Symbolic Execution for Network Functions with Time-Driven Logic

Paper # 23, 6 pages

ABSTRACT

Network functions such as stateful firewall, rate limiter, intrusion detection systems, etc. consist of both event-based logic as well as time-driven logic. But most of the existing NF verification tools verify only packet arrival based event logic. We propose that NF modeling language should include time-driven logic to perform complete verification and avoid false positive/negative results. Thus, we define primitives to express time-driven logic in NF modeling language and further develop a symbolic execution engine NF-SE that can verify such logic for multiple packets. We demonstrate its usefulness and correctness for a stateful firewall example and discuss its scalability in terms of optimization for a large number of input packets.

1 INTRODUCTION

Symbolic execution on network functions (NFs) can statically explore all possible runtime execution paths and generate concrete input test traffic to test all possible valid execution paths. It plays an important role in various network management applications, including network verification [20], configuration validation, and network testing [12, 21, 22]. Existing NF symbolic execution solutions usually use a domain specific language (DSL) to describe NF behaviors (i.e., model) and inject symbolic packets to execute the NF model [20–22].

The expressiveness of a DSL decides whether an NF model can be used to represent its actual implementation. Most existing solutions assume NF logic is triggered by the packet arrival events, however, we find that a variety of stateful NFs contains another kind of logic — *time-driven logic*, where NF states depends on the elapsed time. Without considering such kind of logic, existing symbolic execution engines (SEE) actually execute on a snapshot of the NF, and could lead to false negative/positive results. For example, a stateful firewall with state expiration would reject long-term inactive flow, but the current SEE will predict the flow to pass through.

In this work, our goal is to *complement the existing NF DSL with time-driven logic and enhance symbolic execution engine to execute this kind of logic.* We first study the implementation and application of time-driven logic in existing NFs, and summarize three primitives for expressing such logic. These include a data structure for time value, a function call to get the present time, and a timer to schedule a future event.

Then we build a symbolic execution engine for NF models in the enhanced DSL. The SEE processes the packet-driven logic in NF model by exploring all valid execution paths. For time driven logic, it adds timing constraints for packet arrival and timer events.

We name our solution NF-SE, and prototype it. According to our preliminary evaluation, NF-SE can evaluate NFs with state expiration in acceptable time for multiple packets. We discuss future works, including optimizations such as accelerating NFs with counter attenuation logic, and application scenarios such as network service protocol verification, user flow statistics behavior verification, and network end-to-end verification.

2 BACKGROUND AND MOTIVATION

Symbolic Execution on NFs is important in NF verification and network verification, and existing solutions need to consider time-driven logic for a more precise result.

2.1 NF Symbolic Execution

In individual NF verification, exploring all the execution paths helps in finding out implementation or configuration errors [10, 21]; in network-wide NF verification [12, 20], combining individual NF execution paths with the network topology information helps to reason about network-wide behaviors (e.g., reachability, isolation, etc).

Symbolic execution is a widely-used method to infer NF execution paths. It statically executes instructions in NF code using symbolic packets, forks for each conditional branch, keeps a track of constraints on each branch, and outputs concrete input traffic for each valid path satisfying all constraints on that path. Hence, it generates input test packets which explore all the valid execution paths in an NF.

Applying symbolic execution on general NF code (e.g., Klee [8]) would cause (unnecessary) path explosion [1], causing intolerable execution times and network irrelevant path constraints. So, symbolic execution is usually performed on NF models which are specified by a domain specific language, called a NF modelling language. SymNet [22], BUZZ [12], VMN [20], and Vera [21] are examples of such symbolic execution engines with domain specific languages.

[1]For example, `sprintf(ip_str, "%d.%d.%d.%d", ip >> 24 % 256, ip >> 16 % 256, ip >> 8 % 256, ip % 256)` would causes 3^4 branches. Because each 8-bit segment can be 1 or 2 or 3 digits when printed to string in decimal.

```
state = CLOSE
foreach packet {
  if (syn packet){ state=OPEN; pass }
  else if(state == OPEN){ pass; }
  else{ drop; } }
```

Figure 1: Pseudocode of a stateful firewall

2.2 Time-Driven Logic in NFs

The correctness of the symbolic execution results depends on it's fidelity whether the NF behavior model represents the real NF program (i.e., the code). In most of the existing NF verification work, it is assumed that NF logic is driven by packet arrival events, i.e., the arrival of a packet triggers a series of instructions that processes the packet and updates the NF's internal states. But we observe that there exists another kind of logic — time-driven logic — in NF programs.

Time-driven logic includes the logic that is triggered by timing events and the logic where timing information is utilized for packet processing. We formally define the basic programming primitives for time-driven logic in §3. In practice, many NFs contain time-driven logic; for example, an NAT would store established address mapping between internal addresses and external addresses and expire the states after some time, a rate limiter (e.g., leaky bucket algorithm) needs to accumulate "budget" with time and consume the budget by sending packets, and a stateful firewall would preserve the information of valid flows and expire the information after a threshold time.

Example. We use a simplified stateful firewall as a running example in this paper. Figure 1 shows the basic logic of the firewall — the SYN packet of a TCP flow punches a hole in the firewall, and all subsequent packets are allowed; without the SYN packet, any other packets are dropped by the firewall. In the implementation, a state variable "state" is used to record whether a SYN packet has arrived, and has a value either "CLOSE" or "OPEN".

While in most network verification solutions, a stateful firewall is represented in this way (Figure 1), the practical implementation usually contains another kind of logic — the state would expire after a certain amount of time. If no packet has arrived for some threshold time, the state would return to "CLOSE", meaning further communication requires a preceeding SYN packet to again reopen the connection. Figure 2 shows such an implementation: a timer would be triggered periodically to check the state refreshment, if no state refreshment happens for a time threshold, the state is reset to "CLOSE".

Without modeling such time-driven logic in NF models, the symbolic execution results could possibly mismatch the actual NF behaviors. There may be false negatives (i.e., reporting unsafe behaviors as safe): in the stateful firewall

```
state = CLOSE; timer(30s, handler);
void handler() { % Time-Driven Logic
  if (state == OPEN &&
    currTime()-Modified >=30)
    {state=CLOSE; timer(30s, handler);}}
foreach packet {
  if (syn packet){ state=OPEN; pass;
    Modified=currTime();} % Time-Driven
  else if(state == OPEN){ pass;
    Modified=currTime();} % Time-Driven
  else{ drop; } }
```

Figure 2: Psedudocode of a stateful firewall with state expiration

example (Figure 2), the actual firewall may stop a flow due to its long-term inactiveness, but due to lack of time-driven logic modelling the symbolic execution engine would not verify such logic and report pass for such flows. There may also be false positives (i.e., report safe behaviors as unsafe); for example in a SYN flood detection NF, SYN packets are recorded in a counter, and the counter attenuates with time, if the attenuation time period (time-driven logic) is not considered in the verification, all SYN packets in a long-time period would be falsely reported as bursty SYN flood attack.

Goal. Thus, our goal in this paper is to complement NF modeling language with *time-driven logic*, build a symbolic execution engine for NFs with time-driven logic, and show a few applications where such a complement improves NF verification results.

3 MODELING TIME-DRIVEN LOGIC

We summarize primitives to express time-driven logic, and add them to the NF modeling language.

3.1 NF Modeling Language

We summarize NF modeling language from several existing solutions [6, 20, 22] , and its syntax shows in Figure 3. This language has the following features :

- *Syntax.* The language contains variables and constants as basic operands, and commonly used operators such as arithmetic (+, −, ∗, /, %), relational (>, <, ! =, ==), boolean (&&, ||, !), bitwise (&, |, <<, >>) and indexing ([]) operators. Operands and operators together compose expressions. An NF program consists of simple statements such as assignments and complex statements of branching (if-else-then).
- *Semantics.* In the language, the semantics of expressions follow their mathematical definitions, an assignment statement means to set the value of the lefthand symbol to be that of the righthand expression, and a branching statement means if the condition is true, execute the if branch, otherwise execute the else branch. The whole program

Basic types and expression			
const	c	::=	$(0\mid 1)^+$
header field	h	::=	$sip\mid dip\mid sport\mid dport\mid proto\mid\ldots$
state	s		
variable	var		
expression	e	::=	$c\mid h\mid s\mid var\mid e\mid Expr\_Op(e_1, e_2, \ldots)$
Predicates			
flow predicate	x_f, y_f	::=	$\epsilon\mid * \mid h = c\mid\neg x_f\mid x_f \wedge y_f\mid x_f \vee y_f$
state predicate	x_s, y_s	::=	$*\mid Rel\_Op(s, e)\mid\neg x_s\mid x_s \wedge y_s\mid x_s \vee y_s$
general predicate	x, y	::=	$Rel\_Op(e_1, e_2, \ldots)\mid\neg x\mid x \wedge y\mid x \vee y$
Policies and Statements			
flow policy	p_f, q_f	::=	$h := e\mid p_f; q_f$
state policy	p_s, q_s	::=	$s := e\mid p_s; q_s$
general policy	p, q	::=	$q := e\mid p; q$
Model			
model	$model$	::=	$stmts$
statements	$stmts$	::=	$stmt\mid stmt; stmts$
statement	$stmt$	::=	$p\mid if\mid loop$
if statement	if	::=	if $(x)\{(stmts\}$ else $\{stmts\}$

Figure 3: NF-SE language syntax (following SNAP and NetKAT[5][6])

executes each state sequentially from the beginning to the end.

- *NF Programming Abstractions.* Specifically, a few variables and expressions are summarized and defined as keywords, which expresses NF semantics. In the syntax above, all symbols derived from "header fields" are variables with special meaning, denoting correlating packet header fields; the index operator with a field (e.g., `f[sip]`) stands for parsing a packet and fetching the field; and states are set variables that are created, retrieved, and updated by flows (e.g., `counter[f]++`). These NF programming abstractions (1) simplify the NF model representation (avoiding tedious implementation) and (2) avoid path explosion in later symbolic execution (see §2).
- *Loop-freedom.* The NF-SE language does not contain loop statement (e.g., `while, for`). The reason is that symbolic execution needs to statically find all execution paths, but a loop with an unpredictable number of execution times might cause the path search to not terminate. Most of the existing network verification solutions make the same assumption [20, 22] , and many NF development frameworks use loop-free program structures (e.g., match-action table in SDN, stateful match-action table in Microsoft VFP [13]).

3.2 Adding Time-driven Logic

We studied the time-driven logic in typical NFs such as stateful firewall, rate limiter, and intrusion detection systems and summarize the following primitives to express time-driven logic.

- `timevalue` is a data structure to store time. It can be a timestamp or a time interval. In NF-SE language, `timevalue` is a used as a variable or constant.
- `currentTime()` would return the timestamp of the current time.
- `timer(TIME_INTERVAL, HANDLER, ARGS)` is a function call, which schedules the timer logic in `HANDLER` with arguments of `ARGS` at a future time `TIME_INTERVAL` from now.

Execution Model. Among the three primitives, `timevalue` and `currentTime()` are variables and a function call that can be embedded into the NF-SE language (as operands or an operator), but `timer()` needs special notation. Usually the semantics of "triggering some logic at some time" is maintained by a timing framework (e.g., callout timer in early Linux or timing wheel [23]), and the timing framework executes in parallel with the original logic. The two parallel processes usually interact with each other by operating on the shared variables, and there are three execution models :

(1) *Preemption.* Whenever the `timer()` is triggered, it interrupts the current process and preempts the control flow; after the `HANDLER` is completed, the control flow returns to original execution location before preemption.
(2) *Concurrency.* Both the `timer()` and the current process executes simultaneously; if there are critical sections (e.g., shared variables) in both processes, concurrency control mechanisms such as locking or mutex are needed.
(3) *Sequential.* The current process pauses periodically and checks whether `timer()` is triggered; if yes, the `HANDLER` executes to complete, and then the process resumes.

In NF specific domain, we have the following observations and assumptions, which leads us to choose the *sequential execution model* for NF-SE.

(1) The packet process and timing process share critical NF states. The timing process usually updates these states and indirectly influences packet processing (e.g., the "state" in Figure 1, the "token" in Figure 4).
(2) The timing process usually does not operate on packets directly but on NF states. Because packet arrival events are independent of the time elapsed, and it is difficult to execute the `HANDLER` logic on the packet once packets are not in the packet processing pipeline. [2]
(3) If an NF uses parallel execution model for timing process and packet process, we assume it has correct concurrency control on shared states. For example, in Figure 2, expiring the state by timer and checking the state of a packet should have concurrency control (e.g., locks or mutex).

By the first observation, NF-SE excludes preemption model; and by the third assumption, a correct parallel model should be logically equivalent to a sequential execution of two processes. Thus, NF-SE assumes the sequential execution of timing process and packet process, i.e., when a packet is

[2]It is possible that `timer()` triggers packet generation. We categorize this kind of logic to be a control plane message, and is not in the scope of the data plane verification tool NF-SE.

```
int token = 1000; timer(1s, handler)
void handler(){
  token=1000; timer(1s, handler);}
foreach packet{
  if( token<f[size] ) { drop; }
  else { token-=f[size]; pass}
}
```

Figure 4: Psedudocode of a rate limiter

processed, `timer()` events are temporarily masked and after the packet processing is completed, the timer events are checked and if there are triggered events, the `HANDLER` is executed. This assumption also complies with the actual implementation (e.g., P4 rate limiter [15]).

The stateful firewall example in Figure 2 follows the syntax, and we also show another example of a rate limiter (leaky bucket algorithm) (Figure 4). It maintains a token variable to record the budget to send packets. Sending packets would consume the token until token is 0 and if the token is less than the size of the packet, the packet is dropped; the token is refreshed periodically by a timer.

4 SYMBOLICALLY EXECUTE TIME-DRIVEN LOGIC

In the symbolic execution engine, we use static analysis and constraint solver Z3 to verify packet-driven logic [16]. For time-driven logic, we treat the three primitives as variables and add extra timing constraints (such as constraints of execution order in the time domain).

4.1 Packet-driven Logic

Packet arrival would trigger the NF to execute a series of instructions to process it. With the assumption of loop-freedom, the execution is unidirectional from packet input to packet output or drop.

Our symbolic execution engine, NF-SE injects symbolic packets, and statically analyzes all execution paths. NF-SE maintains a trace for instructions, which is organized as a tree; starting from the packet input, it sequentially adds each instruction to the trace: simple assignment instructions would be added to the trace linearly and sequentially (growing the tree by one child), but for a branching instruction (i.e., `if-else`) , the trace tree branches into two children — one further goes to the positive branch (recording the condition of the `if` statement) and the other goes to the negative branch (recording the negation of the condition). And finally, the static analysis would arrive at the end of the model, exploring all possible paths in the input NF.

A constraint solver solves each path and if the path's constraints are satisfiable, the solver would output an example of a symbolic packet/packets that satisfies all constraints on the valid path.

We made an optimization to delete the trace tree nodes along with its children if branching at that node creates unsatisfiable path. In this way, the static analysis would not explore along that path. This reduces the size of trace tree and thus, execution time for SMT solver to check invalid branches.

For multiple packets execution, we recursively create and execute different possible paths of trace tree depending on the number of packets required, updating the state variables and placing constraints from previous path execution, and finally output all possible execution cases of multiple packets.

4.2 Time-driven Logic

When NF-SE parses an NF model and builds the trace tree, it integrates constraints from the timing primitives. (1) In NF models, timestamps (implemented by `timevalue`) are usually used as an attribute of a variable (e.g., the timestamp when a packet is received, the timestamp when a state is updated). For a variable with a timestamp, the initial value of a timestamp is the same as the beginning time of the variable's lifetime. As NF-SE builds the trace tree, constraints on timestamp initialization are added to the path: the initialization time of a timestamp should be greater than that of its previous neighboring timestamp. For example, the 2nd packet's timestamp is always greater than the 1st packet, thus, constraint `pkt1[ts]<pkt2[ts]` is added to the path.

(2) Timestamps may appear in assignments and conditions (e.g., last modified time is within 30s in Figure 2), they are treated the same as other variables and constraints are added similarly.

(3) `currentTime()` returns a `timevalue` of the current time; NF-SE adds a declaration of a new timestamp variable to replace `currentTime()`, and similarly adds constraints that this new timestamp is greater than its previous neighboring timestamp.

(4) When `timer(TIME, HANDLER, ARGS)` is "called" to invoke an event in the future, NF-SE adds a timestamp to record the current time and a constraint (later than its previous neighboring timestamp initialization), and also records an association of the timer and handler with this timestamp (for the execution in the future).

(5) `timer(TIME, HANDLER, ARGS)` is "executed" between passes of two packet processing (assuming the sequential execution model). When NF-SE gets to a `timer()` execution, it first adds a timestamp (e.g., t1) of current time (as well as its constraints), extracts the timestamp when this timer is invoked (assume t0), and then makes two branches: one with the assumption that the event is triggered (with the constraint t1-t0>=TIME) and the other that not (with the

constraint t1-t0<TIME). The former branch would first proceed with HANDLER which is analyzed as packet processing (like (1), (2), and (3)) and then to the next packet; the latter branch proceeds to process the next packet.

(6) There may be multiple timer events after one packet is processed, and they are organized as a queue and symbolically executed similarly as (5). In some cases, the handler of a timer may invoke another timer. NF-SE adds the new invoked timer to the end of the queue. We add an assumption to avoid infinite loops — the number of recursive invocations from one timer handler to another is bounded. Several practical observations support this assumption, (1) most recursive timer events are idempotent (meaning multiple execution is equivalent to one execution; examples are state expiration); (2) variables that are periodically and monotonically changed usually have a threshold (e.g., rate limiter's token is accumulated to the burst size and does not change any more).

5 PRELIMINARY EVALUATION

Implementation and Settings. We use Antlr[1] to build a parser (247 lines of code in Java) for the NF-SE language syntax, which parses an NF model and builds a trace tree. We create a symbolic execution engine using Z3 SMT solver[9], which has 1600+ lines of code in C++.

In NF implementation, we summarize that time-driven logic usually exists in two formats: state expiration and counter attenuation. Stateful firewalls usually has state expiration, such as the firewall in Figure 2 and TCP 3-way handshake checking firewall and TCP retransmission timout. Counter attenuation usually exists in rate limiters and intrusion detection systems (IDS), such as super spreader detector (SSD), heavy hitter detector (HHD), and SYN flood detector. We implement these NFs.(details in [2])

Case Study: Stateful Firewall. We show an example of verifying the NF in Figure 2. In table 1 , we list some of the possible packet traces of several execution paths, however, due to space limitation, we only show the first two packets. We can see that path 3 is the case when there is no state expiration so that the 2nd packet gets through and path 4 is the case when the state expires so that the 2nd packet is dropped. Thus, NF-SE can overcome the false negative case in §2 .

Performance. We perform our verification on a quad-core, intel i3 machine with 8 GB RAM. The results are in Figure 5, 6, 7. We get the following observations. (1) More than 96% of the time is consumed in solving constraints using Z3. (2) For each NF, the branching factor (i.e., the number of execution paths) increases exponentially with the number of input packets, while the number of constraints on each execution path increases linearly. (3) Different NFs have different branching factors based on their logic. Specifically rate limiters and stateful firewalls have smaller branching factors and acceptable performance (all execution paths can be verified within 4 packets in tens of seconds), but other IDSes are usually not that efficient (e.g., HHD with a threshold of 100).

Optimization Directions. The evaluation above reveals a few optimization directions. (1) Define a traffic equivalent class (EC) and solve constraints for one EC, which avoids repeatedly solving constraints for each packet instance in the same EC. There are three dimensions to define an EC: flow space, data path space, and state space. All packets in one EC should be in the same flow (could be a group according to the packet filter rules), go through the same data path, and operate on the same state variables. For example, in HHD and SSD with a threshold of 100 packets, all execution paths other than case for exceeding threshold can be verified using 3-4 packets. For solving the execution path in which threshold exceeds, we can use the constraint solving result of one packet for 100 packets EC, and the corresponding action takes place for 100th packet.

(2) For timing constraints, we could group a few packets and use one timestamp intervals for the group and execute timer between groups, which reduces the number of timing constraints on each execution path. This sampling-like estimation is a tradeoff between the result precision and the execution efficiency.

6 APPLICATIONS

In addition to find out and reduce false negatives and false positives in existing NF symbolic execution, NF-SE can be used in more scenarios:

Verifying NFs for flows with statistical properties. With NF-SE, the answer to a question like *"Can a flow A get through an NF B with configuration C ?"* can be more accurate. NF-SE first gets all execution paths for multiple packets, each with a packet trace (with timestamp), and checks the flow statistical properties (e.g., packet inter-arrival time [7]) with the packet trace, so that the answer could be *"When flow A gets through an NF B, X% packets would be dropped depending on timestamps of each packet in flow"*.

Verifying off-path network services. Some network services are not on-path to process each data plane packets, instead, they serve as the decision making entity in the control plane. Examples are network time protocol, DHCP, and DNS. Using NF-SE to validate their interactions with other data plane NFs would be an important application.

Verifying network-wide invariants. With NF-SE, verification of multiple NFs along with network topology information will help to reason about the network end-to-end

Table 1: Packet trace on some execution paths in the stateful firewall

	Exec. Path 1			Exec. Path 2			Exec. Path 3			Exec. Path 4			...
packet index	1	2	...	1	2	...	1	2	...	1	2	...	-
packet	SYN	SYN	-	SYN	SYN	-	SYN	!SYN	-	SYN	!SYN	-	-
timestamp(s)	1.123456	2.123456	-	1.123456	30.123456	-	1.123456	2.123456	-	1.123456	30.123456	-	-
state	CLOSE	OPEN	-	CLOSE	CLOSE	-	CLOSE	OPEN	-	CLOSE	CLOSE	-	-
state transition	OPEN	OPEN	-	OPEN	OPEN	-	OPEN	OPEN	-	OPEN	CLOSE	-	-
action	pass	pass	-	pass	pass	-	pass	pass	-	pass	drop	-	-

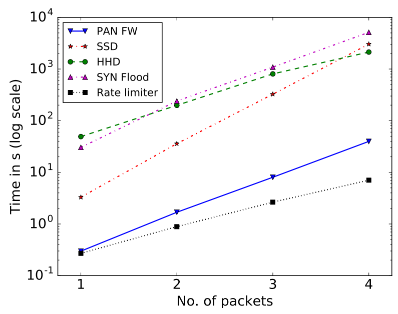

Figure 5: Total execution time, varying no. of packets

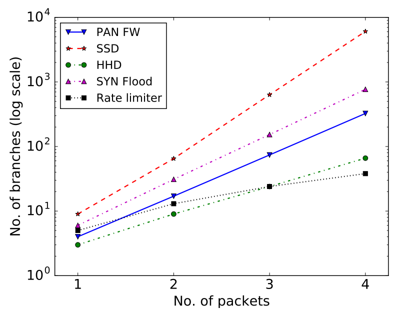

Figure 6: No. of execution paths, varying no. of packets

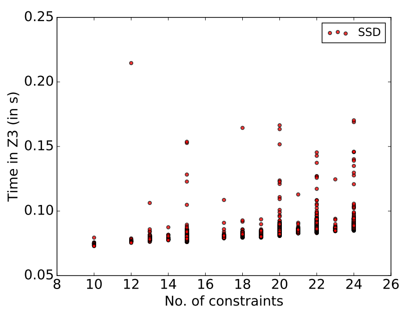

Figure 7: Exec. time v.s. no. of constraints on each exec. path of SSD

behavior. In such network-wide verification, adding time constraints between NFs can mimic their different processing speed, so that more runtime possibilities can be explored.

7 RELATED WORK

Individual NF Verification. Software network functions have large code base, and applying verification techniques such as symbolic execution on large NF implementations results into state explosion because its complexity increases exponentially with number of match action entries, and would introduce non-trivial manual code modification[11]. Other works, such as Vera [21] and P4V [18] targets P4 programs and finds out bugs such as parsing/deparsing errors, invalid memory accesses, loops and tunneling errors. And NF-SE complement them with time-driven logic.

Network-wide Verification/Testing. Large networks verification is a combination of topology discovery and individual NF behavior exploration for verifying entire networks of NFs. Existing solutions [4, 12, 18, 20, 22, 24] employ model checking techniques which involve creating behaviour models of NFs assuming some oracles or context dependent policies, and applying symbolic execution on those models to systematically explore of all possible execution paths of the system. A lot of effort has been made to solve the scalability issue, including using abstract data unit and operations, and slicing the network. NF-SE can enhance network-wide verification by providing more precise and correct behavior models and adding network-wide timing constraints.

Emulating NFs' processing in discrete time steps is another approach to explore NFs' behaviors in time domain (and the verification is described by linear temporal logic)[20]. NF-SE could accelerate this process by using timing constraints to represent multiple discrete time steps.

Another set of works focus on control plane verification[3, 14], and Kinetic[17] verifies network configuration changes, where NF-SE could provide more precise data plane behaviors. SLA-Verifier [25] focuses on verifying performance metrics, which is another domain. Alembic [19] automatically infers behavioral models of stateful NFs viewed as ensemble of finite-state machines. It injects input packets at constant interval for each NF and do not verify the temporal effects and cases where output packets are result of prior input packets.

8 CONCLUSION

We built NF-SE, a symbolic execution solution for NFs with time-driven logic. NF-SE include a DSL with time-driven logic primitives to model NF behaviors and a SEE to execute the model and output all execution paths. The future applications of NF-SE include NFs timing behavior verification, network service verification, and flow statistical properties verification.

REFERENCES

[1] https://www.antlr.org/.

[2] https://www.dropbox.com/s/6mxeobfss8w9iph/appendix.pdf?dl=0.

[3] Abhashkumar, A., Gember-Jacobson, A., and Akella, A. Tiramisu: Fast and general network verification. *arXiv preprint arXiv:1906.02043* (2019).

[4] Alpernas, K., Manevich, R., Panda, A., Sagiv, M., Shenker, S., Shoham, S., and Velner, Y. Abstract interpretation of stateful networks. In *International Static Analysis Symposium* (2018), Springer, pp. 86–106.

[5] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. Netkat: Semantic foundations for networks. *Acm sigplan notices 49*, 1 (2014), 113–126.

[6] Arashloo, M. T., Koral, Y., Greenberg, M., Rexford, J., and Walker, D. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference* (2016), ACM, pp. 29–43.

[7] Benson, T., Akella, A., and Maltz, D. A. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (New York, NY, USA, 2010), IMC '10, ACM, pp. 267–280.

[8] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 209–224.

[9] De Moura, L., and Bjørner, N. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 337–340.

[10] Dobrescu, M., and Argyraki, K. Software dataplane verification. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2014), NSDI'14, USENIX Association, pp. 101–114.

[11] Dobrescu, M., and Argyraki, K. Software dataplane verification. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)* (2014), pp. 101–114.

[12] Fayaz, S. K., Yu, T., Tobioka, Y., Chaki, S., and Sekar, V. Buzz: Testing context-dependent policies in stateful networks. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2016), NSDI'16, USENIX Association, pp. 275–289.

[13] Firestone, D. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 315–328.

[14] Gember-Jacobson, A., Raiciu, C., and Vanbever, L. Integrating verification and repair into the control plane. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), ACM, pp. 129–135.

[15] He, Y., and Wu, W. Fully functional rate limiter design on programmable hardware switches. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos* (New York, NY, USA, 2019), SIGCOMM Posters and Demos '19, ACM, pp. 159–160.

[16] Khurshid, S., Păsăreanu, C. S., and Visser, W. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2003), TACAS'03, Springer-Verlag, pp. 553–568.

[17] Kim, H., Reich, J., Gupta, A., Shahbaz, M., Feamster, N., and Clark, R. Kinetic: Verifiable dynamic network control. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015), pp. 59–72.

[18] 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 490–503.

[19] Moon, S.-J., Helt, J., Yuan, Y., Bieri, Y., Banerjee, S., Sekar, V., Wu, W., Yannakakis, M., and Zhang, Y. Alembic: automated model inference for stateful network functions. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 699–718.

[20] Panda, A., Lahav, O., Argyraki, K., Sagiv, M., and Shenker, S. Verifying reachability in networks with mutable datapaths. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2017), NSDI'17, USENIX Association, pp. 699–718.

[21] Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L., and Raiciu, C. Debugging p4 programs with vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, ACM, pp. 518–532.

[22] Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, ACM, pp. 314–327.

[23] Varghese, G., and Lauck, T. Hashed and hierarchical timing wheels: Efficient data structures for implementing a timer facility, 1996.

[24] Wang, A., Jia, L., Liu, C., Loo, B. T., Sokolsky, O., and Basu, P. Formally verifiable networking.

[25] Zhang, Y., Wu, W., Banerjee, S., Kang, J.-M., and Sanchez, M. A. Sla-verifier: Stateful and quantitative verification for service chaining. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications* (2017), IEEE, pp. 1–9.