Redundant Logic Elimination in Network Functions

Bangwen Deng and Wenfei Wu
Tsinghua University

ABSTRACT

In current NFV ecosystem, most software NFs delivered by NF vendors tend to be monolithic with multiple features. In runtime, an NF is configured with a subset of its features turned on/off. By our code analysis and primary test, we find that some of the (runtime) unused features are executed silently, which wastes CPU cycles and memory. We propose to use program analysis techniques to eliminate runtime redundant logic in NF code so that the NF performance can be accelerated. We prototype our idea and test it on Snort to show the feasibility.

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**;

KEYWORDS

network functions, redundant logic elimination

1 INTRODUCTION

Software network functions (NF) have already got wide deployment in various networks (e.g., cloud networks, enterprise networks, mobile core networks). These NFs are usually developed by NF vendors, who can be traditional network appliance vendors (CISCO, PAN) or software companies. To make their NFs competitive and appealing in the market, each vendor tends to add more features to their NFs. For example, a firewall can be integrated with features of NAT, DDoS attack detection (PAN firewall). In the runtime, the monolithic NF is usually configured with a subset of its multiple features turned on/off. A single version of monolithic NF software can help reduce vendor's cost to develop/maintain the code base as well as deliver the software to customers.

However, we also observe the side effect of building NFs in a monolithic way — *some unused features in the code waste CPU cycles and memory*. For example, if a firewall is configured to filter traffic based on IP addresses, all the port extraction, storage, and processing logic in the code should not take effect. But the redundant logic would reduce the packet-processing efficiency of the firewall.

Thus, we aim to give a solution that eliminates the runtime redundant logic in NFs, and we name our solution `NF-LRE`. `NF-LRE` has the code of NF and the runtime configuration as input, and outputs an executable with redundant logic eliminated. And the executable has the same logic and better performance compared with the original NF code.

`NF-LRE`'s redundant logic elimination consists of four steps. First, runtime configurations are input to the program for initialization, where a set of configuration variables have their values assigned. Second, `NF-LRE` applies constant propagation in the program so that several flows match expressions would have their variables replaced by constants. Third, `NF-LRE` applies constant folding for expressions so that a few flow match branches would have their conditions to be always logically true or false and thus pruned. Finally, `NF-LRE` applies dead code elimination where unused packet parsing (due to constant folding) is eliminated.

We prototype `NF-LRE` and perform primary test on Snort. `NF-LRE` can improve Snort's packet processing rate by 15%. Since `NF-LRE` optimizes and recompiles the NF code, it suits the scenario where the network configuration is relatively static. In the future, we would explore the solution that can support both redundant logic elimination and runtime reconfiguration.

2 MOTIVATING EXAMPLE

Figure 1 shows the simplified logic of Snort[2]. Snort has some configuration rules as input where it declares the action (accept/drop/alert) to a flow (5 tuple <protocol, sip, dip, sport, dport>, line 1-3). For each incoming packet, it by default parses all fields from layer-2 to layer-4 (line 10-21) and stores them in a global variable `net` (line 4-9). Snort then checks the packet with each rule field by field (line 29-37). If the packet matches a rule, Snort would take an action (accept, drop, alert; line 22-34).

In deployment, an administrator may want to block traffic between two subnets, where the port number does not matter and they are configured as a wildcard "any" (line 3), the configuration would be passed to the code variables. In packet processing, no matter what values the packet ports are, they are finally compared with a wildcard and the result

SIGCOMM Posters and Demos '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5915-3/18/08…$15.00
https://doi.org/10.1145/3234200.3234219

```
1  // Two example Snort rules:
2  // drop tcp 10.0.0.0/24 80 -> 10.1.0.0/24 50
3  // accept tcp 10.0.0.0/24 any -> 10.2.0.0/24
       any
4  struct {
5      ...
6      unsigned long sip, dip;
7      unsigned short sport, dport;
8      ...
9  } net;
10 void DecodeEthPkt(..., u_char *pkt){
11     DecodeIPPkt(pkt);
12 }
13 void DecodeIPPkt(..., u_char *pkt){
14     net.dip = ...
15     net.sip = ...
16     DecodeTCPPkt(pkt);
17 }
18 void DecodeTCPPkt(..., u_char *pkt){
19     net.dport = ...
20     net.sport = ...
21 }
22 void ApplyRules(){
23     while(1){
24         r = ... //get a rule
25         if(MatchRule(r)){
26             // take some actions
27             return ;
28 }   }   }
29 int MatchRule(r){
30     if( r->sip != net.sip ) goto bottom;
31     if( r->dip != net.dip ) goto bottom;
32     if( r->sport != net.sport) goto bottom;
33     if( r->dport != net.dport) goto bottom;
34     return 1;
35 bottom:
36     return 0;
37 }
38 void main(){
39     while(1){
40         pkt = ... // get packet
41         DecodeEthPkt(pkt);
42         ApplyRules();
43 }   }
```

Figure 1: Snort code (simplified)

is always true (line 32, 33). In the whole process, all layer-4 packet parsing and processing (e.g., matching) are in vain, wasting CPU cycles.

3 DESIGN AND PRELIMINARY RESULT

Following the example in the previous section, we leverage a program analysis approach to eliminate runtime unused logic. We assume there are only layer-3 rules configured (line 3). There are following four steps.

(1) Input the configuration to the program so that some variables are assigned with constant values. In the example, `r->sport` and `r->dport` are assigned with wildcards.

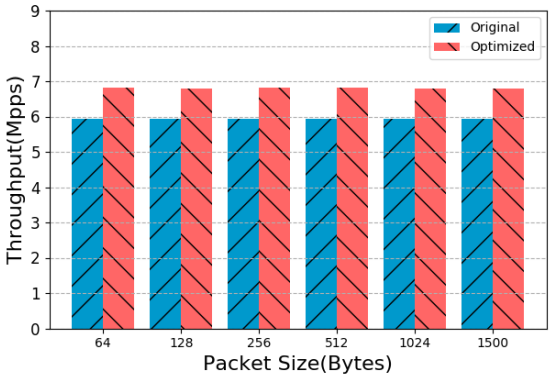

Figure 2: Performance Acceleration for Snort

(2) Propagate the constant value in the program. In the example, every appearance of `r->sport` or `r->dport` is replaced by a wildcard.

(3) Fold expressions whose value can be computed. In the example, the conditions of matching port are always true (line 32, 33). Thus, the two blocks can be folded.

(4) Eliminate dead code. Since matches on port numbers are eliminated, per-packet port number parsing becomes unnecessary. In other words, per-packet port numbers' values are assigned, but the value is not used in following logic; thus, they can be eliminated.

We use LLVM[1] to prototype `NF-LRE` and apply it to Snort. Figure 2 shows the acceleration of packet processing rate. The packet processing rate increases by about 15%.

4 RELATED WORKS AND CONCLUSION

Recent works on analyzing Snort configuration file mainly focus on the rules conflict and redundancy rules elimination, for example, *A. SaÃćdaoui et al.*[4] propose an approach to analyze configuration files and automatically detect rule conflicts or redundancy and remove them. This work does not have the code implementation in consideration. In code analyzing, *Wu. et al.*[5] propose to utilize the symbolic execution tool and program slicing techniques to figure out NF's execution paths. *StateAlyzr [3]* leverages several program analysis techniques (e.g., program slicing, system dependency analysis) to identify the state variables in an NF program. These program analyzing techniques enlighten us to further improve NF program performance.

To summarize, we observe that current NF software is usually designed with multiple features, and runtime deployment usually enables a subset of them. The unused features execute silently, causing the NF to waste runtime computational resources (e.g., CPU). We propose to combine NF match action logic with the program analysis techniques to eliminate the runtime redundant logic in NF code. And our prototype `NF-LRE` shows that Snort can be improved significantly by this means.

REFERENCES

[1] https://llvm.org/.

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

[3] 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, Santa Clara, CA, 2016. USENIX Association.

[4] A. Saadaoui, H. Benmoussa, A. Bouhoula, and A. Kalam. Automatic classification and detection of snort configuration anomalies - a formal approach. pages 27–39, 01 2015.

[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*, HotNets '16, pages 29–35, New York, NY, USA, 2016. ACM.