Symbolic Execution for Network Functions with Time-Driven Logic

Harsha Sharma
harshasha256@gmail.com
Indian Institute of Technology Roorkee

Wenfei Wu
wenfeiwu@tsinghua.edu.cn
Tsinghua University

Bangwen Deng
dbw18@mails.tsinghua.edu.cn
Tsinghua University

Abstract

Symbolic Execution is a commonly used technique in network function (NF) verification, and it helps network operators to find implementation or configuration bugs before the deployment. By studying most existing symbolic execution engine, we realize that they only focus on packet arrival based event logic; we propose that NF modeling language should include time-driven logic to describe the actual NF implementations more accurately and performing complete verification. Thus, we define primitives to express time-driven logic in NF modeling language and develop a symbolic execution engine NF-SE that can verify such logic for NFs for multiple packets. Our prototype of NF-SE and evaluation on multiple example NFs demonstrate its usefulness and correctness.

I. Introduction

Network Functions (NF) are a family of software widely deployed in networks for the purpose of security (firewall, NAT, IPS/IDS), performance (proxy), improving bandwidth consumption (WAN optimizers) and management (load balancer, rate limiter, packet/byte counter). Network function virtualization (NFV) allows deployment of network functions without any change in the physical infrastructure, and eases the network management (handling software rather than hardware). Software NFs are widely deployed which makes it important for the network operators to verify their correctness before deploying them to production networks (avoiding runtime outage) and network researchers have proposed several verification solutions for NFs [1], [2], [3].

Symbolic execution for network functions (NFs) is a promising verification technique that 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 [4], configuration validation, and network testing [5], [2], [6]. Existing NF symbolic execution solutions use a domain specific language (DSL) to describe NF behaviors (i.e., model) and inject symbolic packets to execute the NF model and generate concrete test packets from these symbolic packets with constraints on their header fields. [4], [2], [6].

Harsha Sharma was an intern at Tsinghua University when this project started. Wenfei Wu is the corresponding author. This project is supported by National Natural Science Foundation of China Grant No. 61802225 and Turing AI Institute of Nanjing.

The expressiveness of a DSL decides whether an NF model can be used to represent its actual implementation. Most of the existing symbolic execution solutions assume NF logic is triggered by the packet arrival events, however, we realize that a variety of stateful NFs contain another kind of logic — *time-driven logic*, where NF states depends on the elapsed time. Without considering this 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 existing SEE predicts the flow to pass through.

In this work, our goal is to *extend the existing NF DSL with time-driven logic, enhancing the symbolic execution engine to execute this kind of logic and generating concrete test traffic from symbolic execution.* 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 use these abstract symbolic packets to construct concrete test traffic for the purpose of active testing.

We name our solution NF-SE, and prototype it. Our evaluation shows that NF-SE can verify NFs with time-driven logic, correcting the false positive/negative results of NF verification without time-driven logic. The performance of NF-SE (execution time) is acceptable for several typical NFs (specifically, state-expiration NFs (Section V)); for other NFs (counter-attenuation NFs, (Section V)) NF-SE is not very efficient (still usable) and we suggest ways to optimize the execution time (Section VII-B)).

II. Background and Motivation

Symbolic Execution on NFs is used for verifying the correctness of NFs , and existing symbolic execution solutions need to consider time-driven logic for a more precise result.

A. NF Symbolic Execution

In individual NF verification, exploring all the execution paths helps in finding out implementation or configuration errors [7], [6]; in network-wide NF verification [5], [4], 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 packet, forks for each conditional branch, keeps a track of constraints on each branch, and outputs abstract input packet for each valid path satisfying all constraints on that path. Concrete test traffic is then generated from these abstract input packets and the behaviour of data plane is tested by injecting this concrete test traffic. This is called active testing [5]. Some of the paths in network function code, depends on previous packets arrived i.e. the histories over multiple packets (e.g prior established connections). To verify such paths, we need to inject multiple symbolic packets, such that the packets are injected sequentially after the execution of previous packets is completed.

Applying general symbolic execution engine (e.g.,KLEE [8]) directly on NF code would cause (unnecessary) path explosion, causing intolerable execution times and network irrelevant path constraints. For example, `sprintf(ip_str, "%d.%d.%d.%d", ip >> 24 % 256, ip >> 16 % 256, ip >> 8 % 256, ip % 256)` would cause 3^4 branches, because each 8-bit segment can be 1 or 2 or 3 digits when printed to string in decimal. But in NF specific verification, these branches hardly reflect meaningful network semantics. Thus, symbolic execution is usually performed on NF models which are specified by a domain specific language, called *NF modeling language*. SymNet [2], BUZZ [5], VMN [4], Kinetic [9] and Vera [6] are examples of such symbolic execution engines with domain specific languages.

B. 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 symbolic execution 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 NFs 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 (Section III). In practice, many NFs contain time-driven logic; for example, a 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. Fig. 1 shows the basic logic of the

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

Fig. 1: Pseudocode of a stateful firewall

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

Fig. 2: Pseudocode of a stateful firewall with state expiration

firewall — the SYN packet of a TCP flow punches a hole in the firewall, and all subsequent packets from internal host A to external host B 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". This state can also be used to allow traffic from the external host B to internal host A.

While in previous NF active testing solutions, a stateful firewall is represented in this way (Fig. 1)[10], the practical implementation usually contains another kind of logic — the state would expire after a certain amount of time if no packet arrives during that time. If no packet has arrived for some threshold time, the state would return to "CLOSE", meaning further communication requires a preceding SYN packet to again reopen the connection. Fig. 2 shows such an implementation: a timer would be triggered to check the state refreshment, if no packet arrives for a threshold time period, the state is reset to "CLOSE".

Without modeling such time-driven logic in NF models, the symbolic execution results could possibly mismatch the actual

$\boxed{\text{Basic types and expression}}$			
const	c	$::=$	$(0 \mid 1)^{+}$
header field	h	$::=$	$sip \mid dip \mid sport \mid dport \mid proto \mid ...$
state	s		
variable	var		
expression	e	$::=$	$c \mid h \mid s \mid var \mid e \mid Expr\_Op(e_1, e_2, ...)$
$\boxed{\text{Predicates}}$			
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	$::=$	$Rel\_Op(e_1, e_2, ...) \mid \neg x \mid x \wedge y \mid x \vee y$
$\boxed{\text{Policies and Statements}}$			
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	$::=$	$q := e \mid p; q$
$\boxed{\text{Model}}$			
model	$model$	$::=$	$stmts$
statements	$stmts$	$::=$	$stmt \mid stmt; stmts$
statement	$stmt$	$::=$	$p \mid if$
if statement	if	$::=$	if $(x)\{(stmts\}$ else $\{stmts\}$

Fig. 3: NF-SE language syntax (following SNAP and NetKAT[12][11])

NF behaviors and this may cause false negatives (i.e., reporting unsafe behaviors as safe): in the stateful firewall example (Fig. 2), the actual firewall may stop a flow due to its long-term inactiveness, but due to lack of time-driven logic modeling, 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 easily verifiable *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.

III. Modeling Time-Driven Logic

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

A. NF Modeling Language

We summarize NF modeling language from several existing solutions [2], [11], [4] , and its syntax is shown in the 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 left-hand symbol to be that of the right-hand expression, and a branching statement means if the condition is true, execute the if branch, otherwise execute the else branch. The whole program 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 (Section II)).
- *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 [4], [2] , 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]).

B. 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 [14]), 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 Fig. 1, the "token" in Fig. 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. [1]
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 Fig. 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 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 Fig. 2 follows the syntax, and we also show another example of a rate limiter (leaky bucket algorithm) (Fig. 4). It maintains a token variable to record the budget to send packets. Sending packets would consume the token until token is zero and if the token is less than the size of the packet, the packet is dropped; the token is refreshed periodically by a timer.

IV. 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). We assume the timestamps of incoming packets to be monotonically increasing and add constraints such as timestamp of the packet 2 is greater than timestamp of packet 1 and construct abstract packets with such constraints.

[1]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;
  }
}
```

Fig. 4: Pseduocode of a rate limiter

A. 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 of 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 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. So, our trace tree only contains valid execution paths for different input packets.

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 outputting all possible execution cases of multiple packets.

B. 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 associated with a variable (e.g., the timestamp when a "packet" is received, the timestamp when a "state" is updated). A timestamp's initial value is set 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. 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 conditional constraints (e.g., last modified time is within 30s in Fig. 2), and are treated the same as other variables and constraints.

(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 assuming timestamp for incoming packets are increasing in nature.

(4) When `timer(TIME, HANDLER, ARGS)` is "called" to invoke an event in the future time, NF-SE declares a timestamp to record the current time and a constraint which records an association of the timer and handler with this timestamp (for the execution in the future).

(5) When NF-SE gets to a `timer(TIME, HANDLER, ARGS)` execution, it first adds a timestamp (e.g., t1) of current time (as well as its constraints), extracts the timestamp when this timer is "called" (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 directly.

(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 (i.e multiple execution is equivalent to one execution; examples are state expiration); (2) variables that periodically and monotonically change usually have a threshold (e.g., rate limiter's token is accumulated to the burst size and does not change any more).

V. Implementation

We use Antlr[17] 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[18], which has 1600+ lines of code in C++.

We implement 14 NFs from existing literature[11], [19], [12], and add time-driven logic to 5 of them; we show the models of NFs with and without time-driven logic in appendix[20]. Time-driven logic usually exists in two formats in these NFs: state expiration and *counter attenuation*. Stateful firewalls usually has state expiration, such as the firewall in Fig. 2 and TCP 3-way handshake checking firewall and TCP retransmission timeout. Counter attenuation usually exists in rate limiters and intrusion detection systems (IDS), such as super spreader detector (SSD), heavy hitter detector (HHD)[11], [21], and SYN flood detector.

VI. Evaluation

Our experiments are conducted on a quad-core, Intel i3 laptop with 8 GB RAM. We show that NF-SE can correctly verify NFs' execution paths, and we measure the execution time taken by our SEE.

A. Functional Validation

In experiment results, we observe that NF-SE overcomes the false positive/negative issues mentioned in section II. It generally outputs more data paths (with more constraints) for time-driven NFs as compared to NFs without time-driven logic.

Case Study: IDS. In an intrusion detection system, if the packet rate (Packet Per Second, PPS) is not high, i.e., the time gap between two consecutive packets is high, then the network should not be assumed under attack/abuse. NF-SE could explore such execution paths where if the two consecutive packets timestamp difference exceeds a threshold limit, then counter is multiplied by a ratio less than 1 to decrease the counter, preventing a false positive of marking this scenario as an attack.

Case Study: Stateful Firewall. We show an example of verifying the NF in Fig. 2. In table I , 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 and the second packet gets through and path 4 is the case when the state expires and the second packet is dropped. Thus, NF-SE can overcome the false negative case in section II .

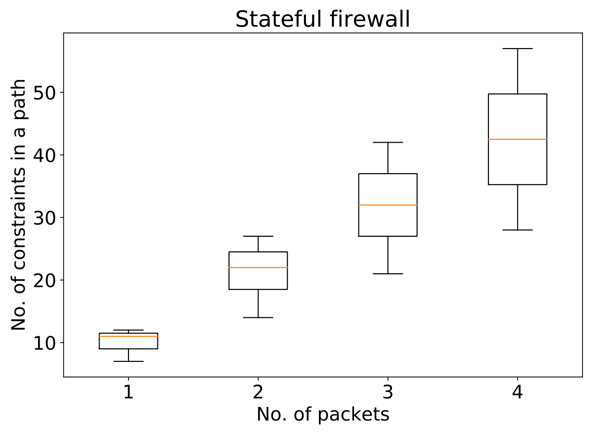

Fig. 5: No. of constraints vs no. of input packets for stateful firewall example

We compare the number of execution paths between stateful firewall with and without time-driven logic in Fig. 5 and 6.

TABLE I: Packet trace on some execution paths in the stateful firewall

	Execution Path 1			Execution Path 2			Execution Path 3			Execution 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	-	-

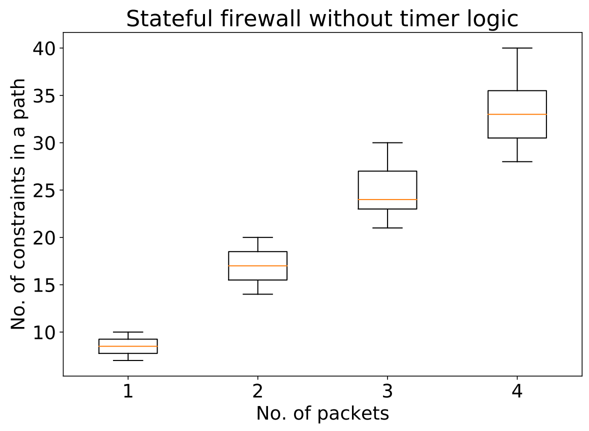

Fig. 6: No. of constraints vs no. of input packets for stateful firewall example not containing any time-driven logic

In both the figures, the x-axis denotes the number of packets processed by the NF (which is given as an input to SEE), and the y-axis shows the number of constraints in different paths for processing those packets; each path has several constraints, and the vertical bar shows that min-25%-50%-75%-max of the number of constraints of all paths when the NF processes a certain packet. We note the following observations: first, the number of constraints increases almost linearly with the number of packets, as we recursively travel the trace tree for multiple number of packets; second, NFs with time-driven logic have more constraints because of the additional time-driven logic, e.g., median 42 v.s. 32 when processing the 4-th packet; third, the additional constraints for time-driven logic in stateful firewall are not very significant, but this depends on the complexity of the time-driven logic in the NF.

B. Overhead

We make extensive evaluation on the overhead of verifying time-driven logic. Fig. 7 shows the execution time to verify NFs with time-driven logic, Fig. 8 shows execution time for NFs without time-driven logic, and Fig. 9 shows the number of branches (execution paths) for NFs with time-driven logic. We have the following observations.

First, we see that NFs with state expiration (stateful firewall) takes less verification time as compared to NFs with counter attenuation. For example, to symbolically execute 2 packets, firewall takes about 10s, and HHD takes about 100s. Because "state" variables are usually of enumeration type and have less branching choices, but "counter" variables are usually integers, whose value space can be very large. For example, an HHD with a threshold of 100 needs 100 symbolic packets to reach the state transition.

Second, the total execution time can be estimated by the product of the number of branches and the constraint solving time of each branch. More than 96% of the time is consumed in solving constraints using Z3; 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 with number of input packets.

Third, adding time-driven logic to NFs causes extra overhead; the overhead is acceptable for state-expiration NFs but is quite significant for counter-attenuation NFs. The difference of the overhead for these two types of NFs is caused by the size of the value space for states (usually enumerations) and counters (usually integers).

VII. Discussion

A. Application Scenarios

We discuss how to use NF-SE to explore all execution paths of NFs with time-driven logic, and use the execution paths to correct the false positives/negatives in previous SEE solutions. The individual NF verification can be extended to more NFs, e.g., off-path services such as DNS and DHCP.

Active Testing Trace Generation. There is a trend to apply model-centric programming in both the networking and software engineering community [9], [4], [2], [22], [23], [24], [11], [25]. In model-centric programming, developers use modeling language to describe the software (NF in our case) behaviors and use compilers to translate the model to runnable code. We similarly use NF models to generate execution paths and symbolic packets to exercise those path, then these symbolic abstract packets are translated to actual packet traces; with the compiled code and the packet trace, active testing can be conducted by injecting the concrete packet traces to deployed NFs for verifying the correctness of both the model and the compiler implementation.

Verifying NFs for flows with statistical properties. Traditional NF verification tools answer queries like *"Can a flow A get through an NF B with configuration C ?"* While NF-SE adds time-driven logic and can give detailed answers like *"When flow A gets through an NF B, X% packets would be dropped depending on timestamps of each packet in flow"*.

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

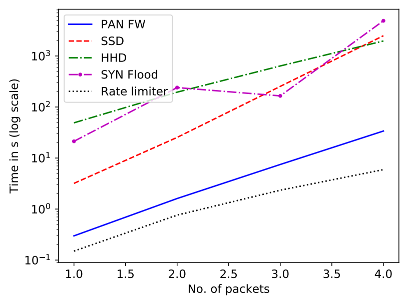

Fig. 7: Total execution time, varying the no. of packets

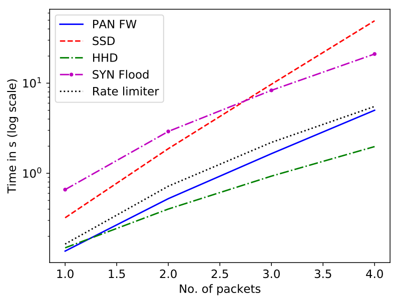

Fig. 8: Total execution time for NFs without time-driven logic

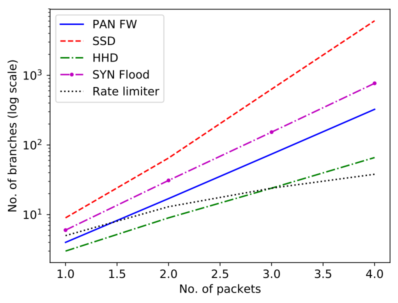

Fig. 9: No. of execution paths, varying no. of packets

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

B. 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 the case for exceeding the 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.

VIII. 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. Works such as Vera [6] and P4V [26] target P4 programs and find bugs such as parsing/deparsing errors, invalid memory accesses, loops and tunneling errors. Whereas NF-SE complements them with time-driven logic.

Network-wide Verification/Testing. Network verification is a combination of topology discovery and individual NF behavior exploration for verifying chains of NFs in the network. Existing solutions [5], [2], [4], [27], [28], [26] employ model checking techniques which involve creating behaviour models of NFs assuming some oracles or context dependent policies, and applying symbolic execution on these models to systematically explore all possible execution paths of the system. 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)[4]. 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[29], [30], and Kinetic[9] verifies network configuration changes, whereas NF-SE provides more precise data plane behaviors. SLA-Verifier [31] focuses on verifying performance metrics, which is another domain. Alembic [32] automatically infers behavioral models of stateful NFs viewed as an ensemble of finite-state machines. It injects input packets at constant interval for each NF and does not verify the temporal effects and cases where output packets depend on histories of previous input packets.

Time Related Logic. Varanus [33] is a network monitoring solution with "timeout" semantics, whereas NF-SE is a generic NF modeling and SEE. Kinetic [9] verifies the controller programs against user-specified computation tree logic (CTLs) whereas NF-SE facilitates active testing by generating concrete test traffic.

IX. Conclusion

We built NF-SE, a symbolic execution solution for NFs with time-driven logic. NF-SE includes a DSL with time-driven logic primitives to model NF behaviors and a SEE to explore all the execution paths of the model. Our prototype of NF-SE and evaluation on 5 NFs shows that NF-SE can be used to verify the implementation and configuration of individual NFs; and we show how NF-SE complements and corrects false positives/negatives in existing NF symbolic execution solutions. We also show the potential application scenarios and optimization directions for NF-SE.

References

[1] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. USA: USENIX Association, 2012, p. 9.

[2] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: Scalable symbolic execution for modern networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 314–327. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934881

[3] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 290–301. [Online]. Available: https://doi.org/10.1145/2018436.2018470

[4] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. Berkeley, CA, USA: USENIX Association, 2017, pp. 699–718. [Online]. Available: http://dl.acm.org/citation.cfm?id=3154630.3154687

[5] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "Buzz: Testing context-dependent policies in stateful networks," in *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 275–289. [Online]. Available: http://dl.acm.org/citation.cfm?id=2930611.2930630

[6] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging p4 programs with vera," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018, pp. 518–532. [Online]. Available: http://doi.acm.org/10.1145/3230543.3230548

[7] M. Dobrescu and K. Argyraki, "Software dataplane verification," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 101–114. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616459

[8] C. Cadar, D. Dunbar, and D. Engler, "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*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

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

[10] D. Joseph and I. Stoica, "Modeling middleboxes," *Netwrk. Mag. of Global Internetwkg.*, vol. 22, no. 5, p. 20–25, Sep. 2008. [Online]. Available: https://doi.org/10.1109/MNET.2008.4626228

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

[12] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," *Acm sigplan notices*, vol. 49, no. 1, pp. 113–126, 2014.

[13] D. Firestone, "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: USENIX Association, Mar. 2017, pp. 315–328. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone

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

[15] Y. He and W. Wu, "Fully functional rate limiter design on programmable hardware switches," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos '19. New York, NY, USA: ACM, 2019, pp. 159–160. [Online]. Available: http://doi.acm.org/10.1145/3342280.3342344

[16] S. Khurshid, C. S. Păsăreanu, and W. Visser, "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*, ser. TACAS'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 553–568. [Online]. Available: http://dl.acm.org/citation.cfm?id=1765871.1765924

[17] "https://www.antlr.org/."

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

[19] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[20] "Appendix with nf models code." [Online]. Available: https://www.dropbox.com/s/7i9s4um35d4ecyp/

[21] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[22] H. Huang and W. Wu, "Nfd: Using behavior models to develop cross-platform nfs," 08 2019, pp. 153–155.

[23] K. Gao, T. Nojima, and Y. R. Yang, "Trident: toward a unified sdn programming framework with automatic updates," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 386–401.

[24] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mos: A reusable networking stack for flow monitoring middleboxes," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 113–129.

[25] G. Liu, Y. Ren, M. Yurchenko, K. Ramakrishnan, and T. Wood, "Microboxes: high performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 504–517.

[26] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "P4v: Practical verification for programmable data planes," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: ACM, 2018, pp. 490–503. [Online]. Available: http://doi.acm.org/10.1145/3230543.3230582

[27] A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu, "Formally verifiable networking," 2009.

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

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

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

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

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

[33] T. Nelson, N. DeMarinis, T. A. Hoff, R. Fonseca, and S. Krishnamurthi, "Switches are monitors too! stateful property monitoring as a switch design criterion," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, ser. HotNets '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 99–105. [Online]. Available: https://doi.org/10.1145/3005745.3005755