Lancet: Extracting Forwarding Models from NF Programs

Wenfei Wu
Tsinghua University

Ying Zhang
Hewlett Packard Labs

***Abstract*—**
Network functions (NFs) or middleboxes have been widely deployed in modern ISPs and enterprise networks. Due to the proprietary and complex nature of these NFs, they have become a critical source of misconfigurations and errors in the network. Moving towards Network Function Virtualization (NFV), the network management is becoming more challenging given the increasing agility and frequency in configuration changes. Network verification has been widely used to statically analyze errors in switches and routers. However, applying network verification to NFs has significant challenges because of the lack of proper NF models that unveil the NFs' behavior to the packets. Accurately modeling complex NFs is critical to several network management tasks, including verification, active testing, debugging and state management. Existing work to develop NF models generate approximate models using manual and time consuming techniques. This paper proposes a system called Lancet that automatically synthesizes NF models by applying program analysis techniques. Lancet uses program slicing and symbolic execution on NF source code, and generate a concise and accurate model automatically. Our preliminary evaluation shows the feasibility of Lancet to extract a concise model from NF source code to an abstract model.

I. Introduction

Network middleboxes or network functions (NFs), like firewalls, deep packet inspections, network address translates, caches, load blancers, have been widely deployed in the network. Due to their proprietary and complex nature, they often complicate the process of network verification and fast troubleshooting. Network verification, which has been widely used to diagnose problems on switches and routers, have encountered a critical challenge with NFs because there is no standardized forwarding model of these NFs.

Recently, there have been several recent efforts to improve our understanding of NF operations and create models that can be used in verification and troubleshooting applications. These modeling efforts can be roughly classified into blackbox and whitebox based techniques. Of the blackbox approaches, most efforts are based on domain knowledge (e.g., [1], [2], [3]) or based on active testing and measurement [4]. Other approaches analyze the code of either open source NFs or Click-based NFs but most rely on manual techniques [3], [4]. Further, open source NFs or Click-based NFs are typically not deployed in production environments; production NFs from vendors are written using a variety of code styles. Additionally, each of the prior approaches has its technical limitations and we believe that a combination of approaches will ultimately yield the most robust and accurate models. Code analysis techniques have not been exploited fully in the area of NF modeling, and hence our goal is to build on top of existing code analysis systems.

We extend our prior work [5] to develop NF packet forwarding models based on source code analysis of stateful NF programs. Our goal is to analyze general NF code structures and develop an automated framework to generate accurate abstract NF models. Similar to [6], we leverage advances in code analysis techniques for our purposes. As mentioned, we focus on modeling abstract NF forwarding behaviors, like prior work. The challenges in doing so are to uncover all necessary state information in the NF code that affects NF forwarding behavior. Note that these may not be easily discoverable via blackbox testing or domain knowledge based modeling techniques. From a practicality standpoint, we realize that commercial production grade NF programs will not be available to analyze and model. On the other hand, it will be possible to develop tools like ours that can be used by NF vendors to run on their code and provide the abstract model to network operators for verification and troubleshooting purposes.

In the poster, we show a code analysis methodology to analyze general NF programs, using advances in program analysis and generate a concise subset of the code that contains the NF forwarding logic. We build a tool called Lancet, which can systematically and efficiently extract the model from a whitebox NF. When applying Lancet on open source NFs, we demonstrate the modeling process can be done in acceptable time and outputs concise and accurate models.

II. Lancet Methodology

Our work is built on existing technologies in the program analysis community. In this section, we first review the background information that enables model synthesis. And then we demonstrate our methodology using a sample of NF code.

A. Program analysis techniques

Program Slicing. Program slicing is used to find out all statements in a program that lead to the value of a variable at a certain statement (called *backward slice*) or all statements that are influenced by a variable at a statement (called *forward slice*). The basic methodology to compute a program slice is *influence analysis* (or dependency analysis). Within a statement, a set of variables whose values are altered (called DEF) are influenced by the set of variables referred (called REF). For example, the right-side variable of an assignment statement influences its left-side variables. Across statements, a variable at a statement is influenced by its preceding statement with that variable as DEF, and condition statements branching to that statement.

Symbolic Execution. Symbolic execution (SE) substitutes one or several program variables by symbolic values; as

```
 1 #                    +----------------+
 2 #    client  ------+--incoming--+----> servers
 3 # frontend <------+--outgoing--+----- backend
 4 #                    +----------------+
 5 from scapy.all import *
 6 from sets import Set
 7 TRANSPARENT, FIREWALL = 0, 1         # constants
 8 servers = [("1.1.1.1"), ("2.2.2.2")] # config
 9 FW_IFACE = "eth0"                    # config
10 mode = FIREWALL                      # config
11 seen = Set()          # output-impacting state
12 drop_stat, pass_stat = 0, 0            # logs
13 def PktCb(pkt):              # callback function
14   global drop_stat, pass_stat, seen
15   if mode == TRANSPARENT: # for debugging
16     sendp(pkt, iface=FW_IFACE)
17     return
18   si, di = pkt[IP].src, pkt[IP].dst
19   sp, dp = pkt[TCP].sport, pkt[TCP].dport
20   allowed = False
21   if di in servers:            # incoming
22     seen.add((di, dp, si, sp))
23     allowed = True
24   elif (si, sp, di, dp) in seen:
25     allowed = True             # outgoing (allowed)
26   if allowed:
27     sendp(pkt, iface=FW_IFACE)
28     pass_stat += 1
29   else:                        # outgoing (denied)
30     drop_stat += 1
31 def Firewall():
32   sniff(iface=FW_IFACE, prn=PktCb, filter='tcp')
33 if __name__=="__main__":
34   Firewall()
```

Fig. 1: Stateful firewall program and a slice (highlighted)

the program executes, whenever a branch instruction is encountered, the program is forked and both branches proceed. Branching conditions are kept as path constraints by each path separately. The execution proceeds until the end of each path, then concrete values of symbolic variables are computed according to the path constraints.

NF State Analysis. StateAlyzr [6] is proposed to automatically identify state variables (using program slicing and system dependency analysis). It proposes four features for variables in NF programs and identifies state variables accordingly. The four features are (1) *persistent*: a variable has a lifetime longer than the packet processing loop, (2) *top-level*: a variable is used during the packet processing, (3) *updateable*: a variable's value is updated during the packet processing, and (4) *output-impacting*: a variable influences the variable values in the packet output function.

B. Methodology using an example

We use a stateful firewall based on the model in [2] to illustrate how the above techniques can be used to synthesize NF models. The firewall program is shown in Figure 1, and is deployed between client and servers. In this firewall, incoming traffic (i.e., destination IP is to servers) is always allowed and recorded in a set "seen" (line 21-23). Outgoing traffic is allowed if it is initiated by an incoming flow (line 24-25, recorded in "seen") and denied otherwise (line 29-30).

First, Lancet perform backward slicing starting from packet output function `sendp()`, so that all program statements about *packet forwarding* are obtained (e.g., the highlighted

	Match		Action	
	Flow	**State**	**Flow**	**State**
Fire wall	ALLOW={r1, r2, ...}			
	f$\sqsubseteq$ALLOW	-	f[output]:=IFACE	seen:=seen$\cup${f}
	f	f$\in$seen	f[output]:=IFACE	-
	f$\not\sqsubseteq$ALLOW	f$\notin$seen	f[output]:=ϵ	-

Fig. 2: The model of the stateful firewall

lines in Figure 1). Second, Lancet identifies state variables in the program[1], and performs backward slicing to get the *state transition slice*. Third, both slices are merged and symbolically executed, so that all execution paths are identified (e.g., there are the three branches from line 21 to 25.). Finally, each execution path is refactored and put into a stateful match-action table, i.e., statements about state variables are put into state match/action fields and statements about packet processing are put into flow match/action fields.

The final model is in Figure 2, and this modeling process can be complete in 3 minutes using Frama-C [7] and LLVM KLEE [8]. In the implementation and experiments, we also overcome several challenges: we identify and eliminate program structures (e.g., loops) that easily cause path explosion in SE, we improve program slicing tool to make the slicing extended to external storage (file systems), and we also induce the final model to a concise NF modeling language SNAP [9].

III. Conclusion and Future Work

We propose Lancet, a framework to synthesize NF forwarding model from source code. We use a stateful firewall as a showcase of how models are extracted. In the future, we would test on more NF programs, and leverage the extracted models in existing network verification and troubleshooting applications.

References

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

[2] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying Reachability in Networks with Mutable Datapaths," in *Proc. NSDI*, 2017.

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

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

[5] W. Wu, Y. Zhang, and S. Banerjee, "Automatic synthesis of nf models by program analysis," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 29–35, ACM, 2016.

[6] 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)*, pp. 239–253, 2016.

[7] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c," in *International Conference on Software Engineering and Formal Methods*, pp. 233–247, Springer, 2012.

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

[9] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proc. ACM SIGCOMM*, 2016.

[1]State variables are persistent, top-level, updateable and output-impacting according to StateAlyzr [6]