Automatic Synthesis of NF Models by Program Analysis

Wenfei Wu, Ying Zhang and Sujata Banerjee

Hewlett Packard Labs

ABSTRACT

Network functions (NFs), like firewall, NAT, IDS, have been widely deployed in today's modern networks. However, currently there is no standard specification or modeling language that can accurately describe the complexity and diversity of NFs. Recently there have been research efforts to propose NF models. However, they are often generated manually and thus error-prone. This paper proposes a method to automatically synthesize NF models via program analysis. We develop a tool called NFactor, which conducts code refactoring and program slicing on NF source code, in order to generate its forwarding model. We demonstrate its usefulness on two NFs and evaluate its correctness.

1. INTRODUCTION

Due to recent advances in software defined networking (SDN) and in particular, network programmability, there is an opportunity to make significant breakthroughs in network troubleshooting and verification. What used to take hours involving mostly manual debugging of network problems, can perhaps be accomplished in near real-time and in some cases, even proactively before problems occur. However, a snag in this vision is the wide spread use of closed network middleboxes in deployed networks. These middleboxes host important network functions (NF) operate at different layers (L2 frames up to application level packets) - examples include caches, proxies, load balancers, firewalls, intrusion prevention/detection systems, deep packet inspection systems, etc. In addition, NFs can be stateless or stateful, the latter being harder to analyze. When network packets traverse NFs, it gets harder to troubleshoot and verify network operations since there is no easy way to reason about the operation of complex closed NFs. While network programmability and SDN have improved global visibility, and made it possible to reason about network paths, there are few effective ways to reason about paths with arbitrary NFs.

In this paper, we address the issue of modeling NFs with a view to enabling network verification, service chaining policy composition and testing. While modeling all aspects of NF operations is challenging, we focus on modeling just the forwarding behavior of network NFs, to drive the above applications. Given the growing importance of this topic, there has been recent work on modeling NFs, that can be classified into a few high-level categories based on domain knowledge [14, 21] or manual code analysis [26, 10]. However, most of these models are abstract and approximate. Further, these models are developed either semi-manually and/or using high-level domain knowledge. We need a systematic and automated methodology and framework to develop accurate NF models, which is the focus of this paper.

Developing accurate models of arbitrary NFs is challenging for a number of reasons. Many NFs operate on network packets based on states stored internally in the NF. This NF state processing may be hidden deep in the NF code, and may not be uncovered easily from either blackbox testing methods or high-level domain knowledge. The same network function may be implemented in a number of ways - thus implementations of the same network function by different vendors may not be modeled correctly by the same abstract model of the function. Currently there is no standard and structured way of writing NF code, which makes this problem more challenging. The research community thus far has mostly analyzed click-based NFs [18], which are not the typical network functions in deployment today.

Our approach to NF modeling builds on prior work to develop a new methodology based on program analysis. We strongly leverage recent advances in code analysis techniques in our system - NFactor to build a network function modeling tool. NFactor analyzes the source code of a given network function to automatically generate an abstract packet forwarding model for the NF. As mentioned above, we focus on just the NF forwarding model, motivated by the network verification/troubleshooting/testing and service chaining policy composition applications. We are well aware that it is unlikely that the source code of commercial NFs will be

HotNets-XV, November 09 - 10, 2016, Atlanta, GA, USA

 ISBN 978-1-4503-4661-0/16/11…$15.00

DOI: http://dx.doi.org/10.1145/3005745.3005754

freely available to analyze and build models. Our goal is to make our tool available to NF vendors who can run it on their proprietary code and provide only the resultant models to network operators for verification, troubleshooting and testing purposes (e.g., to construct effective compliance tests). Our thesis is that a combination of domain knowledge, code analysis and testing will ultimately be needed to build highly accurate models. Of these approaches, code analysis techniques have not received as much attention, which is what we focus on here. In the future, we plan to extend this work to model other functionality of NF's in addition to the forwarding.

The contributions of this paper are the following:

- We design and develop a NF code analysis methodology NFactor, which does not assume any specific code structure, unlike prior approaches. For this purpose, we leverage advanced tools and techniques from the program analysis community.
- We build the NFactor tool implementing the code analysis methodology.
- We demonstrate the effectiveness of the tool with two open source network functions showing that the models generated are functionally complete and accurate for the purposes of network verification and chaining. NFactor can process two commonly used NFs in less than 10 minutes.

2. OVERVIEW

In this section, we first review the background on program analysis techniques that enable model synthesis. Then we show how program analysis can be applied with an example of NF code. Finally, we present our model format and the overview of our methodology.

2.1 Background

Program Slicing. A program slice is a minimum set of statements in a given program that lead to certain behaviors. The behavior is expressed as the values of variables in a certain statement. Program slicing has been shown to be useful for program debugging, parallelization, integration, etc. [28].

The basic methodology to get a program slice is to analyze the dependency between program statements. Within one statement in a given program, the value of the left-hand-side (LHS) variable depends on that of the right-hand-side (RHS) variables; and between statements, the value of an RHS variable in a statement depends on the preceding statements where that variable is on the LHS. This dependency analysis results in a "static" program slice where all statements in that slice *might* lead to the final behavior. A "dynamic" program slice is all statements that *really* lead to the final behavior, which requires execution analysis based on actual variable values [3]. Several research projects have improved program slicing techniques - e.g., inter-procedure slicing, system dependency analysis [13, 11], etc.

Symbolic Execution. Symbolic execution is another program analysis approach that substitutes one or several program variables by symbolic values. As the program runs, whenever a branch instruction is met, the program is forked and both branches proceed. The program state variables and path constraints are kept by each path individually. At the end of a path's execution, concrete values of symbolic variables are computed according to the path constraints [6, 5].

Symbolic execution can exercise all possible execution paths, but path explosion can happen with a large code base. Various efforts have made symbolic execution practical in NF verification. To reduce the branching factor (number of branches at each branch instruction), Dobrescu et al. [9] and SymNet [26] propose to write NF programs in a style with bounded loops and data structures, and BUZZ [10] constrains the number and scope of symbolic variables.

NF state analysis. OpenNF [12] is a flow state management framework that enables joint control with SDN and Network Functions Virtualization (NFV) frameworks. An important issue when integrating existing NFs into OpenNF is to identify state variables in NF programs. StateAlyzer [16] defines features of state variables in an NF program, leveraging several program analysis techniques (e.g., program slicing, system dependency analysis) to identify them. These features are listed below.

- *Persistent:* the variable has a lifetime longer than the packet processing loop.
- *Top-level:* the variable is actually used during the packet processing.
- *Updateable:* the variable's value is updated during packet processing, i.e., usually an LHS variable.
- *Output-impacting:* the variable impact variables in the packet output function.

Our work is built on top of these features and uses more fine-grained categorization for variables.

2.2 An NF Example: Load balancer

We use a layer-4 load balancer as an example of the NF source code and illustrate how the above techniques can be used to synthesize NF models. Figure 1 is its implementation based on the model in [14]. The high-level logic of the code is as follows. Inbound packets that have the IP addresses/ports of clients and the load balancer, would be mapped to the IP address/ports of the load balancer and servers. If an inbound packet is a new flow never seen before, one of the backend servers is picked for the mapping and the mapping is stored; otherwise, the mapping is read from the dictionary and used for the address/port translation. For the outbound packets, the packets of existing flows would have address/port translation in the same way, but outbound packets of a non-existing flow would be dropped (i.e., only inbound packets can initiate address/port translation mapping.).

From reading this code, we gain important insights about NFs. First, different configurations can lead to different program behaviors. For example, the variable "mode" is used to configure how a backend server is selected for a new flow,

```
1  from scapy.all import *
2  # Constants
3  ROUND_ROBIN = 1
4  MTU = 1500
5  # Configurations
6  mode = ROUND_ROBIN
7  LB_IFACE = "eth0"
8  LB_IP, LB_PORT = "3.3.3.3", 80
9  servers=[("1.1.1.1", 80), ("2.2.2.2", 80)]
10 # Output-Impacting States
11 f2b_nat, b2f_nat={}, {}
12 rr_idx = 0
13 cur_port = 10000
14 # Log States
15 pass_stat, drop_stat=0, 0
16 # callback function
17 def pkt_callback(pkt):
18   global drop_stat, pass_stat, rr_idx, cur_port
19   si, di = pkt[IP].src, pkt[IP].dst
20   sp, dp = pkt[TCP].sport, pkt[TCP].dport
21   if dp == LB_PORT: # pkt from client to server
22     cs_ftpl, sc_ftpl = (si,sp, di,dp), (di,dp, si,sp)
23     if cs_ftpl not in f2b_nat:  # new connection
24       if mode == ROUND_ROBIN:
25         server = servers[rr_idx]
26         rr_idx = (rr_idx+1) % len(servers)
27       else: # Hash to a backend server
28         server = servers[hash(si) % len(servers)]
29       n_port = cur_port
30       cur_port+=1
31       cs_btpl = (LB_IP,n_port, server[0],server[1])
32       sc_btpl = (server[0],server[1], LB_IP,n_port)
33       f2b_nat[cs_ftpl],b2f_nat[sc_btpl]=cs_btpl,sc_ftpl
34       nat_tpl = cs_btpl
35     else: # existing connection
36       nat_tpl = f2b_nat[cs_ftpl]
37   else: # pkt from server to client
38     sc_btpl = (si, sp, di, dp)
39     if sc_btpl in b2f_nat:
40       nat_tpl = b2f_nat[sc_btpl]
41     else: # no initial outbound traffic is allowed
42       drop_stat+=1
43       return
44   pass_stat+=1
45   pkt[IP].src,pkt[TCP].sport = nat_tpl[0],nat_tpl[1]
46   pkt[IP].dst,pkt[TCP].dport = nat_tpl[2],nat_tpl[3]
47   for f in fragment(pkt[IP],fragsize=MTU-len(Ether())):
48     sendp(Ether()/f, iface=LB_IFACE)
49 def LoadBalancer():
50   sniff(iface=LB_IFACE,prn=pkt_callback,filter="tcp")
51 if __name__=="__main__":
52   LoadBalancer()
```

*varaiable format (cs|sc)_(f|b)(s|d)(i|p|tpl): sc|cs is direction between client and server, f|b is a side of frontend/backend, s|d is source/destination and i|p|tpl is IP, port or 4-tuple.

Figure 1: Load balancer code and a slice (highlighted)

and it can be either round robin or random hash. Some existing NF models [14] fail to capture this detail. Second, state variables are dynamically updated as packets are processed. It causes the action on packets to be different at runtime. In this example, whether a flow's 4-tuple is stored in the dictionary is a state that causes the actions on the flow's 1st packet and that on the remaining packets to be different. Third, NF programs naturally have two key pieces of logic, i.e., packet processing and state management. Thus, we could use program analysis to refactor the code and identify the minimum set of statements that capture such forwarding logic. For example, in Figure 1, the highlighted lines are a (dynamic) program slice where the load balancer relays the first packet of a flow. This small code snippet captures the forwarding behavior of this NF. Identifying this snippet automatically from the original code would improve the efficiency of manual analysis and automatic verification.

2.3 NF Model

Referring to various existing NF models [26, 14, 20, 10, 21, 27] and the current hardware programmability [30, 19, 4], NFactor adopts an OpenFlow-like model with a stateful data plane extension. The abstract model is expressed as tables in Figure 2a, and each table describes the packet processing logic under a certain configuration (e.g., c_1 in the figure). Each entry in the table represents certain processing logic and consists of match and action fields. As a stateful data plane, the match/action fields operate on both flows and state variables. The match is executed on flows and states, and the action not only forwards packets (with possible transformation) but also triggers state transition.

In the table c_1 in Figure 2a, if an incoming packet matches a flow pattern f_1, the internal state is in s_1 and a predicate $P(f_1, s_1)$ is satisfied, then the packet is sent out with possible transformation $Fwd(f_1, s_1)$ and the internal state is transited to $Upd(f_1, s_1)$. In our running example code, each frontend incoming packet would be checked to determine whether the flow was seen before (in the address/port translation mapping), which is the predicate of flow and states in the match field; the first packet of each new flow triggers address and port translation (stored in a dictionary), which is a state transition; and sendp() is an action on packets.

2.4 NFactor Model Extraction

The focus of this paper is to refactor existing NF programs and extract the logic that can fit in the previous model. NFactor achieves this in three steps.

NFactor first leverages StateAlyzer and program slicing on the NF code and obtains a packet slice and a state slice (Figure 2b). Then NFactor leverages control flow analysis and symbolic execution to find out all possible execution paths in the union of both slices (Figure 2c), where each execution path is actually an entry in the tables in the NFactor model. Finally, flow/state match/action fields are obtained by refactoring the logic in each execution path (Section 3.1).

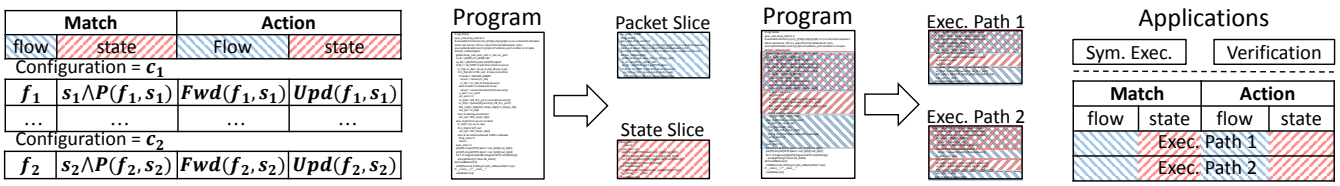
The model in Figure 2d not only is useful in code analysis/debugging but also benefits several existing network verification applications. For example, NFactor extracts the key (minimum) logic of a NF program, i.e., forwarding logic and state transition logic, so that the symbolic execution can be sped up. The state transition logic can be used to build a finite state machine, which is proposed and used in network testing solutions [10].

3. METHODOLOGY

We detail our algorithm to refactor NF programs and then discuss several issues in applying the algorithm to general NF code.

3.1 Algorithm

Code structure. The NFactor algorithm is shown in Algo-



(a) NFactor Model (b) Step 1: slice program (c) Step 2: find execution paths (d) Step 3: fill in model

Figure 2: NFactor overview

Algorithm 1 NF Program Slicing

```
    Input: prog, Output: table
    ▷ Identify packet slice
1:  for stmt in prog do
2:      if stmt calls PKT_OUTPUT_FUNC then
3:          s := BackwardSlice(stmt, Vars(stmt.RHS))
4:          pktSlice := pktSlice ∪ s
    ▷ Categorize variables
5:  (pktVar, oisVars, cfgVars) := StateAlyzer(pktSlice)
    ▷ Identify state slice
6:  for stmt in prog do
7:      if Vars(stmt.LHS) in oisVars then
8:          s := BackwardSlice(stmt, Vars(stmt.LHS))
9:          stateSlice := stateSlice ∪ s
    ▷ Find out execution paths
10: execPaths := FindExecPaths(pktSlice ∪ stateSlice)
    ▷ Refactor logic into model
11: for p in execPaths do
12:     cndStmts := GetConditionStatements(p)
13:     config := cndStmts ∩ cfgVars
14:     match := (cndStmts ∩ pktVars, cndStmts ∩ oisVars)
15:     action := (p ∩ pktSlice, p ∩ stateSlice)
16:     table[config].add( <match, action> )
```

rithm 1. The input of the algorithm is an NF program and the output is tables that satisfy the NF model 2a. NFactor makes two standard non-limiting assumptions on the code structure. First, since the NF program needs to continuously processes incoming packets, there exists a packet processing loop. The same assumption has been made in StateAlyzer [16]. Second, NF programs usually use standard library or system functions to exchange packets with the OS kernel/network devices - thus, NFactor leverages this knowledge to locate packet read/write statements in the program. And NFactor identifies the variable that stores a packet via fetching the return value of the packet input function or the argument of the packet output function.

Packet processing slice. NFactor first computes the packet processing slice (line 1-4). It starts from each packet output function and performs backward slicing, and returns the union of all slices as the packet processing slice.

State variables. NFactor extends the variable categories in StateAlyzer as listed in Section 2.1. We show the categories together with those variables in the load balancer example in Table 1. According to their definition, Line 5 in the algorithm identifies packet variables (pktVar), output-impacting state variables (oisVar), and configurations variables (cfgVar). Different from StateAlyzer, NFactor inputs the packet processing slice instead of the whole program so it reduces the amount of code to process.

State transition slices. Next, NFactor computes the state transition slice (line 6-9). It iterates over all statements in the program; if an oisVar appears on the left-hand-side (LHS) of a statement, the statement is an assignment to that oisVar, thus a backward slice is computed for this state update. And finally, the state processing slice is the union of all these individual slices.

The *packet processing slice* and the *state transition slice* capture the key forwarding logic of an NF program. NFactor then computes all possible execution paths in the union of the two slices (line 10) using symbolic execution. Symbolic execution on NF programs can be sped up using techniques in Section 2.1.

Refine execution paths. Finally, NFactor refines the logic in each execution path and fills in the NFactor model (line 11-16). On each execution path, the statements with conditionals are collected, and the conjunction of these condition statements is the match field in the model. The intersection of condition conjunction and cfgVars constitutes the model configurations. The packet-matching and state-matching field are computed by intersecting the condition conjunction with the pktVars and oisVars respectively. Similarly, the actions on packets and states are obtained by intersecting the execution path with the packet processing slice and the state transition slice.

Table 1: NFactor variable categorization and examples

Category	Features	In code example
pktVar	packet I/O function parameter/return value	pkt
cfgVar	persistent, top-level, not updateable	mode, LB_IP
oisVar	persistent, top-level, updateable, output-impacting	f2b_nat, rr_idx
logVar	persistent, top-level, updateable, not output-impacting	pass_stat, drop_stat

3.2 Discussion

Execution Paths. We use symbolic execution to find all the execution paths. However, sometimes the complexity of NF programs makes symbolic execution unable to complete in tractable time [16]. For example, assume that the number of iterations of a while loop depends on a symbolic variable (say an integer). Then static analysis cannot figure out how many times the loop would be executed, and symbolic execution would try out all possible inputs (from INT_MIN to INT_MAX), which causes path explosion.

NF programs typically will not contain input-dependent loops, or they can be written or modified according to the style in [26, 9], to ensure loops are bounded. During the symbolic execution, the number of symbolic variables and their scope should be constrained to reduce the branching

```
1  sockfd = listen(...);
2  for(;;){
3    cltfd=accept(sockfd, ...);
4    server = servers[idx++]; idx%=N;
5    if(fork()){
6      bind(srvfd, ...);
7      connect(srvfd, server.serv_addr);
8      for(;;){
9        FD_SET(cltfd, &rdfds); FD_SET(srvfd, &rdfds);
10       sr = select(..., &rdfds, ...);
11       if (FD_ISSET(cltfd, &rdfds) )
12         read(cltfd, buf, ..), write(srvfd, buf, ..);
13       else
14         read(srvfd, buf, ..), write(cltfd, buf, ..);
15 } } }
```

Figure 3: Key program statements of *Balance*

space [10]. In the scope of the NF programs in our study, these techniques are feasible to enable symbolic execution in exploring execution paths.

Hidden States. NFactor models NF behaviors at the packet level (as shown in our illustrative example). However, a set of NFs are developed based on higher-level libraries such as TCP socket or httplib in Python. For example, *balance* [1] is another implementation of a layer-4 load balancer (Figure 3). It accepts TCP connections from clients (line 3), chooses a backend server, and then *balance* forks itself and creates a TCP connection to the server (line 7); finally in the forked process, *balance* relays traffic to the server.

This kind of NF has hidden states not shown in the NF programs, but hidden in the OS. And these hidden states influence the final packet processing result. For example, each TCP connection has its own state transition diagram (including states such as LISTEN, ESTABLISHED, etc.), and data packets without 3-way handshake established would be dropped.

Analyzing an NF program itself does not capture these stateful behaviors. We propose to fall back to analyzing packet level operations by unfolding these wrapped-up functions (e.g., listen(), connect()). NFactor replaces these functions/system calls with packet level operation together with the TCP state transition.

Code Structure. Code structure is another challenge to apply NFactor. We summarize four typical code structures for NF programming (Figure 4). Figure 4a 4b and 4c show code structure of one main processing loop, a loop with callback function and consumer-producer loops respectively. Our code example (Figure 1) has the structure of a loop with callback function 4b. The code structure of Figure 4b and 4c are easy to transformed into that in Figure 4a. Thus, NFactor can be easily applied into these three kinds. We believe that these assumptions of code structures hold for most NFs.

Figure 4d shows an NF program structure with nested loops. The outer loop processes the first few control packets of a flow (TCP connection), while the inner loop processes data packets. In this case, it is hard for NFactor to find out per-packet execution path from the beginning of either loop.

Essentially, the two loops executes independently in two processes after fork(). Thus, we propose to transform this code structure into one single loop format in Figure 4a. The

```
def MainLoop():
  while True:
    pkt = Read(IFACE)
    if state[pkt].tcpState!=ESTABLISHED:
      ProcessCtrlMsg(pkt, state)
      Send(pkt)
    else:
      ProcessDataMsg(pkt, state)
      Send(pkt)
MainLoop()
```

Figure 5: Nested loop transformed into one loop

code structure after transformation is in Figure 5, where TCP function unfolding techniques in the previous section are required; and finally NFactor can be applied to refactor the transformed code.

Drop Action. Usually, the packet drop action in NF programs is not explicitly specified. For example, in our running code (Figure 1) line 43, the packet processing procedure returns without performing any actions on the packet. Thus, in NFactor model, we define a low-priority default action to be drop. If there is no explicit action (e.g., sendp()) on packets in an execution path, the action to the packets is also drop.

4. APPLICATIONS

NFactor is a tool that can be used to model a variety of NFs. The model is useful for many network management applications such as verification, troubleshooting, and service deployment, which we describe here. In the context of NFV, network functions are decomposed into micro-services, which can be recomposed into new network services. Using NFactor, we can obtain models for well-known and new NFs.

Network Verification. Network verification [17, 15, 26] has been widely used to detect configuration errors. A key challenge in verifying NF is the lack of NF models. NFactor models can be used for verification in two ways.

1) Model checking: A straightforward way to verify the NF is to run model checking on its code. However, it incurs high overhead due to the complexity of NF code structure [26]. Running model checking using symbolic execution on our model can significantly reduce the overhead compared to original execution, as we show in our evaluation.

2) Extending stateless verification: Our model is similar to Openflow match-action abstraction. Since existing data plane verification tools, e.g., HSA, are based on the match-action abstraction, we can extend them to stateful verification. Each rule is modeled as a network transfer function: $T(h, p, s)$, where h is the packet header, p is the port, and s is the state in the model. With the extended transfer function, we can handle stateful verification. We have plugged in the model from NF to a stateful verification tool [27] and demonstrated the models' usefulness in verification.

Service Policy Composition. Different customers and applications may require different service chains, which need to be composed together correctly. Consider two service chaining policies: *{FW, IDS}* and *{LB}*. What should be the right order after composition, *{FW, IDS, LB}* or *{FW, LB, IDS}*? Existing work PGA [22] uses the NF model to de-

```
# IFACE: packet input
  interface

def MainLoop():
  while True:
    pkt = Read(IFACE)
    Process(pkt, state)
    Send(pkt)

MainLoop()
```
(a) One processing loop

```
def Callback(pkt):
  Process(pkt, state)
  Send(pkt)

# sniff: packet input
  function in library with
   interface and callback
  function as parameters

sniff(IFACE, Callback)
```
(b) Callback

```
def ReadLp():
  pkt = Read(IFACE)
  queue.add(pkt)
def ProcLp():
  pkt = queue.pop()
  Process(pkt, state)
  Send(pkt)

thread.start(ReadLp)
thread.start(ProcLp)
```
(c) Consumer-producer

```
def MainLoop():
  while True:
    clt=socket.accept()
    if os.fork()==0:
      srv=socket.connect()
      while True:
        buf = clt.recv()
        Process(buf, state)
        srv.send(buf)
MainLoop()
```
(d) Nested loop

Figure 4: typical NF code structures

termine the orders. It generates the input and output space constraints of each NF based on its behavior model. However, PGA uses the model described in Pyretic [24], which only describes stateless simple functions. Our model can be used as input to PGA to perform accurate service chain composition.

Testing. Active testing is commonly used to test the compliance of network policy. However, generating the test packets intelligently to test all the possible scenarios is challenging. BUZZ [10] creates the testing packets by using the NF models. However, their model is generated manually from domain knowledge so it may not be complete or even accurate. NFactor is complementary to BUZZ: the NFactor model can be used to guide the generation of testing packets.

5. IMPLEMENTATION AND EVALUATION

We implement NFactor using LLVM giri [25] and KLEE [5]. LLVM giri is a program slicer that can identify packet slices and state slice in Algorithm 1. KLEE is a symbolic executor that can be used to find out execution paths in NFactor algorithm. We studied 2 NF programs: snort 1.0 [2] and balance 3.5 [1], and the result is shown in Figure 6 and Table 2.

Match		Action	
Flow	**State**	**Flow**	**State**
mode = RR			
f	idx	send(f, server[idx])	(idx+1)%N
mode = HASH			
f	*	send(f, server[hash(f)%N])	*

Figure 6: NFactor output for balance

Functional Validation. We show the logic we extracted from the *balance* NF in Figure 6. Due to limited space, we do not fill in the whole logic in the NFactor model. If balance is configured to be round robin mode, the round-robin index is figured out as output-impacting states, new flows are sent to the indexed backend server; and the index is increased circularly. If balance is configured in hash mode, there is no index state, and the backend server is picked randomly.

Table 2: NFactor on Snort and Balance

	LoC			Slicing	# of EP		SE time	
	orig	**slice**	**path**	**Time**	**orig**	**slice**	**orig**	**slice**
snort	2678	129	112	158s	>1000	3	>1hr	484ms
balance	1559	64	34	79s	20	10	3.4s	404ms

LoC: lines of code; **EP:** execution paths; **SE:** symbolic execution. **orig.:** the original code; **slice:** the packet and state slice; **path:** one execution path in the slice.

Efficiency. The two NF programs consist of thousands of lines of code (excluding comments); while only around one hundred lines of code belong to packet/state slice (129 in snort and 64 in balance), and a specific execution path has even fewer lines. Program slicing can be done relatively fast (158s for snort and 79s for balance). In addition, the complexity of the packet/state slices is much smaller than the original code and both the number of execution paths and the symbolic execution time are reduced compared with the original program. Finally, due to the diversity of NF programs, the reduction in complexity varies. In our experiment, snort's logic is more complex (it checks many fields in the packets), and benefits more from NFactor. Note that the number of paths are significantly reduced because the path explosion caused by external libraries are eliminated. The pruned code includes logs, failure handling, locking, etc.

Accuracy. To test whether NFactor outputs a logically equivalent forwarding model with the original program, we use symbolic execution to exercise all possible execution paths on both sides. We have compared and confirmed that the two sets of paths are the same. Further, we generate random inputs (i.e., packets) to both NFactor model and the original program, and test whether they output the same result. We repeat the experiments for 1000 times for the 2 NFs respectively, and the outputs in each experiment are the same. In the future work, we plan to explore more rigorous proofs of the logical equivalence of NFactor.

6. RELATED WORK AND CONCLUSIONS

The NF model used in this paper is inspired by a long line of research on understanding and modeling NFs [14, 22, 21, 12, 23, 7, 8]. In particular, [23, 12] models the internal states of NFs. Our work focus on the NF forwarding abstraction. Our model is similar to Openflow stateful extensions [19, 29] but focus on NFs. Middlebox verification needs NF models. BUZZ [10], which builds the FSM from domain knowledge. Panda [21] and SymNet [26] both propose a new modeling language and uses a SAT solver or symbolic execution to perform verification. Our work is orthogonal to theirs: our code analysis can automatically generate the model defined in their language. This will be a part of our future work.

In this paper, we present NFactor, a tool that automatically synthesizes NF *forwarding* model from analyzing the source code. We focus on accurate NF forwarding modeling for the purpose of network verification. As future work, we plan to explore more usage of the model generated. We plan to compare the model generated by NFactor with existing models generated from manual inspection. We will test it on more

open source NFs and perform compliance testing to further evaluate its accuracy.

7. REFERENCES

[1] https://www.inlab.de/balance.html.

[2] https://www.snort.org/.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Notices*, volume 25, pages 246–256. ACM, 1990.

[4] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2), April 2014.

[5] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[6] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[7] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. Revealing middlebox interference with tracebox. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, 2013.

[8] M. Dischinger, M. Marcon, S. Guha, K. P. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling end users to detect traffic differentiation. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, 2010.

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

[10] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar. Buzz: Testing context-dependent policies in stateful networks. In *Proc. NSDI*, 2016.

[11] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[12] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[14] D. Joseph and I. Stoica. Modeling middleboxes. *Netwrk. Mag. of Global Internetwkg.*, 2008.

[15] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI*, 2012.

[16] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for nfv: simplifying middlebox modifications using statealyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 239–253, 2016.

[17] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[19] M. Moshref, A. Bhargava, A. Gupta, M. Yu, and R. Govindan. Flow-level state transition as a new switch primitive for sdn. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[20] A. Panda, K. Argyraki, M. Sagiv, M. Schapira, and S. Shenker. New directions for network verification. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[21] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker. Verifying isolation properties in the presence of middleboxes. Technical report: arXiv preprint:1409.7687,2014.

[22] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proc. ACM SIGCOMM*, 2015.

[23] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2013.

[24] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular SDN Programming with Pyretic. *USENIX*, 38(5), October 2013.

[25] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 139–152, New York, NY, USA, 2013. ACM.

[26] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proc. ACM SIGCOMM*, 2016.

[27] B. Tschaen, Y. Zhang, T. Benson, S. Benerjee, J. Lee, and J.-M. Kang. SFC-Checker: Checking the Correct Forwarding Behavior of Service Function Chaining. In *IEEE SDN-NFV Conference*, 2016.

[28] M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[29] S. Zhu, J. Bi, and C. Sun. Sfa: Stateful forwarding abstraction in sdn data plane. In *Presented as part of the Open Networking Summit 2014 (ONS 2014)*, 2014.

[30] S. Zhu, J. Bi, C. Sun, and C. Wu. Sdpa: Enhancing stateful forwarding for software-defined networking. In *Proc. International Conference on Network Protocols*, 2015.