APPENDIX

A FORMAL SEMANTICS OF SNAP LANGUAGE EXTENSION

Operators

$$\text{eval}_e(Expr\_Op(e_1, e_2, \ldots), pkt) = Expr\_Op(\text{eval}_e(e_1, pkt), \text{eval}_e(e_2, pkt), \ldots)$$
$$\text{eval}_e(Rel\_Op(e_1, e_2, \ldots), pkt) = Rel\_Op(\text{eval}_e(e_1, pkt), \text{eval}_e(e2, pkt), \ldots)$$
$$\text{eval}_e(New\_Op(e_1, e_2, \ldots), pkt) = New\_Op(\text{eval}_e(e_1, pkt), \text{eval}_e(e_2, pkt), \ldots)$$

entry and model

$$\begin{aligned}
\text{eval}(ent, m, pkt) &= \text{eval}(\text{if } (x_f \wedge x_s) \text{ then } (p_f; p_s) \text{ else } \bot, m, pkt) \\
&= \text{let } (m', PKT, l) = \text{eval}(x_f \wedge x_s, m, pkt) \text{ in} \\
&\quad \begin{cases} \text{eval}(p_f; p_s, m, pkt), & \text{if } PKT = pkt \\ (m, pkt, \bot), & \text{if } PKT = \emptyset \end{cases} \\
\text{eval}(ent_1 + ent_2, m, pkt) &= \text{let } (m', PKT, l) = \text{eval}(ent_1, m, pkt) \\
&= \begin{cases} (m', PKT, l), & \text{if } l \neq \bot \\ \text{eval}(ent_2, m, pkt), & \text{if } l = \bot \end{cases} \\
\text{eval}(model, m, pkt) &= \text{eval}(ent + model, m, pkt)
\end{aligned}$$

Figure 1: Semantics of SNAP extension

We use the notations in SNAP[1] (the Appendix A in paper[1]). We do not elaborate the overlapping part, but go on adding the semantics of operators, parallel composition, entry, and model.

B SEMANTIC COMPLETENESS OF NFCC MODEL

B.1 Proof

In the following proof, we only consider programs composed of assignment statements, return statements, if-statements, and loop-statements. We argue that other program language features (e.g., goto-statement, case-statement) can be transformed to the four kinds of statements above. We prove that *a program composed of the four structures has an logical equivalent NFcc model*. The proof is as follows.

Theorem 1. An if-else-statement can be filled into a stateful match action table with equivalent logic.

Proof. The transformation from the program to the table is as follows. In the table, the match and action fields can be further split according to whether the statements are about flow or states.

if(condition)
{body1}
else {body2}

$\Longrightarrow$

Match	Action
condition	body1
¬condition	body2

□

Theorem 2. A loop-statement can be filled into a stateful match action table with equivalent logic.

Proof. The transformation is as follows. f[BR] is a tag attached with a packet, it has an initial value which matches the branches of initial execution. And the tag can be re-assgined by actions; when the packet is resubmitted, the new tag value would direct the packet to a specific entry in the table.

while(cnd)
{
body
}

$\Longrightarrow$

Match	Action
cnd ∧ f[BR]=INIT	body, f[BR]=X, f[output]=resubmit
cnd ∧ f[BR]=X	body, f[BR]=X f[output]=resubmit

□

Theorem 3. A sequence of statements (both simple and compound) can be split into multiple entries in a stateful match action table with equivalent logic.

Proof. The transformation is as follows.

statement1	⟹	**Match**	**Action**
statement2		f[BR]=INIT	statement1, f[BR]=X
statement3		f[BR]=X	statement2, statement3

□

Theorem 4. A program can be recursively applied the three transformations above to be filled into a stateful match-action table with equivalent logic.

Proof. (1) If there exist compound statements, split them into a separate branch using theorem 3. (2) Rewrite each compound statement branch into match-action entries using theorem 1 and 2. (3) If there are interleaved state/flow statements, split them into multiple entries where each entry executes flow action first and state action afterward. □

B.2 Transform Code to Model

Each entry in SMAT is an ordered sequence of flow predicates, state predicates, flow policies, and state policies. When translating a program, if a basic block is met, put the predicates to be wildcards (∗), and put flow policies and state policies in the same entry. If state policies and flow policies are disordered or interleaved (e.g., $state_1$, $flow_1$, $state_2$), break the sequence into two subsequences, resubmit at the border of disorder (i.e., between $state_1$ and $flow_1$), and rewrite each subsequence recursively (getting $< f[output] = resubmit, state_1 >$ and $< flow_1, state_2 >$). If an if-statement is met, put the conditions of the if-statement in the flow and state predicates, and disassemble the body of the if-statement as a basic block to fill in flow/state policies. If a while-loop is met, put the conditions in flow/state predicates, disassemble the loop body into flow/state policies, and resubmit to the beginning entry of the loop.

B.3 Examples of Transformation

TTL−−;		**Match**	**Action**
if(TTL==0) {drop;}	⟹	f[BR]=INIT	f[TTL]−−, f[BR]=X, f[output]=resubmit
else {pass;}		f[BR]=X ∧ f[TTL]==0	f[output]=ϵ
		f[BR]=X ∧ f[TTL]≠ 0	f[output]=IFACE

(a) TTL decrement (from program to model)

// ETH:0x800, TCP:		**Match**	**Action**
// 0x06, UDP: 0x11		f[BR]=INIT	EthCnt++, f[BR]=X1 f[output]=resubmit
EthCnt++			
if(type==ETH){		f[BR]=X1 ∧ f[type]=ETH	IPCnt++, f[BR]=X2 f[output]=resubmit
IPCnt++;	⟹		
if(proto==TCP) {TCPCnt++;}		f[BR]=X2 ∧ f[proto]=TCP	TCPCnt++
else if(proto==UDP) {UDPCnt++;}}		f[BR]=X2 ∧ f[proto]=UDP	UDPCnt++

(b) Packet statistics (from program to model)

Match		**Action**	
flow	**state**	**flow**	**state**
f[TTL]≤1	*	f[output]:=ϵ	-
f[TTL]>1	*	f[TTL]:=f[TTL]-1,f[output]:=IFACE	-

(c) TTL decrement (rewrite program)

Figure 2: Examples of translating programs to models

Example: interleaved flow/state match/action. When a packet passes network devices, a common operation at each hop is to decrement TTL, and drop the packet if TTL is 0. The flow match/action are interleaved: flow action first, then flow match, and finally flow action. This TTL logic can be transformed into a match-action table without changing its logic as in Figure 2a.

Example: nested if-statements. When an NF processes packets, header parsing is usually done layer by layer. In the example of a packet statistics NF, if an ethernet packet has an IP (0x0800) packet as its payload, it would continue to be processed; if the IP packet's protocol field is TCP(0x06), the packet would be delivered to TCP processing logic; if UDP(0x11), it would be directed to UDP processing logic. When an if-statement is nested with another one, the inner if-statement can be written as a new branch resubmitted right after its preceding statement (Figure 2b).[1]

[1]Another approach is to find each execution path and collect all conditions on the path, and then rewrite each execution path as a match-action entry. This approach needs to figure out all execution paths using techniques such as symbolic execution, and our approach above can avoid this exhaustion.

Practical NF development. Although this section shows that a program with both simple and compound statements can always be transformed to a stateful match-action table. However, in practical development, we still suggest reconstructing the NF logic to simplify the table structure. For example, the TTL decrement example above can be rewritten as in Figure 2c: the TTL is first compared with 1; if the TTL value is equal to or less than 1, drop the packet; otherwise, decrement TTL and send the packet.

C EVALUATION RESULTS

C.1 Test on Correctness of NFcc Firewall

Table 1: Number of reported alerts

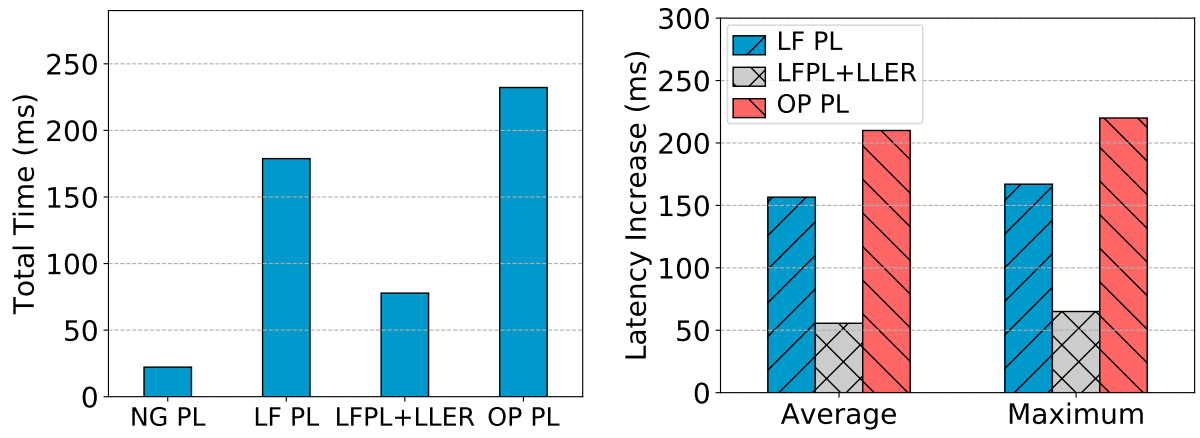
Blocked Subnet	Firewall	Snort
41.177.117.0/24	15999	15999
244.3.160.0/24	107259	107259
244.253.127.0/24	229	229
41.0.0.8/8	271791	271791
41.177.117.184/32	14347	14347

C.2 Results of Synthesizing LB and IDS

Table 2: Tests of a load balancer with a blacklist

Blocked Subnet(%)	LB(%)	FW(%)	FW+LB(%)
244.3.0.0/17(5.0)	(18.7,25.1,22.4,21.4,12.4)	95.0	(18.6,23.9,21.5,19.1,11.9)
244.3.128.0/17(33.4)	(18.7,25.1,22.4,21.4,12.4)	66.6	(22.1,16.5,9.2,8.4,10.4)
244.3.0.0/16(38.5)	(18.7,25.1,22.4,21.4,12.4)	61.5	(18.7,14.6,8.6,9.1,10.5)

C.3 Result of OpenNF Experiment



(a) State move time **(b) Packet latency**

Figure 3: Repeat state move experiments with NFcc-based Firewall and OpenNF.

We draw the similar conclusion as in OpenNF[2]. (1) the stricter state migration requirement (no guarantee (NG) > loss-free (LF) > order-preserving (OP)) makes the state move time and packet latency longer; (2) the optimizations (parallelizing (PL) and late-locking-early-release (LLER)) in OpenNF improves the state move time and packet latency.

D SUPPORTED NFS AND THEIR PERFORMANCE

We wrote 14 NF models and perform unit test on them. The functionality description and the performance of these NFs are listed in Table 3.

Table 3: NF examples and throughput (with 64-Byte packets)

NF Model	Function	Throughput (kpps)
Stateful Firewall	Security	4317
Heavy Hitter Detection	Security	4735
Super Spreader Detection	Security	3883
SYN Flood Detection	Security	3910
DNS Amplification Mitigation	Security	3474
UDP Flood Mitigation	Security	4386
NAPT	Application-Based Network	3233
Rate Limiter	Network Management	1650
Packet Load Balancer	Network Management	4930
Bytestream Load Balancer	Network Management	28400(mbps)
Stateless Firewall	Security	4702
NAT	Application-Based Network	3517
Packet Counter	Network Management	1548
Byte Counter	Network Management	1514

Init: int token = TOKENS;			
Match		**Action**	
flow	**state**	**flow**	**state**
f[tag]==1	*	-	f[output]=timer(1, 5000 , 1); token =10000;
f[tag]==0	token<f[iplen]	f[output]=ϵ;	-
f[tag]==0	token>=f[iplen]	-	token = token- f[iplen];

Figure 4: The model of a rate limiter

Init: rule R=sip:192.168.0.0/16; IP base=219.168.135.100/32; int port=12000; map<IP,map<int, int» mapping; map<int, IP> port_ip; map<int, int> port_port;			
Match		**Action**	
flow	**state**	**flow**	**state**
f ⊏ R	(f[sip]∈mapping) ∧ (f[sip]∈mapping ∧ (f[sport]∈mapping[f[sip]]))	f[sip]=base; f[sport]=port; port=port+1;	mapping[f[sip]][f[sport]] = f[port]; port_ip[port]= f[sip]; port_port[port]=f[port];
f ⊏ R	f[sip]∈mapping ∧ (f[sport]∈mapping[f[sip]])	f[sip]=base; f[sport]=mapping[f[sip]][f[sport]];	-
f ⋤ R ∧ f[dip] == base	f[dport]∈port_port	f[dip]= port_ip[f[dport]]; f[dport]=port_port[f[dport]];	-
f ⋤ R ∧ f[dip] != base	*	f[output]=ϵ;	-

Figure 5: The model of an NAT

Init: map<IP,int> udpcounter; map<IP,int> udpflood; int threshold=100;			
Match		**Action**	
flow	**state**	**flow**	**state**
f[UDP]==1	udpflood[f[sip]]!=1 ∧ udpcounter[f[sip]]!=threshold	-	udpcounter[f[sip]]=udpcounter[f[sip]]+1;
f[UDP]==1	udpflood[f[sip]]!=1 ∧ udpcounter[f[sip]]==threshold	f[output]=ϵ;	udpflood[f[sip]]=1;
f[UDP]!=1	*	-	-

Figure 6: The model of UDP Flood Mitigation

Init: rule ALLOW = sip:192.168.22.0/24;set<IP> seen;			
Match		**Action**	
flow	**state**	**flow**	**state**
f ⊏ ALLOW	*	-	seen= seen | f[dip] ;
f ⊄ ALLOW	f[sip]∈seen	-	-
f ⊄ ALLOW	(f[sip]∈seen)	f[output]=ϵ;	-

Figure 7: The model of a Stateful Firewall

Init: int counter (sip&dip) = 0;			
Match		**Action**	
flow	**state**	**flow**	**state**
*	*	-	counter = counter + f[iplen];

Figure 8: The model of a Byte Counter

Init: map<IP, int> hh;map<IP,int> hh_counter;int threshold=100;			
Match		**Action**	
flow	**state**	**flow**	**state**
f[flag_syn]==1	hh[f[sip]]!=1 ∧ hh_counter[f[sip]]!=threshold	-	hh_counter[f[sip]]=hh_counter[f[sip]]+1;
f[flag_syn]==1	hh[f[sip]]!=1 ∧ hh_counter[f[sip]]==threshold	-	hh[f[sip]]=1;
f[flag_syn]==1	hh[f[sip]]==1	-	-
f[flag_syn]!=1	*	-	-

Figure 9: The model of a Heavy Hitter Detection

Init: rule ALLOW = sip:192.168.22.0/24;			
Match		**Action**	
flow	**state**	**flow**	**state**
f ⊏ ALLOW	*	-	-
f ⊄ ALLOW	*	f[output]=ϵ;	-

Figure 10: The model of a Stateless Firewall

Init: rule R=sip:192.168.0.0/16; IP base=219.168.135.100/32; int port=8; map<int,IP> listIP; map<int,int> listPORT;			
Match		**Action**	
flow	**state**	**flow**	**state**
f ⊏ R	*	listIP[port]=f[sip]; listPORT[port]=f[sport];	f[sip]=base; f[sport]=port; port=port+1;
f ⊄ R ∧ f[dip] == base	f[dport]∈listIP	f[dip]=listIP[f[dport]]; f[dport]=listPORT[f[dport]];	-
f ⊄ R	(f[dport]∈listIP)	-	-
f ⊄ R ∧ f[dip] != base	*	-	-

Figure 11: The model of an NAPT

Init: map<IP,map<IP,int» bq;			
Match		**Action**	
flow	**state**	**flow**	**state**
f[dport]==53	*	-	bq[f[sip]][f[dip]]=1;
f[dport]!=53 ∧ f[sport]==53	bq[f[sip]][f[dip]]!=1	f[output]=ϵ;	-
f[dport]!=53 ∧ f[sport]!=53	*	-	-
f[dport]!=53	bq[f[sip]][f[dip]]==1	-	-

Figure 12: The model of a DNS Amplification Mitigation

Init: int round=0; int max_round=2; map<int, IP> pool= 0: "192.168.1.1/32", 1: "192.168.1.2/32"; map<int, port> poolport = 0:11111, 1:11112; map<int, int> connections;			
Match		**Action**	
flow	**state**	**flow**	**state**

connection f	*	-	int s1 = accept();
			round = (round+1) % max_round;
			int s2 = connect(pool[round]);
			connections[s1]=s2;
			connections[s2]=s1;

!(connection f)	*	send_to(connections);	-

Figure 13: The model of an L4 Load Balancer

Init: map<IP,int> list; map<IP,int> tlist; int threshold=100;			
Match		**Action**	
flow	**state**	**flow**	**state**
f[flag_syn]==1	tlist[f[sip]]!=1 ∧ list[f[sip]]!= threshold	-	list[f[sip]]=list[f[sip]]+1;
f[flag_syn]==1	tlist[f[sip]]!=1 ∧ list[f[sip]]== threshold	-	tlist[f[sip]]=1;
f[flag_syn]==1	*	-	list[f[sip]]=list[f[sip]]-1;
f[flag_syn]!=1	*	-	-

Figure 14: The model of a Super Spreader Detection

Init: int counter (sip&dip) = 0;			
Match		**Action**	
flow	**state**	**flow**	**state**
*	*	-	counter= counter+1;

Figure 15: The model of a Packet Counter

Init: map<IP,int> blist; int threshold=100;			
Match		**Action**	
flow	**state**	**flow**	**state**
f[flag_syn] ==1 ∧ f[tag]!=1	*	-	blist[f[sip]]=blist[f[sip]]+1; f[tag]=1; f[output]=resubmit;
f[tag]==1	blist[f[sip]]== threshold	f[output]=ϵ;	-
f[tag]==1	blist[f[sip]]!= threshold	-	-
f[tag]!=1 ∧ f[flag_syn] !=1 ∧ f[flag_ack] ==1	*	-	blist[f[sip]]=blist[f[sip]]-1;
f[tag]!=1 ∧ f[flag_syn] !=1 ∧ f[flag_ack] !=1	*	-	-

Figure 16: The model of a SYN Flood Detection

Init: map <int, IP> list=0:10.0.1.10/32,1:10.0.1.20/32,2:10.0.1.30/32,3:10.0.1.40/32,4:10.0.1.50/32; int round = 0; int round_max = 5;			
Match		**Action**	
flow	**state**	**flow**	**state**
*	*	f[dip] = list[round];	round = (round +1)% round_max;

Figure 17: The model of an L3 Load Balancer

REFERENCES

[1] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 29–43.

[2] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*.