show experimentally that this approach significantly reduces the average verification time for canonical applications compared to a naive packet-based baseline.

3.1 Network as a Transition System

We can model the network as a *transition system*, an analytical framework for reasoning about the behavior of dynamic systems where nodes represent the states of the system (each state corresponds to a valuation of system variables) and edges represent state transitions [8]. Each transition system has a set of initial states as well as a labeling function that maps each node to a set of properties that hold in that state. More formally, a *transition system TS* is a tuple $(S,\ Act,\ I,\ AP,\ L)$:

- S is a set of states,
- Act is a set of actions,
- $\rightarrow\ \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L\ :\ S \rightarrow 2^{AP}$ is a labeling function.

For convenience, we write $s\ \rightarrow^{\alpha}\ s'$ instead of $(s,\ \alpha,\ s')$. Intuitively, a transition system starts in some initial state $s_0 \in I$ and evolves according to the transition relation $\rightarrow$. That is, if s is the current state, then a transition $s\ \rightarrow^{\alpha}\ s'$ is selected nondeterministically and taken, *i.e.,* the action α is performed, and the system evolves from state s into state s'.

In networks, the state of the network at each point is its forwarding state (*e.g.,* rules and counters), and transitions are the events that change the state, *e.g.,* policy updates. We next show the properties that can be expressed on these transition systems, explain what makes liveness verification hard, and demonstrate how we can model the network as a compact "packet-less" transition system.

Atomic propositions: *Atomic propositions* are Boolean-valued propositions that express simple known facts about the state of the system. We define these propositions as (h,a) pairs where h and a are, respectively, an equivalence class of packets, and a list of actions (*e.g.,* `send`(I)). For the network transition system TS, an atomic proposition (h,a) holds in a state $s \in S$ if action a applies to all the packets in h.

Labeling function: A *labeling function* L relates a set $L(s) \in 2^{AP}$ of atomic propositions to any state $s \in S$.[3] In the network transition system TS, L maps each state to the set

[3]Recall that 2^{AP} denotes the power set of atomic propositions.

of atomic propositions that hold in that state. That is, (h, a) exists in $L(s)$ if the list of actions a is applied to all the packets in h in state s.

Properties: An execution of a program can be shown as an infinite sequence of states: s_0, s_1..., where each state s_i results from executing a single action in state s_{i-1}.[4] A program's behavior is the set of all such sequences. A *property* is also defined as a set of such sequences [6]. A property *holds* in a program if the set of sequences defined by the program is contained in the property [6]. A partial execution is a finite sequence of program states.

Temporal logic to express properties: Temporal logic is an extension of ordinary logic that facilitates expressing properties via adding assertions about time. Here, we adopt linear temporal logic (LTL) [19], which can express various liveness and safety properties [37, 47]. *Temporal logic assertions* are built up from atomic propositions using the ordinary logical operations $\wedge, \vee$, and $\neg$ and some temporal operators—if P and S are atomic propositions: (1) $\mathtt{G}P$ implies "now and forever" P holds, (2) $\mathtt{F}P$ implies "now or sometime in the future" P holds,[5] (3) $P \rightarrow S$ shows logical implication and implies that if P is true now then S will always be true, (4) $P\mathtt{U}S$ implies P remains true "until" S becomes true, and (5) $\mathtt{O}P$ implies P holds in the "next" state. Using the above temporal operators, we can express various properties, *e.g.,* $\mathtt{F}((\mathtt{src}{=}E \wedge \mathtt{dst}{=}I), \mathtt{send}(I))$ [47] asserts that eventually E-to-I packets are delivered, *i.e., E* can eventually reach I.

3.2 Liveness vs. Safety

A key categorization of properties in distributed systems is into safety and liveness. The categorization is important as the two groups are proved using different techniques [6]. Informally, a *safety* property stipulates that some "bad thing" (deadlocks, two processes executing in critical sections simultaneously, *etc.*) never happens and *liveness* guarantees that "something good" (termination, starvation freedom, guaranteed service, *etc.*) eventually happens [47]. That is, for a property P to be a safety property, if P does not hold for an execution, then at some point, some irremediable "bad thing" must happen. Most of the properties verified in networks today are safety: if a property—such as reachability invariants (E is always reachable from I) [33, 34, 42, 43], waypoint (a certain class of traffic always traverses an intrusion detection system) [41, 58], congestion-freedom [27, 29, 38], and loop-freedom [33, 34, 42, 43]—is violated, then there is an identifiable point—such as a change in the latest snapshot of the network— at which the "bad thing" happens [6].

A partial execution γ is *live* for a property P if and only if there is a sequence of states β such that P holds in $\gamma\beta$. A property for which *all* partial executions are live is a liveness property. In contrast to safety, for liveness properties, no partial execution is irremediable—it is always possible for the required "good thing" to happen in the future [6]. This makes detecting liveness violations challenging as it fundamentally requires an exhaustive search of the entire state space of the network. In the next section, we discuss our approach to overcoming this challenge by modeling networks with compact transition systems. Some examples of liveness properties in networks are (a) the intrusion detection system *eventually* detects all infected hosts, (b) all hosts *eventually* become reachable, *e.g.,* after routing convergence, and (c) showing a recognizable fingerprint of a backdoor application *leads-to* the host being blocked. More generally, "event A leads-to event B" and "event B eventually happens" are two classes of liveness properties [47] as it is always possible for "something good" (*i.e.,* event B) to happen. An example of a property that includes liveness is *total correctness* which is composed of partial correctness (the program never generates an incorrect output; a safety property) and termination (the program generates an output; a liveness property) [6].

3.3 Packet-less Model

Feasibility of model checking is tied inherently to handling state explosion [19]. To mitigate this problem, we try to provide a compact "packet-less" structure that models only the entities that perform the functions in the network: packet handling rules. Plus, we model rules in the most abstract form: as *Boolean* variables, abstracting away all of the attributes of rules, such as their match fields, actions, and priorities. This is in contrast to pervasive network verification techniques that model the network behavior in terms of packets and equivalence classes of packets (ECs) [34, 35, 43].

Boolean variables and formulas provide a more compact way to represent the state space of a transition system. State-of-the-art model checkers like NuSMV [18] use symbolic techniques, such as Binary Decision Diagrams (BDDs), for efficiently exploring transition systems that have a Boolean representation. Such representations enable model checkers to explore extremely large state spaces efficiently [15]. We next explain how we can encode the behavior of dynamic network functions as packet-less models. We then show in §4 that this approach of packet-less modeling results in a significant reduction in the verification time of canonical network functions compared to a packet-based approach.

Compact, packet-less models: We initially model the network as a transition system TS with a single Boolean variable corresponding to each rule. This variable is true if the rule is active in the network and false otherwise. As a starting point, we abstract away counters, assuming that rules do not depend on them. (We will later refine this model to incorporate counter semantics.) Each node in this structure shows one **state** of the network defined by a valuation of Boolean

[4] Terminating executions are modeled as sequences where the last state reached by the terminating trace repeats infinitely.

[5] Note that $\mathtt{F}$ is the dual of $\mathtt{G}$, *i.e.,* $\mathtt{F}P$ is equivalent to $\neg\mathtt{G}\neg P$, where P is an atomic proposition [47].

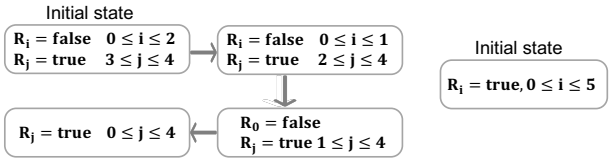


Figure 3: Packet-less models of the (a) IDPS and (b) monitor.

rule variables. S is the set of all these states. A rule can be *executed* in a state if, for at least one packet that it matches, it is the highest priority rule with value=`true` in that state. **Actions** are the execution of rules that update the forwarding state by adding or deleting rules, and **transitions** show how the state evolves as a result of these actions. If in a network state $s \in S$, rule R_i can be executed and its execution adds or deletes rules, then there is a transition $s \rightarrow^{R_i} s', s' \in S$, where for each added rule R_j, R_j=`true` in s' and for each deleted rule R_k, R_k=`false` in s'. In the initial state, the value of each R_i is equal to the initial value of R_i.`active` in the original program. Figure 3 shows the packet-less models for the IDPS and the campus monitor functions in Figures 1 and 2 (not including labeling functions).

Despite being more efficient, not explicitly modeling packets causes some key challenges: (1) Abstracting away the notion of packets makes it challenging to incorporate the semantics of traffic statistics such as counters. (2) As §3.1 explains, properties are defined on packets and whether a property holds in a network state depends on the actions of the highest priority rule that matches the packet. Thus, it is challenging to verify properties on a structure that abstracts away any explicit notion of packets, headers, priorities, *etc.* For the IDPS function in Figure 1, for example, the packet-less model has two rules in the initial state that can match packets sent to I with port number 2222, as Figure 3 (a) shows. With each rule modeled as a Boolean variable, it is not possible to determine which matching rule handles a packet (and therefore verify properties). We address these challenges next.

3.3.1 Boolean Formulas of Counters

The first challenge of the Boolean packet-less modeling is preserving the semantics of stateful programs with traffic counters. Recall that in packet-less models, we initially abstracted away counters. We next show how to refine these models to incorporate semantics of counters.

Refining states: To model counters, we observe that if the only variable that changes across a set of network states is a counter value and the value of this counter does not pass counter conditions, then the forwarding behavior remains the same in all those states. In the monitor function, the network behavior is identical for all counter values between 0 and v_1. This allows us to track counter *predicates*, Boolean-valued functions on counters, instead of actual counter values. In the monitor function, we can define the following three predicates: $(0 \le c_0 < v_1)$, $(v_1 \le c_0 < v_2)$, and $(v_2 \le c_0 < m)$. In the initial state, only the first predicate, $(0 \le c_0 < v_1)$, is true.

The fact that the forwarding behavior is determined not by exact counter values but by counters passing thresholds makes counters amenable to *predicate abstraction*, a powerful technique to mitigate the challenges of verifying programs with large base types such as integers [19]. This technique reduces the size of the model by tracking only predicates on data and eliminating invisible data variables.

Concretely, counter conditions partition an interval into subintervals that may have distinct forwarding behaviors. Let R_i be a rule that depends on the j^{th} counter, c_j, *i.e.,* it is active if c_j's value is in the $[l_i, u_i)$ range, and $P_j = \text{order}(\cup_i(l_i \cup u_i))$, *i.e.,* an ordered list (in non-decreasing order) of all lower and upper bounds of all rules that depend on c_j. $P_{j,k}$, $l_{j,k}$, and $u_{j,k}$ denote, respectively, the k^{th} subinterval of counter c_j, its lower bound, and its upper bound. In the monitor program, $P_{0,0} = [0, v_1)$ is the first subinterval of the first counter, $l_{0,0} = 0$, and $u_{0,0} = v_1$. When the counter value is in this subinterval, rule R_0 can handle packets as its counter conditions, $(l_0 \le c_0 < u_0)$, are satisfied, *i.e.,* $(l_0 \le l_{0,0}) \wedge (u_{0,0} \le u_0)$ where $l_0 = 0$ and $u_0 = v_1$. R_2, on the other hand, cannot handle packets because its counter conditions are not satisfied in this subinterval, *i.e.,* $(l_2 \not\le l_{0,0}) \wedge (u_{0,0} \not\le u_2)$ where $l_2 = v_1$ and $u_2 = v_2$.

We refine TS by adding a Boolean variable R'_i for every rule R_i. Our goal is to set the value of this variable to `true` in a state s if the counter conditions of R_i are satisfied in s and to `false` otherwise. For any rule R_k that does not depend on counters, R'_k=`true`.[6] The numbers in P_j are the only places where R'_i variables of the rules that depend on c_j can change. In the monitor program, $P_0 = [0, v_1, v_2, m]$ lists the only points in the $[0, m)$ range where the condition of a rule that depends on counter c_0 such as R_0 can transition from `true` to `false` and vice versa.

For each counter c_j in a state in the packet-less model, we partition the state into $|P_j| - 1$ states, where $|P_j|$ is the number of points in P_j, *e.g.,* P_0=4 in the monitor example. Each of these $|P_j| - 1$ states corresponds to one subinterval. Suppose that R_i is a rule that depends on counter c_j, *i.e.,* R_i can handle packets when the value of the counter satisfies its counter conditions: $l_i \le c_j < u_i$. The value of R'_i in each refined state should show whether the counter conditions of rule R_i are satisfied in the corresponding subinterval $P_{j,k}$: $R'_i = ((l_i \le l_{j,k}) \wedge (u_{j,k} \le u_i))$.

In any given state $s \in S$ and for a subinterval of $P_{j,k}$, the network behavior is determined by the rules that (a) are active in that state (*i.e.,* R_i=`true`) and (b) either do not depend on counters or their counter conditions are satisfied in that subinterval (*i.e.,* R'_i=`true`), *e.g.,* in the monitor example's initial state, R_0 and R_1 are active (*i.e.,* R_0=R_1=`true`) and their counter conditions are satisfied in the first subinterval $P_{0,0} = [0, v_1)$ (*i.e.,*

[6] Note that it is possible to avoid defining these variables for the rules that do not depend on counters. We define these variables for all rules here for ease of exposition.

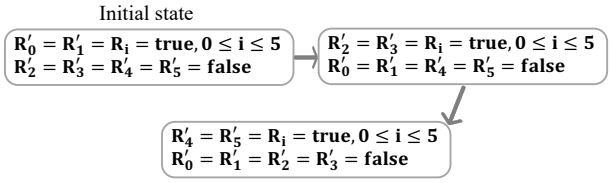


Figure 4: The refined structure for the monitor function.

R'_0=R'_1=true).

In summary, the set of variables in the packet-less model TS includes a pair of Boolean variables R_i and R'_i for each rule. The values of these variables at any point define the **state** of the packet-less model at that point. In the **initial state**, for every rule R_i, the value of R_i is equal to R_i.true in the original program. In the initial state, R'_i=true *iff* R_i either does not depend on any counters or if it depends on counter c_j and its counter conditions are satisfied in the first subinterval of c_j, $P_{j,0}$, *i.e.,* $((l_i \le l_{j,0}) \wedge (u_{j,0} \le u_i))$.

In the refined model, a rule R_i can be *executed* in a state if (a) for at least one packet that R_i matches, it is the highest priority rule, (b) R_i is active in that state (*i.e.,* R_i=true), and (c) R_i either does not depend on any counters or its counter conditions are satisfied in that subinterval (*i.e.,* R'_i=true).

Transitions in the refined model: Let s_k and s_{k+1} be, respectively, the k^{th} and $(k+1)^{\text{th}}$ refined state if state s is refined for counter c_j, *i.e.,* the states representing the Boolean formulas for subintervals k and $k+1$ of P_j. We add a transition from state s_k to state s_{k+1} if there is at least one rule R_i that depends on c_j and can be executed in s_k. The reason is that c_j increases monotonically so if a counter-dependent rule can be executed in s_k, it can result in c_j transitioning to the next subinterval (corresponding to state s_{k+1}). For each transition $s \rightarrow^{\alpha} s'$ in the original packet-less model before refinement, where α is the execution of a rule R_i that adds or deletes rules, there is a transition $s_k \rightarrow^{\alpha} s'_k$ if R_i can be executed in s_k. As before, for each added rule R_j that R_i adds, R_j=true in s'_k and for each rule R_k that R_i deletes, R_k=false in s'_k. Figure 4 shows the Boolean packet-less states and the transitions for the monitor function in Figure 2 (see Figure 3 (b) for its corresponding pre-refinement packet-less model). There is a transition from the initial partitioned state (in which $R'_0 = R'_1$=true) to a state in which $R'_0 = R'_1$=false, reflecting the fact that the network state evolves from an initial state that satisfies the counter conditions of these rules to one where the counter conditions of these rules are no longer satisfied. If the program has multiple counters, the state and transition refinements are performed sequentially for each counter.

3.3.2 Boolean Formulas of Properties

We explained earlier how we could express *atomic propositions* in terms of packets and the desired actions on them. Properties are built out of atomic propositions, and in a transition system such as the packet-based model in prior work [35], the atomic proposition (h,a) holds for a state *iff* the list of actions a is applied to all the packets in h in that state. The second challenge of packet-less modeling is evaluating a proposition on Boolean packet-less models. In this part, we explain how an (h,a) proposition can be expressed as a Boolean formula on rules exploiting a priori known rule priorities.

We say that a rule R_i can be *applied* if (a) R_i is active, *i.e.,* R_i=true and (b) R_i either does not depend on any counters or its counter conditions are satisfied, *i.e.,* R'_i=true. For a rule to be executed, in addition to satisfying the conditions above, for at least one packet that it matches, it should be the highest priority rule.

Let $W_{0,n} = [R_0, R_1, ..., R_n]$ and $W'_{0,n} = [R'_0, R'_1, ..., R'_n]$ be, respectively, the list of rules R_i and the list of variables R'_i, sorted in non-increasing order of the priorities of rules, *i.e.,* R_i.priority $\ge R_j$.priority, R'_i.priority $\ge R'_j$.priority if $i < j$. Our goal is to express whether a proposition holds in a state as a Boolean formula on this list. We achieve this with a recursive function: Let $K((h,a), W_{0,n}, W'_{0,n})$ be a function that is true if $W_{0,n}$ and $W'_{0,n}$ satisfy the proposition (h,a) and false otherwise. For the two special cases, where (1) h is empty and (2) h is not empty and the list of rules is empty, we assume that $K((h,a), W_{0,n}, W'_{0,n})$ evaluates to, respectively, true and false because any condition holds for a non-existing packet (item (1) above, $h = \varnothing$) and an empty set of rules does not satisfy any conditions for packets (item (2) above, $h \neq \varnothing$ and the list of rules=[]). In other conditions, we have the following cases:

Case 1: If the highest priority rule R_0 matches some packets in h, *i.e.,* if $(h \cap R_0$.pkts$) \neq \varnothing$, where R_i.pkts denotes the set of packets that R_i matches, and its action includes the actions in the proposition, *i.e.,* $a \subset R_0$.action, then for the proposition to hold, one of these two conditions should hold (a) either R_0 can be applied ($R_0 \wedge R'_0$) and for all the packets of h that do not match R_0, the proposition should hold for the next, lower-priority matching rules, *i.e.,* $(R_0 \wedge R'_0) \wedge (K((h - R_0$.pkts$, a), W_{1,n}, W'_{1,n}))$, or (b) R_0 cannot be applied ($\neg(R_0 \wedge R'_0)$, R_0 either is not installed in the network or its counter conditions are not satisfied), but in this case, for all the packets of h, the proposition should hold for the next, lower-priority matching rules, *i.e.,* $\neg(R_0 \wedge R'_0) \wedge (K((h,a), W_{1,n}, W'_{1,n}))$. In other words, $K((h,a), W_{0,n}, W'_{0,n}) = ((R_0 \wedge R'_0) \wedge (K((h - R_0$.pkts$, a), W_{1,n}, W'_{1,n}))) \vee (\neg(R_0 \wedge R'_0) \wedge (K((h,a), W_{1,n}, W'_{1,n})))$.

Case 2: If the highest priority rule R_0 matches some packets in h, *i.e.,* if $(h \cap R_0$.pkts$) \neq \varnothing$, and its action does not include the actions in the proposition, then for the proposition to hold, it should not be possible to apply R_0, *i.e.,* $\neg(R_0 \wedge R'_0)$. Otherwise, it matches the packets but does not apply the intended actions on them. Plus, for all the packets of h, the proposition should hold for the next, lower-priority matching rules, *i.e.,* $K((h,a), W_{0,n}, W'_{0,n}) = \neg(R_0 \wedge$

$R'_0) \wedge (K((h,a), W_{1,n}, W'_{1,n}))$.

Case 3: If the highest priority rule R_0 does not match any packets in h, then irrespective of whether R_0 can be applied or not, for all the packets of h, the proposition should hold for the next, lower-priority matching rules, *i.e.,* $K((h,a), W_{0,n}, W'_{0,n}) = K((h,a), W_{1,n}, W'_{1,n})$.

Applied recursively to the ordered list (according to priority) of all rules in the network, $K((h,a), W_{0,n}, W'_{0,n})$ expresses the proposition (h,a) as a Boolean formula on R_i and R_i' variables. As an example, in the IDPS function of Figure 1, the proposition $((\texttt{src=}I, \texttt{dst=}E), \texttt{send()})$ is translated to

$$
\begin{aligned}
K(((\texttt{src=}I, \texttt{dst=}E), \texttt{send()}), W_{0,4}, W'_{0,4}) &= \\
\neg(R_0 \wedge R'_0) \wedge (K(((\texttt{src=}I, \texttt{dst=}E), \texttt{send()}), W_{1,4}, W'_{1,4})) &= \\
\neg(R_0 \wedge R'_0) \wedge K(((\texttt{src=}I, \texttt{dst=}E), \texttt{send()}), W_{2,4}, W'_{2,4}) &= \\
\neg(R_0 \wedge R'_0) \wedge K(((\texttt{src=}I, \texttt{dst=}E), \texttt{send()}), W_{3,4}, W'_{3,4}) &= \\
\neg(R_0 \wedge R'_0) \wedge K(((\texttt{src=}I, \texttt{dst=}E), \texttt{send()}), W_{4,4}, W'_{4,4}) &= \\
\neg(R_0 \wedge R'_0) \wedge & \\
(\ (\ (R_4 \wedge R'_4) \wedge K((\varnothing, \texttt{send()}), [])) \vee & \\
(\neg(R_4 \wedge R'_4) \wedge K(((\texttt{src=}I, \texttt{dst=}E), \texttt{send()}), [])) &= \\
\neg(R_0 \wedge R'_0) \wedge (R_4 \wedge R'_4). &
\end{aligned}
$$

This Boolean formula results from applying the rules specified in cases (2), (3), (3), (3), (1), and the first two special cases, respectively. It has different truth values in different states in Figure 4 depending on the truth values of rules in those states, *e.g.,* it is true in the initial state S_0 where R_0=`false` and R_4=`true`, meaning that the assertion $((\texttt{src=}I, \texttt{dst=}E), \texttt{send()})$ holds (I may talk to E in this state), but is false in the final state (I is blocked), where R_0=`true` and R_4=`true`. Note that R'_i variables are always `true` in this example as the rules do not depend on counters.

In the network transition system TS, **labeling function** L maps each state to the set of atomic propositions that hold in that state. That is, (h,a) exists in $L(s)$ if the list of actions a is applied to all the packets in h in state s, *i.e.,* $K((h,a), W_{0,n}, W'_{0,n})$=`true` where $W_{0,n}$ and $W'_{0,n}$ are, respectively, the list of rules R_i and the list of variables R'_i (sorted in non-increasing order of rule priorities) in s.

4 Implementation and Evaluation

To evaluate the performance of our design, we build a prototype that enables network operators to program and verify their functions and a compiler that converts these functions to programs executable on programmable switching ASICs. After a brief overview of our prototype, we show that our network abstraction and specification language are expressive and impose only minimal overhead. We also show that compared to a packet-based baseline, the packet-less model's verification of different properties is faster and more scalable, *e.g.,* for a network with 100 hosts, the packet-less model results in 8× speedup in the verification of liveness properties of a UDP flood mitigation function compared to the packet-based model.

4.1 Implementation

Interfaces: Our system exposes two interfaces: a *one-big-switch* interface that enables a network operator to program her functions on our abstraction (§2) and a *specification* interface that allows her to express her desired properties (§3). Our *generator* then automatically builds the packet-less model and the Boolean formula representing the specification as explained in §3 and interacts with NuSMV, a state-of-the-art model checker [18], to verify specification properties.

Compiling the abstraction to P4 programs: P4 [14] is a language for expressing the packet processing of programmable data planes. Along with the programmable data plane, the control plane is responsible for populating the tables defined by the P4 program. We build a compiler to compile a one-big-switch program using our abstraction to a P4_16 program for the P4 behavioral model [3], an open-source programmable software switch. Along with the software switch, we develop a control plane which adds and deletes table rules. We describe some of the salient features of the compilation:

(a) Functional decomposition: We map each table in the abstraction to a P4 table, whose match fields and actions are constructed using the rules in the table. Network function traversal policies are implemented using P4 control flow constructs, *e.g.,* to traverse a firewall table `fw` conditionally:

```
if (meta.visit_fw) == 1)
      fw.apply();
```

where `meta.visit_fw` is a metadata variable used by the tables before the firewall table to ensure the packet goes through the firewall table.

(b) Actionable measurement: We use P4's registers to support incrementing and matching on counters shared across multiple rules. If a table in the abstraction uses counters, we create a separate P4 table which is responsible for updating the shared counters and transferring the counter state to metadata such that the function table can use the value for matching. This helps us confine register accesses to a single table. Thus, packet processing can happen at the line rate [54].

(c) Add/delete actions: Currently, P4 data planes do not support add/delete actions, *i.e.,* rule actions that add or delete other table rules, due to hardware limitations of existing platforms. We support this functionality by cloning the packet [55] to the control plane, acting similar to the PacketIn functionality in OpenFlow [2]. When a rule in the program has an add/delete action, our switch program clones the packet in the data plane and sends it to the switch-local

Function source	Function	One-big-switch LoC	Packets calling controller [%]	P4 LoC	Compilation time[ms]
Pyretic [44]/Kinetic [35]	Simple counter	1	0	144	1.8
	Port knocking	3	6×10^{-3}	133	1.8
	Simple firewall	2	0	128	1.6
	IP rewrite	2	0	134	1.8
	Simple rate limiter	3	0	141	2.1
Floodlight [4]	Firewall/ACL	2	0	132	1.7
Chimera [13]	Phishing/Spam detection	3	6×10^{-3}	151	2.3
FAST [45]	Simple stateful firewall	3	0.3	133	2.1
	FTP monitor	2	0	135	2.1
	Heavy-hitter detection	2	0	148	2.0
	Super spreader detection	2	0	148	2.0
	Sampling based on flowsize	6	0	189	2.5
Bohatei [23]	Elephant flow detection	3	0	182	2.4
	DNS amplification mitigation	3	7×10^{-3}	135	1.9
	UDP flood mitigation	2	0	148	1.8

Table 1: Applications written on one-big-switch.

control plane; the control plane (which we also generate) then adds/deletes table rules as decided by the add/delete actions in the program. This approach has the overhead of punting some packets (specifically, first packets in a connection matching specific rules) to the local switch CPU—this is a cost we pay for lack of data plane support for add/delete actions.

4.2 Evaluation

Expressiveness: Despite its simplicity, our one-big-switch abstraction enables developers to express a broad range of applications and network functions. Table 1 shows a list of functions that we have developed in our framework. Full descriptions of these programs are provided in §8. Network policies can be succinctly expressed on our one-big-switch abstraction in only a few lines of code (column 3 in Table 1), *e.g.,* a simple stateful firewall policy that allows only the traffic whose connection was initiated by a host in a given department can be expressed in 3 lines of code on the one-big-switch abstraction. The compiled P4 code of the same policy is expressed in 133 lines of code (comprising of 100 lines of boilerplate code for headers and parsers) and ∼50 lines of code for the P4 control plane for cloning the packets at the data plane and adding rules from the control plane. Various specification properties, including the most common specification patterns in practice [21], can also be specified in the temporal logic presented in §3.

Limited overhead: Add/delete actions of our abstraction are not directly supported in the switching ASIC today. Our P4 compiler implements these by involving the controller whenever a rule has such actions. To measure the overhead of involving the controller, we deploy the functions listed in Table 1 and replay a packet trace of a university datacenter [12], with over $102K$ packets and 1,791 IP addresses and measure the frequency of calling the controller. This overhead is modest for all functions, *i.e.,* 0-0.3% (column 4, Table 1).

Verification time: We test the efficiency of bounded-time verification of packet-less and packet-based models at scale to answer questions such as, what functions and properties are verifiable with each approach? How does the verification time scale with respect to the network (and hence the model) size? How does it scale with respect to the property size? To do so, for the functions in Table 1 with per-host policies (*e.g.,* the heavy-hitter detector, port knocking, rate limiter, phishing/spam detector, and UDP flood mitigation function), we define one policy for each host in the network. The heavy hitter detector, for example, deploys a per-source IP counter that is incremented for every new SYN packet and starts dropping packets when the counter exceeds a threshold (Table 4). For the functions that define policies between communicating pairs of hosts or on flows, *e.g.,* FTP monitor and DNS amplification mitigation functions, we run the same datacenter trace as above to find the communicating pairs and matching flows and define one policy for each. For functions that classify hosts into sets, *e.g.,* the firewall function with sets of internal and external hosts, we randomly assign each host to a set. Finally, for counter bounds, we draw random samples from the uniform distribution on the set of possible values.

We measure the verification time for various functions, properties, and network scales for packet-less and packet-based models. Note that increasing the network size in this manner results in larger models. For each experiment, we run 20 repeated trials, each trial with a time budget of 1,000 *sec.*, *i.e.,* we stop the verification process when the verification time exceeds 1,000 *sec*. We give a brief overview of our packet-based baseline below before presenting our results.

Packet-based transition systems as the baseline: A powerful technique for scaling static verification is slicing the network into a set of *equivalence classes* (ECs) of packets [33, 34, 43]. Each EC is a set of packets that always experience the same forwarding actions throughout the network [43]. As shown in prior work, this approach can be extended to model dynamic networks [35]. To do so, one can detect a program's ECs using verifiers that classify packets into ECs [43].

Rule	Init	Priority	Match	Action
R_0	true	100	(protocol=UDP) & (source IP=A) & ($c_0 < X$)	send()
R_1	true	100	(protocol=UDP) & (source IP=A) & ($X \leq c_0 < m$)	drop()

Table 2: A UDP flood mitigation function.

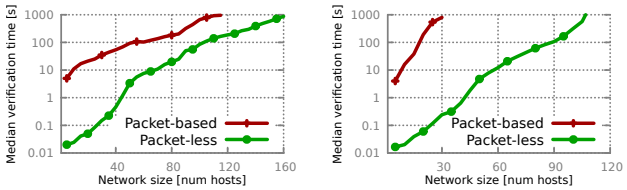
Alternatively, and similar to Kinetic [35], the programmer may be tasked with providing ECs and their transition systems.

In the packet-based transition system, each *node* represents a *state* of the network, *i.e.,* the set of ECs in that state. For instance, for the IDPS in Figure 1, in the initial state, two ECs exist in the network: EC_0 that includes all packets destined to I and with port number 2222 and EC_1 that includes all remaining packets. *Transitions* are the events that change the state of the network, *e.g.,* receiving packets from the ECs whose forwarding actions update the network's forwarding behavior. In the example above, receiving packets from EC_0 transitions the system into another state in which three different ECs exist: EC_2 that includes all packets from I that are destined to F, EC_3 that includes all packets not included in EC_2 that are sent to I with port number 2222, and EC_4 that includes all packets not included in EC_2 and EC_3.

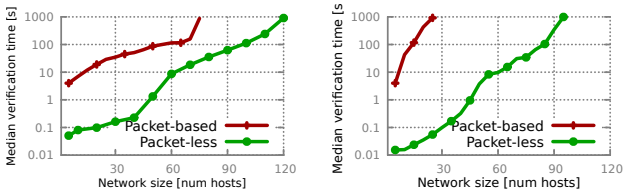
We implement this packet-based model as our baseline. For classifying packets into ECs in this baseline, we use the heuristic developed in VeriFlow [43]. Despite their similarities, the packet-based model and Kinetic [35] have a few key differences: the greater expressiveness of our programming abstraction (*e.g.,* to allow for matching on shared packet counters) increases the difficulty of the verification problem. Plus, in contrast to Kinetic that requires operators to provide the state machine as an input, the packet-based model automatically generates these from the rules written on our one-big-switch abstraction.[7] Thus, despite their conceptual similarities, we refrain from calling our baseline Kinetic.

Our results demonstrate that the packet-less model is faster than the packet-based model for different categories of liveness properties such as "leads-to" (*e.g.,* host A sending traffic to host B leads to A being blocked) and "eventually" (*e.g.,* host A is eventually blocked). Figure 5 (a) and (b) show examples for a UDP flood mitigation function (Table 2) which deploys a per-source IP counter that is incremented for every UDP packet and starts dropping packets when the counter exceeds a threshold. Verifying a leads-to property—host A sending more packets than a threshold leads to it being blocked—in a network with 100 hosts, for example, is 7.8× faster with the packet-less model than the packet-based model (85 vs. 667 *sec.*). In 1,000 *sec.*, the packet-less model verifies a different liveness property, "eventually" (A is eventually blocked), for networks that are 3.5× larger than those verifiable with the packet-based model (105 hosts *vs.* 30). By reducing the size of the state machine, the packet-less model also improves the verification time for safety properties. Figure 5 (c) and (d) show examples for an "absence" property (host A is never reachable) and "universality" (A is always reachable), respectively. Similarly, verifying the same liveness and safety properties, *e.g.,* "eventually" and "universality", in a stateful firewall (Table 13) is, respectively, 3.3× and 4.9× faster with the packet-less model than the packet-based model in a 30-host network (figures not shown).

[7] In Kinetic, a knowledgeable operator can conceivably provide compact state machines, smaller than the packet-based model, and hence experience lower verification times.



(a) Leads-to (liveness) (b) Eventually (liveness)



(c) Absence (safety) (d) Universality (safety)

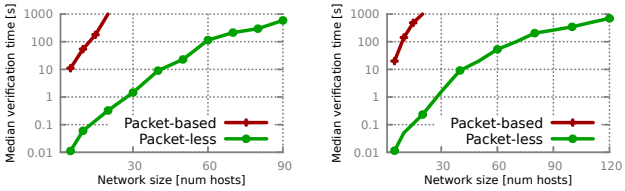
Figure 5: Packet-less verification reduces the verification time of different properties for a flood mitigation function.

We observe that greater degrees of state sharing across rules (*e.g.,* counters shared by multiple rules) result in slower verification for both approaches, but the performance degradation is more pronounced for the packet-based model, *e.g.,* for a rate limiter (Table 5), in 1,000 *sec.*, we can verify an "eventually" property in networks with up to 90 hosts with the packet-less model (*v.s.* 105-host networks for the UDP flood detection application above), and for networks with at most 15 hosts with the packet-based model (*v.s.* 30 hosts for the UDP flood detection application above). Figure 6 shows the results for a liveness and a safety property for this function. The packet-less model can verify them for network 6× larger than the packet-based model, and even for small-scale networks, it is at least two orders of magnitude faster.

In addition to testing the scalability with respect to the network (and consequently the model) size, we also scale properties. Verification time is a function of the size of the transition system and the property formula, *e.g.,* there exists an LTL model-checking algorithm whose running time depends linearly on the size of the transition system and exponentially

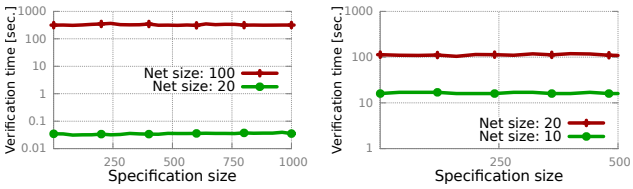
Property	LTL specification [17, 21]	Meaning and example
Leads-to (a.k.a Response)	$\mathtt{G}(P \rightarrow \mathtt{F}S)$	P must always be followed by S, a host showing a malicious activity must always be followed by the IDPS blocking the host.
Universality	$\mathtt{G}P$	P always holds, *e.g.,* A is always blocked.
Absence	$\mathtt{G}\neg P$	P never holds, *e.g.,* A can never send traffic to B.
Eventually (a.k.a Existence)	$\mathtt{F}P$	P eventually happens, *e.g.,* A can eventually reach B.
Precedence	$\mathtt{F}P \rightarrow (\neg P \mathtt{U}(S \wedge \neg P))$	P must always be proceeded by S, *e.g.,* blocking a host must always be proceeded by the host exhibiting some malicious activity.

Table 3: List of properties verified.



(a) Eventually (b) Universality

Figure 6: Verification times for a rate limiter.



(a) Packet-less (b) Packet-based

Figure 7: The property size does not impact the verification time.

on the length of the LTL formula [20]. Although in practice, the structure size is usually the dominant factor in determining the verification time, in our approach, the size of property formulas can potentially be large and increase the verification time.[8] To test the sensitivity of the verification time to the property size, we first construct atomic propositions for reachability of randomly selected (with replacement) hosts, *e.g.,* ($\mathtt{dst}$=A,$\mathtt{send}(A)$). Then, we use logical operations to build formulas of various sizes out of these atomic propositions, as explained in §3, *e.g.,* ($\mathtt{dst}$=A,$\mathtt{send}(A)$)$\wedge$($\mathtt{dst}$=B,$\mathtt{send}(B)$) for a property with size 2. Finally, we use these assertions to construct the liveness and safety properties listed in Table 3. We observe that scaling the property size does not affect the verification time of either approach. For the same UDP flood mitigation function as above, in a network with 200 hosts, changing the property size from 1 to 200, results in a standard deviation of less than 7.2. This holds even for smaller networks where the relative impact of the property size is expected to be greater. Figure 7 shows two examples for verifying $\mathtt{F}(\wedge_{h \in S}(\mathtt{dst}{=}h, \mathtt{send}(h))$, *i.e.,* eventually, all the hosts in the set S will become reachable, for small networks with 10, 20, and 100 hosts.

[8] Recall that in a packet-less model, we also need to express properties in terms of Boolean formulas on rules (§3.3.2).

5 Related Work

Static network verifiers [33,34,39,42,43,61] verify various aspects of reachability invariants such as loop-freedom and lack of blackholes on a snapshot of the network. More recently, reachability verification and enforcement techniques are extended to incorporate degrees of dynamism [10, 22, 26, 40, 48, 60], *e.g.,* with failures and policy changes [26, 27, 29, 31, 38, 52], with mutable data planes [48], and with focusing on controllers instead of snapshots of the data plane [10, 11, 22, 60]. However, the targeted properties in all these proposals are safety invariants such as reachability and loop-freedom. Our focus on verifying a computationally more complex category of properties (liveness) drives our novel packet-less model, which is distinct from prior models. Plus, some of the assumptions in prior work restrict the set of network functions that they can verify. VMN, for example, models network functions in which (a) flows are independent, and (b) forwarding is not affected by transaction orderings. We find that many crucial network functions such as IDPS and Trojans detectors [16] do not possess those properties.

We share the goal of designing verifiable programming languages with VeriCon [9], FlowLog [46], and Kinetic [35], but since our programs are compiled to P4 programs (instead of the OpenFlow rules that VeriCon's CSDN language, FlowLog, and Kinetic programs are compiled to), we are able to express programs that these frameworks cannot, *e.g.,* programs with multiple rules that share counters. Plus, VeriCon's use of first-order logic makes it infeasible to specify dynamic properties such as liveness. Finally, VeriCon, FlowLog, and Kinetic are packet-based. Kinetic, for example, extends the Pyretic controller [44] to add support for the verification of dynamic networks based on packet equivalent classes [35]. In Kinetic, the programmer needs to specify "located packet equivalence classes" (LPECs), maximal regions of the flow space (*e.g.,* packets with a given source IP) that experience the same forwarding behavior in each state, and their associated finite state machines (FSMs) that encode the handling of LPECs. We show experimentally and theoretically that compared to packet-based approaches deployed in these proposals, our packet-less approach results in faster verification.

Many recent projects in network verification leverage classical concepts of model checking to control the state explo-

Rule	Init	Priority	Match	Action
R_0	true	100	(TCP flag=SYN) & (source IP=A) & $(0 \leq c_0 < X)$	send()
R_1	true	100	(TCP flag=SYN) & (source IP=A) & $(X \leq c_0 < m)$	drop()

Table 4: A heavy-hitter detection function.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & $(0 \leq c_0 < v_1)$	send(port=1)
R_1	true	100	(source IP=A) & $(v_1 \leq c_0 > v_2)$	send(port=2)
R_2	true	100	(source IP=A) & $(v_2 \leq c_0 < v_3)$	send(port=3)
R_3	true	100	(source IP=A) & $(v_3 \leq c_0 < v_4)$	send(port=4)
R_4	true	100	(source IP=A) & $(v_4 \leq c_0 < v_5)$	send(port=5)
R_5	true	100	(source IP=A) & $(v_5 \leq c_0 < m)$	drop()

Table 5: A simple rate limiter.

sion challenge in verifying dynamic systems such as slicing [48, 50], symbolic execution [48, 57, 63], and abstraction refinement [53]. Our work shares similarities with these in terms of the high-level techniques for scaling verification (*e.g.,* we also use symbolic modeling and abstraction). As our targeted properties (liveness vs. safety invariants) are different, however, our application of these techniques diverges from existing works, *e.g.,* we deploy symbolic representation not to abstract packet header fields (as NoD does for verifying reachability invariants in dynamic networks [40]), but to abstract away packets altogether. Works on verifying networks via testing [24] and simulations [25, 51] are complementary to our approach. Via applying model checking, we strive to provide a fully *automatic* verifier that, unlike testing and simulation, searches the state space of our abstraction *exhaustively*.

6 Limitations and Future Work

The focus of our work is on functional correctness; this leaves out large sets of functions and properties, including those focused on path and traffic engineering properties (is a path congested? is the load balanced across multi-paths? *etc.*). Our one-big-switch abstraction is not suited for programming such functions. Plus, while a familiar abstraction to operators, the one-big-switch abstraction is relatively low-level. An interesting direction for future research is developing higher-level abstractions amenable to efficient liveness verification.

Our verifier is not a "full-stack" one; it is not designed to verify low-level properties such as memory safety and crash freedom that tools such as Vigor [62] and VigNAT [63][9] can verify and does not verify the compiled P4 code. Consequently, tools such as compiler verifiers (such as p4v [39]), P4 debuggers (*e.g.,* Vera [56])), and testers are still essential to guarantee the faithful implementation of our verified abstractions, *e.g.,* to detect and debug switch firmware and P4 compiler bugs. Finally, while our packet-less modeling improves the verification time compared to a packet-based model and enables the verification of complex properties such as liveness ones, the absolute verification times remain high for large-scale networks (note the logarithmic scale in figures). Further reducing the verification complexity of stateful functions remains an open challenge.

[9] VigNAT [63] partitions programs into stateful and stateless components. While the stateless component is verified automatically via applying symbolic model checking, the verification of the stateful part is done via theorem proving and requires human assistance. In Vigor [62], the function code that cannot be symbolically executed is stored in a specialized library and verified by experts using theorem proving (*i.e.,* writing proofs).

7 Conclusion

Modern networks rely on a variety of stateful network functions to implement rich policies. Correct operation of such networks relies on ensuring that they support key liveness properties. Unfortunately, despite exciting recent work on network verification, no existing approach is practical for, or applicable to, validating liveness. We take a top-down approach to this problem by first designing a new programming model built with verification in mind. It offers natural extensions to the convenient one-big-switch abstraction and allows decomposition of different network functions. We develop a novel encoding of these programs under dynamic events such as network state changes using Boolean formulas that can capture rich semantics (*e.g.,* counters) while also ensuring that the encoding remains bounded-size and amenable to fast liveness verification. We develop a compiler that translates our programs into those runnable on modern hardware. Our evaluation shows that the programming model can succinctly express a variety of functions, our compilation is fast, our encoding is compact and orders of magnitude more scalable to verify than naive encodings, *i.e.,* it results in substantial verification speedup.

Acknowledgments: We thank our shepherd, Meg Walraed-Sullivan, Nate Foster, Loris D'Antoni, Aws Albarghouthi, and the anonymous reviewers for their valuable comments. We gratefully acknowledge the support of the National Science Foundation through grant CNS 1910821.

References

[1] Floodlight forwarding module. https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/9142336/Forwarding.

[2] OpenFlow. openflow.org.

[3] P4 behavioral model repository. https://github.com/p4lang/behavioral-model.

[4] Project Floodlight. http://www.projectfloodlight.org/floodlight/.

[5] AL-SHABIBI, A., DE LEENHEER, M., GEROLA, M., KOSHIBE, A., PARULKAR, G., SALVADORI, E., AND SNOW, B. OpenVirteX: Make your virtual SDNs programmable. In *HotSDN* (2014).

[6] ALPERN, B., AND SCHNEIDER, F. B. Recognizing safety and liveness. *Distributed Computing 2*, 3 (1987).

[7] ARASHLOO, M. T., KORAL, Y., GREENBERG, M., REXFORD, J., AND WALKER, D. SNAP: Stateful network-wide abstractions for packet processing. In *SIGCOMM* (2016).

[8] BAIER, C., AND KATOEN, J.-P. *Principles of model checking*. 2008.

[9] BALL, T., BJØRNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., AND VALADARSKY, A. VeriCon: Towards verifying controller programs in software-defined networks. In *Sigplan Notices* (2014), vol. 49.

[10] BECKETT, R., GUPTA, A., MAHAJAN, R., AND WALKER, D. A general approach to network configuration verification. In *SIGCOMM* (2017).

[11] BECKETT, R., GUPTA, A., MAHAJAN, R., AND WALKER, D. Control plane compression. In *SIGCOMM* (2018).

[12] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *IMC* (2010).

[13] BORDERS, K., SPRINGER, J., AND BURNSIDE, M. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security Symposium* (2012).

[14] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., ET AL. P4: Programming protocol-independent packet processors. *SIGCOMM CCR 44*, 3 (2014), 87–95.

[15] BURCH, J. R., CLARKE, E. M., MCMILLAN, K. L., DILL, D. L., AND HWANG, L. J. Symbolic model checking: 10ˆ20 states and beyond. *Inf. Comput. 98*, 2 (1992).

[16] CARLI, L. D., SOMMER, R., AND JHA, S. Beyond pattern matching: A concurrency model for stateful deep packet inspection. In *CCS* (2014).

[17] CHECHIK, M., AND PAUN, D. O. Events in property patterns. In *SPIN* (1999).

[18] CIMATTI, A., CLARKE, E. M., GIUNCHIGLIA, F., AND ROVERI, M. NUSMV: A new symbolic model verifier. In *CAV* (1999).

[19] CLARKE, E. M., GRUMBERG, O., AND PELED, D. *Model Checking*. MIT press, 1999.

[20] CLARKE, E. M., HENZINGER, T. A., VEITH, H., AND BLOEM, R. P. *Handbook of model checking*. 2016.

[21] DWYER, M. B., AVRUNIN, G. S., AND CORBETT, J. C. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering* (1999).

[22] FAYAZ, S. K., SHARMA, T., FOGEL, A., MAHAJAN, R., MILLSTEIN, T. D., SEKAR, V., AND VARGHESE, G. Efficient network reachability analysis using a succinct control plane representation. In *OSDI* (2016).

[23] FAYAZ, S. K., TOBIOKA, Y., SEKAR, V., AND BAILEY, M. Bohatei: Flexible and elastic DDoS defense. In *USENIX Security Symposium* (2015).

[24] FAYAZ, S. K., YU, T., TOBIOKA, Y., CHAKI, S., AND SEKAR, V. BUZZ: Testing context-dependent policies in stateful networks. In *NSDI* (2016).

[25] FOGEL, A., FUNG, S., PEDROSA, L., WALRAED-SULLIVAN, M., GOVINDAN, R., MAHAJAN, R., AND MILLSTEIN, T. D. A general approach to network configuration analysis. In *NSDI* (2015).

[26] GEMBER-JACOBSON, A., VISWANATHAN, R., AKELLA, A., AND MAHAJAN, R. Fast control plane analysis using an abstract representation. In *SIGCOMM* (2016).

[27] GHORBANI, S., AND CAESAR, M. Walk the line: Consistent network updates with bandwidth guarantees. In *HotSDN* (2012).

[28] GHORBANI, S., SCHLESINGER, C., MONACO, M., KELLER, E., CAESAR, M., REXFORD, J., AND WALKER, D. Transparent, live migration of a software-defined network. In *SoCC* (2014).

[29] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving high utilization with software-driven WAN. In *SIGCOMM* (2013).

[30] JAYARAMAN, K., BJØRNER, N., PADHYE, J., AGRAWAL, A., BHARGAVA, A., BISSONNETTE, P.-A. C., FOSTER, S., HELWER, A., KASTEN, M., LEE, I., ET AL. Validating datacenters at scale. In *SIGCOMM* (2019).

[31] JOHN, J. P., KATZ-BASSETT, E., KRISHNAMURTHY, A., ANDERSON, T., AND VENKATARAMANI, A. Consensus routing: The Internet as a distributed system. In *NSDI* (2008).

[32] KANG, N., LIU, Z., REXFORD, J., AND WALKER, D. Optimizing the one big switch abstraction in software defined networks. In *CoNEXT* (2013).

[33] KAZEMIAN, P., CHAN, M., ZENG, H., VARGHESE, G., MCKEOWN, N., AND WHYTE, S. Real time network policy checking using header space analysis. In *NSDI* (2013).

[34] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header space analysis: Static checking for networks. In *NSDI* (2012).

[35] KIM, H., REICH, J., GUPTA, A., SHAHBAZ, M., FEAMSTER, N., AND CLARK, R. J. Kinetic: Verifiable dynamic network control. In *NSDI* (2015).

[36] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., INGRAM, P., JACKSON, E. J., ET AL. Network virtualization in multi-tenant datacenters. In *NSDI* (2014).

[37] LAMPORT, L. What good is temporal logic? In *IFIP congress* (1983), vol. 83.

[38] LIU, H. H., WU, X., ZHANG, M., YUAN, L., WATTENHOFER, R., AND MALTZ, D. zUpdate: Updating data center networks with zero loss. In *SIGCOMM* (2013).

[39] LIU, J., HALLAHAN, W., SCHLESINGER, C., SHARIF, M., LEE, J., SOULÉ, R., WANG, H., CAŞCAVAL, C., MCKEOWN, N., AND FOSTER, N. P4v: Practical verification for programmable data planes. In *SIGCOMM* (2018).

[40] LOPES, N. P., BJØRNER, N., GODEFROID, P., JAYARAMAN, K., AND VARGHESE, G. Checking beliefs in dynamic networks. In *NSDI* (2015).

[41] LUDWIG, A., ROST, M., FOUCARD, D., AND SCHMID, S. Good network updates for bad packets: Waypoint enforcement beyond destination-based routing policies. In *HotNets* (2014).

[42] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P., AND KING, S. T. Debugging the data plane with Anteater. In *SIGCOMM* (2011).

[43] MAI, H., KHURSHID, A., AGARWAL, R., CAESAR, M., GODFREY, P., AND KING, S. T. VeriFlow: Verifying network-wide invariants in real time. In *NSDI* (2013).

[44] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing software defined networks. In *NSDI* (2013).

[45] MOSHREF, M., BHARGAVA, A., GUPTA, A., YU, M., AND GOVINDAN, R. Flow-level state transition as a new switch primitive for SDN. In *HotSDN* (2014).

[46] NELSON, T., FERGUSON, A. D., SCHEER, M. J., AND KRISHNAMURTHI, S. Tierless programming and reasoning for software-defined networks. In *NSDI* (2014).

[47] OWICKI, S., AND LAMPORT, L. Proving liveness properties of concurrent programs. *TOPLAS 4*, 3 (1982).

[48] PANDA, A., LAHAV, O., ARGYRAKI, K. J., SAGIV, M., AND SHENKER, S. Verifying reachability in networks with mutable datapaths. In *NSDI* (2017).

[49] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E. J., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., ET AL. The design and implementation of Open vSwitch. In *NSDI* (2015).

[50] PLOTKIN, G. D., BJØRNER, N., LOPES, N. P., RYBALCHENKO, A., AND VARGHESE, G. Scaling network verification using symmetry and surgery. In *SIGPLAN Notices* (2016).

[51] QUOITIN, B., AND UHLIG, S. Modeling the routing of an autonomous system with C-BGP. *IEEE Network 19*, 6 (2005).

[52] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for network update. In *SIGCOMM* (2012).

[53] RYZHYK, L., BJØRNER, N., CANINI, M., JEANNIN, J.-B., SCHLESINGER, C., TERRY, D. B., AND VARGHESE, G. Correct by construction networks using stepwise refinement. In *NSDI* (2017).

[54] 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 *SIGCOMM* (2016).

[55] SIVARAMAN, A., KIM, C., KRISHNAMOORTHY, R., DIXIT, A., AND BUDIU, M. DC.P4: Programming the forwarding plane of a data-center switch. In *SOSR* (2015).

[56] STOENESCU, R., DUMITRESCU, D., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Debugging P4 programs with Vera. In *SIGCOMM* (2018).

[57] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Symnet: Scalable symbolic execution for modern networks. In *SIGCOMM* (2016).

[58] SUBRAMANIAN, K., D'ANTONI, L., AND AKELLA, A. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *POPL* (2017).

[59] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM* (2013).

[60] WEITZ, K., WOOS, D., TORLAK, E., ERNST, M. D., KRISHNAMURTHY, A., AND TATLOCK, Z. Bagpipe: Verified BGP configuration checking. In *OOPSLA* (2016).

[61] XIE, G. G., ZHAN, J., MALTZ, D. A., ZHANG, H., GREENBERG, A., HJALMTYSSON, G., AND REXFORD, J. On static reachability analysis of IP networks. In *INFOCOM* (2005).

[62] ZAOSTROVNYKH, A., PIRELLI, S., IYER, R., RIZZO, M., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. Verifying software network functions with no verification expertise. In *SOSP* (2019).

[63] ZAOSTROVNYKH, A., PIRELLI, S., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. A formally verified NAT. In *SIGCOMM* (2017).

8 Appendix

Table 1 lists the functions that can be represented in our language. Below we write these functions on our one-big-switch abstraction.

Simple counter: A packet counter for every source, destination IP address pair (Table 6).

Port knocking: Open a certain port O by attempting to open a connection on a prespecified closed port K (Table 7).

Simple firewall: Allows traffic between certain source, destination MAC addresses (Table 8).

IP rewrite: Rewrite IP address of all traffic coming from and directed to a particular IP address (Table 9).

Simple rate limiter: There are multiple implementations of this function. The first example (Table 5) assumes different ports are capable of sending traffic at different rates. It uses per-source IP counters to decide the traffic rate.

The second example (Table 10) uses counters on subsets of traffic to decide what traffic will be forwarded normally, redirected to a rate limiter, or dropped.

Firewall/ACL: Allows or Blocks traffic based on certain packet fields (Table 11).

Phishing/Spam detection: A per-MTA (Mail Transfer Agent) counter to detect MTAs that send a large amount of mail (Table 12).

Simple stateful firewall: A firewall that allows only the traffic whose connection was initiated by a host in I, where I is the set of departmental addresses (Table 13).

FTP monitor: Allows traffic on data port only if it received a signal on the control port (Table 14).

Heavy-hitter detection: A per-source IP counter that is incremented for every new SYN. It starts dropping packets when the counter exceeds a threshold value (Table 4).

Super spreader detection: Similar to heavy-hitter detection, the counter is incremented for every SYN. But it is also decremented for every FIN.

Sampling based on the flow size: This can be done using two tables. The first table (Table 15) uses counters to classify the flow size into three categories - small, medium, and large. It adds this metadata information into the packet (*e.g.,* using the QoS field). The second table (Table 16) samples packets based on its flow size, using its own counters.

Elephant flow detection: This is similar to sampling based on flow size, where flows of large size are elephant flows.

DNS amplification mitigation: Allows DNS reply (source port=53) to a particular IP only if it receives a DNS request/query from that IP (Table 17).

UDP flood mitigation: A per-source IP counter that is incremented for every UDP packet. It starts dropping packets when the counter exceeds a threshold value (Table 2).

Application chaining: Our language can represent non-linear chaining of applications. For example, consider a system that wants to rate-limit phishing and heavy-hitter traffic. This can be represented in our language using three tables.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & (destination IP=B) & ($0 \leq c_0 <$ m)	send()

Table 6: Simple counter.

Rule	Init	Priority	Match	Action
R_0	true	100	(source MAC=M) & (destination port=K)	add(R_1),add(R_2)
R_1	false	100	(source MAC=M) & (destination port=O)	send()
R_2	false	100	(destination MAC=M) & (destination port=O)	send()

Table 7: Port knocking.

Rule	Init	Priority	Match	Action
R_0	true	100	(source MAC=A) & (destination MAC=B)	send()
R_1	true	100	(source MAC=B) & (destination MAC=A)	send()

Table 8: Simple firewall.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A)	modify(source IP=X),send()
R_1	true	100	(destination IP=X)	modify(destination IP=A),send()

Table 9: IP rewrite.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & ($0 \leq c_0 < v_1$)	send()
R_1	true	100	(source IP=A) & ($v_1 \leq c_0 < v_2$)	send(rate limiter)
R_2	true	100	(source IP=A) & ($v_2 \leq c_0 <$ m)	drop()

Table 10: Simple rate limiter 2.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A1) & (destination IP=B1) & (source MAC=M1) & (destination MAC=N1) & (source port=P1) & (destination port=Q1)	send()
R_1	true	100	(source IP=A2) & (destination IP=B2) & (source MAC=M2) & (destination MAC=N2) & (source port=P2) & (destination port=Q2)	drop()

Table 11: Floodlight firewall/ACL.

Rule	Init	Priority	Match	Action
R_0	true	100	SMTP.MTA=A	add(R_1),add(R_2),delete(R_0),send()
R_1	false	100	(SMTP.MTA=A) & $0 \leq c_0 <$ X	send()
R_2	false	100	(SMTP.MTA=A) & X $\leq c_0 <$ m	drop()

Table 12: Phishing/spam detection.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=I) & (destination IP=E)	add(R_1),delete(R_0),send()
R_1	false	100	(destination IP=I) & (source IP=E)	send()
R_2	true	50	source IP=I	drop()

Table 13: Simple stateful firewall.

The first table does phishing detection (Table 12), the second one does heavy-hitter detection (Table 4), and third one does rate-limiting (Table 5). The first table sends suspicious traffic to the third table (instead of $drop()$) and normal traffic to the second table (instead of $send()$). The second table sends suspicious traffic to the third table (instead of $drop()$) and forwards

Rule	Init	Priority	Match	Action
R_0	true	100	(destination port=$port_{control}$) & (source IP=A) & (destination IP=B)	add(R_1),delete(R_0),send()
R_1	false	100	(source port=$port_{data}$) & (source IP=B) & (destination IP=A)	send()

Table 14: FTP monitor.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & (destination IP=B) & ($0 \leq c_0 < v_1$)	modify(flow=small),send(Sampler Table 2)
R_1	true	100	(source IP=A) & (destination IP=B) & ($v_1 \leq c_0 < v_2$)	modify(flow=medium),send(Sampler Table 2)
R_2	true	100	(source IP=A) & (destination IP=B) & ($v_2 \leq c_0 <$ m)	modify(flow=large),send(Sampler Table 2)

Table 15: Sampling based on the flow size - Sampler Table 1.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & (destination IP=B) & (flow=small) & $(c_0 = 5)$	$c_0 = 0$,send()
R_1	true	100	(source IP=A) & (destination IP=B) & (flow=medium) & $(c_0 = 500)$	$c_0 = 0$,send()
R_2	true	100	(source IP=A) & (destination IP=B) & (flow=large) & $(c_0 = 50000)$	$c_0 = 0$,send()

Table 16: Sampling based on the flow size - Sampler Table 2.

Rule	Init	Priority	Match	Action
R_0	true	100	(source IP=A) & (destination IP=B) & (destination port=53)	delete(R_0),add(R_2),add(R_3),send()
R_1	true	10	source port=53	drop()
R_2	false	100	(source IP=A) & (destination IP=B) & (destination port=53)	send()
R_3	false	100	(source IP=A) & (destination IP=B) & (source port=53)	send()

Table 17: DNS amplification mitigation.

non-suspicious traffic normally using *send*() action.