Redundant Logic Elimination in Network Functions

Bangwen Deng[‡], Wenfei Wu[‡], Linhai Song[†]

[‡]Tsinghua University [†]The Pennsylvania State University

ABSTRACT

Network functions (NFs) are critical components in the network data plane. Their efficiency is important to the whole network's end-to-end performance. We identify three types of runtime redundant logic in NFs when they are deployed with concrete configured rules. We propose to use compiler techniques (*e.g.*, program slicing, constant propagation, dead code elimination, symbolic execution) to optimize away the redundancy. We implement a prototype named NFReducer using LLVM. Our evaluation on two IDSes shows that after eliminating the redundant logic, the packet processing rate of the two IDSes can be significantly improved.

CCS CONCEPTS

• **Networks** → **Middle boxes / network appliances**; • **Software and its engineering** → *Compilers*;

KEYWORDS

Network Function, Software Optimization

ACM Reference Format:

Bangwen Deng[‡], Wenfei Wu[‡], Linhai Song[†]. 2020. Redundant Logic Elimination in Network Functions. In *Symposium on SDN Research (SOSR '20), March 3, 2020, San Jose, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3373360.3380832

1 INTRODUCTION

Virtualized network functions (NFs) are software appliances that process all traversing network traffic in the data plane. Thus, their efficiency in flow processing affects the whole network's end-to-end performance in a significant way (*e.g.*, latency accumulation, throughput bottleneck). However, we find that when an NF is deployed with concrete configured rules, it may conduct more than necessary packet processing logic, which consequently causes negative effects on its performance (*e.g.*, wasting CPU cycles, increasing memory usage) and further impacts the whole network.

In this paper, we define *redundant logic* in an NF as the piece of code whose execution does not influence the correctness of the NF's packet processing. And such redundant logic usually happens when an NF is configured with concrete *runtime* rules. We propose to take an NF instance's configured rules into account and leverage classic compiler optimization techniques to eliminate the redundant logic. Our proposed technique is orthogonal to the existing solutions that either accelerate individual NFs [26, 40] or parallelize/consolidate multiple NFs [15, 21, 23, 33, 35, 43].

The essential reason for the redundant logic is the mismatch of the protocol space in the development and that in the deployment. Network protocols are organized in a layered stack (*e.g.*, layer-3, layer-4), with multiple protocol options (*e.g.*, TCP, UDP) at each layer. And NF developers might try to cover a large protocol space in the NF code for completeness [29]. However, in the runtime deployment, NF operators might only configure a subspace of the entire protocol space due to the requirements (*e.g.*, cloud tenant filtering away some traffic [17]). If the incoming packets exercise extra protocols in the NF code than the configuration, the redundant processing will happen.

To eliminate such redundancy and to improve NF's performance, we prototype a tool named NFReducer, which follows three steps. First, NFReducer identifies the *packet processing logic* using program slicing so that the unrelated logic (*e.g.*, logging) could be excluded. Second, within an individual NF, NFReducer injects the NF configurations into its variables and applies constant propagation, constant folding, and dead code elimination to remove the redundant logic. In this step, we overcome a challenge where an NF may have infeasible paths which disable the identification of some redundant logic. We apply symbolic execution [16, 19, 34] to filter out the infeasible paths. Finally, when multiple NFs are chained, we consolidate them and leverage NFReducer to eliminate the redundant logic. In this case, the cross-NF redundancy (*e.g.*, late drop, duplicated parsing [15, 21, 23]) can be eliminated. In the cross-NF optimization, if the NF execution is a run-to-completion mode [20, 32], the optimized NF can be deployed directly; otherwise (pipeline mode) [42], the optimized NF needs to be decomposed back to multiple NFs.

We do not regard that NF developers should be blamed for the redundant logic. Indeed, all features and functionalities in an NF program come from a combination of various factors, such as historical evolvement, development tools, and market requirements. Supporting rich features helps an NF to take over a larger market share. In practice, when the development and operation of NF (*i.e.*, DevOps) are jointly considered (*e.g.*, NF infrastructure vendors like cloud providers and enterprise network constructors delivering solutions), a tool like NFReducer would be useful to improve the network performance, and it should be applied between the phases of development and deployment.

We implement NFReducer using LLVM [25], a widely used compiler infrastructure. We use two IDSes (Snort [10] and Suricata [11])

Acknowledgement: This project is supported by National Natural Science Foundation of China Grant No. 61802225 and the Zhongguancun Haihua Institute for Frontier Information Technology. Wenfei Wu is the corresponding author.

SOSR '20, March 3, 2020, San Jose, CA, USA

ACM ISBN 978-1-4503-7101-8/20/03…$15.00
https://doi.org/10.1145/3373360.3380832

```
1  /* One example Snort rule:
2  drop tcp 10.0.0.0/24 any -> 10.1.0.0/24 any
3  */
4  struct {
5     unsigned long sip, dip;
6     unsigned short sport, dport;
7     ...
8  } net;
9  void main() {
10    LoadRules();
11    while(1) {
12       pkt = ... // get a packet
13       DecodeEthPkt(pkt); // decode a packet
14       ApplyRules(); // match rules
15 }  }
16 void DecodeEthPkt(u_char *pkt) {
17    DecodeIPPkt(pkt);
18 }
19 void DecodeIPPkt(u_char *pkt) {
20    net.dip = ...
21    net.sip = ...
22    net.protocol = ...
23    log(net.sip, net.dip, net.protocol);
24    if (net.protocol == TCP)
25       DecodeTCPPkt(pkt);
26    else if(net.protocol == UDP)
27       DecodeUDPPkt(pkt);
28    else if(...) { ... }
29 }
30 void DecodeTCPPkt(u_char *pkt) {
31    net.dport = ...
32    net.sport = ...
33    log(net.sport, net.dport);
34 }
35 void DecodeUDPPkt(u_char *pkt) {
36    net.dport = ...
37    net.sport = ...
38    log(net.sport, net.dport);
39 }
40 void ApplyRules() {
41    while(...) { //iterate each rule r
42       if(MatchRule(r)) {
43          Action();
44          return;
45 }  }  }
46 int MatchRule(Rule *r){
47    if(r->sip != net.sip) return 0;
48    if(r->dip != net.dip) return 0;
49    if(r->protocol != net.protocol) return 0;
50    if(r->sport != net.sport) return 0;
51    if(r->dport != net.dport) return 0;
52    return 1;
53 }
```

Figure 1: Snort code (simplified)

to evaluate NFReducer. Our evaluation results show that NFReducer can significantly improve the packet processing rate of the two IDSes. In total, we make the following three contributions.

- We show the existence of the redundant logic in NF programs and categorize the redundant logic.
- We propose to apply classic compiler techniques to eliminate the redundant logic and prototype our idea by building NFReducer.
- Our experiments validate the existence of the redundant logic and the performance benefits after eliminating it.

2 BACKGROUND

This section uses Snort [10] as an example to illustrate the three types of redundant logic and discusses the related compiler techniques used by NFReducer for eliminating the redundancy.

2.1 A Motivating Example

Figure 1 shows the simplified code of Snort. After started, Snort loads the configured rules (line 10) and executes the packet processing loop (lines 11–15). The loop handles an incoming packet in each iteration through two steps. First, it decodes the packet (line 13) and saves the extracted header information in the global structure `net`. Second, the loop conducts rule matching (line 14) by comparing the header information (saved in `net`) with each configured rule (lines 41–45). If a rule is matched (*i.e.*, function `MatchRule()` returning 1), the corresponding action is taken (line 43), and all the following rules are skipped (line 44).

From this example, we observe that whether a computation (or parsing) result is used or has effects later may depend on runtime configurations. If a computation result is not used, its corresponding computation operations become redundant logic. In total, we identify three types of redundant logic in Figure 1. The first two types are within one single NF, and the last one is across multiple NFs.

First, all NF logic designed for a protocol layer may be redundant (*type-I*). For example, if all rules use "any" to match arbitrary port numbers (*e.g.*, line 2), no matter what port number an incoming packet has, its port number always match all configured rules, and the conditions at line 50 and line 51 are always false (*i.e.*, matched). Therefore, the port number decoding (*e.g.*, lines 31–32, lines 36–37) is redundant and can be eliminated to save CPU cycles.

Second, the logic for a particular protocol option may be redundant (*type-II*). For example, if the Snort code in Figure 1 is configured to only process TCP packets, all UDP packets would be ignored — the condition at line 49 would always be true for them (*i.e.*, not matched). The port number of incoming UDP packets would not influence rule matching results since line 50 and line 51 are never executed for the UDP packets. Therefore, the computation to decode port numbers for UDP packets inside function `DecodeUDPPkt()` is unnecessary and can be removed.

Third, the computation conducted by a previous NF may become redundant due to the configured rules of another NF deployed at a later stage (*type-III*). For example, if an active flow monitor deployed before a Snort instance who blocks UDP packets, all the parsing and counting for UDP packets in the monitor is redundant. If multiple NFs form a chain, consolidating all on-path NFs [15, 21, 33] can transform the *Type-III* redundancy into the type (*type-I* and *type-II*) within one single NF.

In this paper, we build a tool named NFReducer to *identify and eliminate redundant logic in NF programs*. We propose to use program instructions as the granularity instead of modules [15, 23, 33]. There are two reasons. First, not all commodity NFs are built by chaining modules, and module-level redundancy elimination techniques cannot cover all NFs (*e.g.*, Figure 1). Second, instruction-level analysis has the potential to conduct fine-grained elimination and achieve better performance.

2.2 Compiler Techniques

Program Analysis Techniques. We leverage several classic compiler techniques in NFReducer. We briefly describe them below.

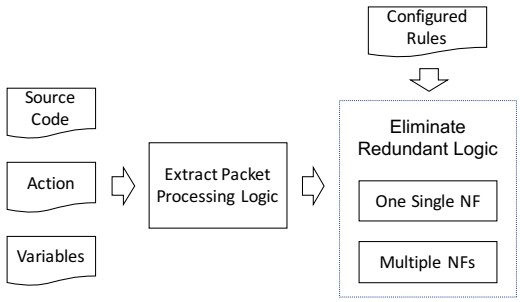

Figure 2: The Architecture of NFReducer

Constant Propagation. If a variable's value is assigned (or defined) by an instruction using a constant value, then between the assignment and the next re-assignment on the execution flow, the variable's usage can be replaced by the constant.
Constant Folding. If an instruction's result can be computed at compile time, all usages of the result can be replaced with the computed value during compilation.
Dead Code Elimination. If the execution of an instruction does not influence anything in the following execution flow (*e.g.*, its computed value never used), the instruction can be eliminated.
Program Slicing. Starting from an instruction and one of its operands (*i.e.*, a criterion <instruction, variable>), program slicing can find all successor instructions with computation influenced by the criterion (*i.e.*, forward slicing), and all predecessor instructions whose computation influences the criterion (*i.e.*, backward slicing) [38].
Symbolic Execution. After marking inputs as symbolic variables, symbolic execution collects execution condition for each program path and then generates concrete inputs that can lead the path to be executed or validates the path is infeasible [16, 19, 34].
Compiler Infrastructure. LLVM is a compiler infrastructure widely used in programming languages, software engineering, and systems communities to build different techniques [25]. All the previously mentioned program analysis techniques have existing implementations under LLVM. Thus, we also build NFReducer using LLVM to fast prototype our idea.

3 NFREDUCER DESIGN

An overview of NFReducer's architecture is shown in Figure 2. NFReducer takes the source code of NFs and the user-labeled NF actions and critical variables as its input. It first extracts the packet processing logic (§3.2) and then refers to configurations to eliminate the redundancy, including that within one single NF (§3.3) and that across multiple NFs (§3.4).

3.1 NFReducer Input Labeling

The current version of NFReducer takes user-labeled critical variables and NF actions as its inputs. In the future, we will enhance NFReducer by building static analysis routines to identify the labeled information automatically.
Critical Variables. Intuitively, an NF program relies on a loop structure to process packets as a stream. We name the loop structure as the *packet processing loop* (*e.g.*, lines 11–15 in Figure 1). Inspired by the previous work [24], we divide an NF program's variables that influence how packets are handled into three categories, packet variables, state variables, and config variables, based on how the variables are used in the packet processing loop. Packet variables and state variables are important to identify the NF actions, while config variables are the key to eliminate the redundancy.

An NF's packet processing loop uses a received packet to overwrite a *packet variable* (*e.g.*, `pkt` at line 12 in Figure 1) at the beginning of each iteration and refers to the variable's value to conduct the remaining computation in the same iteration. NFReducer users can identify packet variables of an NF by searching network I/O functions and labeling their return values or referenced arguments.

Stateful NFs leverage *state variables* to maintain information across packets. How a packet is processed depends on the current values of state variables [22, 24]. State variables can be labeled by examining each variable used in the packet processing loop to check whether the variable satisfies the following conditions. First, it is not a local variable of the packet processing loop. Second, its value is modified in the packet processing loop. Third, its value influences how to handle incoming packets.

In the runtime, before an NF handles packets, configurations are loaded into *config variables* (*e.g.*, all fields of parameter `r` of function `MatchRule()` in Figure 1). A config variable has the following features. Its value is generated during parsing configuration files (or command line), it is used in the packet processing loop, and its value is not changed in the loop.
NF Actions. After receiving a packet, an NF can take external actions by replying or forwarding packets. It can also take internal actions by updating its state variables. NFReducer users can identify the external actions by searching the network I/O functions used to send packets (*e.g.*, function `Action()` at line 43 in Figure 1) and localize the internal actions by inspecting where the state variables are updated.

3.2 Packet Processing Logic Identification

Before eliminating the redundant logic, we first extract the packet processing logic from each analyzed NF for two reasons. First, removing functionalities unrelated to packet processing can improve an NF's performance. Second, some compiler techniques applied later (*e.g.*, symbolic execution) have scalability issues on the whole NF program, and only focusing on packet processing logic can enable NFReducer on large NFs.

Algorithm 1 shows the details of extracting the packet processing logic. The algorithm takes labeled NF actions as inputs, applies backward slices to search instructions whose execution can influence the execution of the labeled actions, and reports all searched instructions as identified packet processing logic.

We recommend having the developers or operators involved to decide whether to eliminate this kind of unrelated program logic. The reason is twofold. First, some features (*e.g.*, logging at line 33 in Figure 1) are key to debugging or testing. Although they do not impact how incoming packets are processed, removing them can increase the difficulty of understanding an NF's behaviors. Second, some logic may affect the correctness of an analyzed NF. For example, synchronization operations (*e.g.*, locks) provide thread safety if an NF is configured to run with multiple threads, but they are useless if there is only one thread.

3.3 Individual NF Optimization

We design an algorithm to optimize away redundant logic within a single NF. The algorithm takes an NF's packet processing logic

Algorithm 1 Packet Processing Logic (PktProcLogic())

Require: NF Action Set acS and NF Source Code S
Ensure: Packet Processing Logic S'
1: **function** PktProcLogic(acS, S)
2: initialize code set $CS = \{\}$
3: **for** each instruction in acS **do**
4: $CS = CS \cup$ ProgramSlicing($instruction, S$)
5: **end for**
6: $S' \leftarrow$ mergecode(CS)
7: **return** S'
8: **end function**

and configured rules as its input, eliminates *Type-I* and *Type-II* redundancy, and outputs the optimized processing logic (Algorithm 2).

As the first step, the algorithm conducts constant folding and constant propagation to replace config variables with the constant values specified in the configured rules (line 2 in Algorithm 2). Sometimes, an NF uses a loop to compare an incoming packet with its configured rules (*e.g.*, line 41–45 in Figure 1). In this case, the algorithm unrolls the loop by cloning its loop body and functions called from the loop *n* times, with *n* equal to the number of the configured rules, and applies constant folding and propagation to each cloned version by referring to the corresponding rule. For example, if the Snort in Figure 1 is only configured with the rule at line 2, the rule matching loop (lines 41–45) is changed to one clone of the loop body. Function `MatchRule()` called in the loop is also cloned. In the cloned version of `MatchRule()`, all config variables are changed to values in the rule firstly (*e.g.*, `r->protocol` at line 49 changed to `TCP`), and then constant folding is further conducted (*e.g.*, the condition at line 50 changed to `false`).

After this step, the *type-I* redundancy can be eliminated by applying dead code elimination. Take the Snort in Figure 1 as an example. After replacing the two conditions at lines 50 and 51 with `false`, there is no other place using port numbers. Thus, all the computation for port number decoding (*e.g.*, lines 30–31, lines 36–37) is dead, and the classic dead code elimination can identify the computation.

However, *type-II* redundancy cannot be removed by dead code elimination right now. For example, if the Snort in Figure 1 is only configured to match TCP packets with concrete port numbers, the two conditions at lines 50 and 51 cannot be changed to `false`. Thus, path-insensitive dead code elimination cannot eliminate any computation about decoding port numbers. However, the decoding in function `DecodeUDPPkt()` actually has no side effect and should be eliminated, since if an incoming packet is in UDP, the condition at line 49 is `true`, and the function `MatchRule()` returns before the port numbers are used at lines 50 and 51. Actually, after changing `r->protocol` to `TCP`, line 26 → line 27 → line 36 → line 49 → line 50 is an infeasible path, since if line 50 is executed, then the incoming packet is in TCP, and lines 26, 27 and 36 cannot be executed.

We leverage symbolic execution to filter out infeasible paths effectively. We extract all possible execution paths (line 4 in Algorithm 2). For each path, we conduct constant folding and propagation (line 6), and then we leverage symbolic execution to judge whether the path is feasible (lines 7–9). If it is feasible, we conduct dead code elimination (line 10). We compute the union of all code left in each feasible path and use the union as the optimized result.

Algorithm 2 Individual NF Optimization (NFOpt())

Require: Packet Processing Logic S, Configuration $conf$
Ensure: Optimized NF Code S'
1: **function** NFOpt($config, S$)
2: $S =$ apply $config$ to S.
3: initialize code set $CS = \{\}$
4: $paths =$ ExtractExecutionPath(S)
5: **for** each path p in $paths$ **do**
6: $p' \leftarrow$const_fold_propagate(p)
7: **if** SymbolicExecution(p') is infeasible **then**
8: **continue**
9: **end if**
10: $code \leftarrow$dead_code_elimination(p')
11: $CS = CS \cup code$
12: **end for**
13: $S' \leftarrow$ mergecode(CS)
14: **return** S'
15: **end function**

3.4 Cross-NF Optimization

As we discussed in §2.1, we can consolidate multiple NFs into one single NF to convert the redundancy across multiple NFs (*type-III*) into that within one single NF (*type-I* and *type-II*), so that we can apply the algorithm in §3.3.

Specially, assuming multiple NFs are chained together, NFReducer first extracts the packet processing logic between the packet receiving operation and the sending operation for each NF. NFReducer then merges the extracted logic into one program (NF'). Variables and functions in the extracted logic need to be renamed to avoid conflicts. In the end, NFReducer applies the algorithm in §3.3 to generate the optimized version of NF'.

How to deploy the optimized version depends on the execution model of the original NFs. If a run-to-completion model (*i.e.*, multiple NFs running as one single process) is used, the optimized version of NF' can be deployed directly as a single process. However, if the original NFs are deployed in a pipeline model (*i.e.*, multiple NFs running as independent processes and chained by inter-procedure I/O), NFReducer decomposes the optimized version of NF' into individual NFs to better utilize system resources (*e.g.*, multi-core processors, distributed systems). To decompose the optimized version of NF', NFReducer basically creates an optimized version for each individual NF. Given an NF (`NFi`), NFReducer first computes the intersection between the optimized version of NF' and `NFi`'s packet processing logic, and then use the intersection to replace the original packet processing logic in `NFi` to generate an optimized version for `NFi`. Operators can deploy the optimized version in the same way as the original instance of `NFi`.

4 IMPLEMENTATION

We rely on several existing implementations of program analysis to build NFReducer. We use the DG library [3] of Symbiotic [12] as the program slicing discussed in §3.2. The slicing takes the tuples of (instruction, instruction operand) as input. We use all the combinations between the user-labeled NF actions and their operands as input tuples.

We implement a pass to clone the code when it is necessary and replace the user-labeled config variables with corresponding constant values in configured rules (§3.3). The existing implementation of constant propagation only processes variables in primitive types and does not consider structs. We enhance the existing implementation to enable constant propagation on struct fields when they are used as config variables.

We use KLEE [16] as the symbolic execution engine. We configure KLEE to only analyze NF code, without inspecting functions in standard libraries.

5 EVALUATION

5.1 Experimental Setup

We implement NFReducer using LLVM-5.0.0. All our experiments are conducted on a Linux workstation, with ten 1200MHz CPU cores and 128GB memory.

Benchmarks. We select two IDSes (Snort-1.0 and Suricata-3.1) as our benchmark programs. We choose them because their implementations cover a large protocol space, and it is easy to change their configured rules. We use throughput (*i.e.*, the number of processed packets per second) as the performance metric.

Research Questions. Our experiments aim to validate whether the redundant logic exists in read-world NFs and whether NFReducer can effectively eliminate the redundancy. Since whenever an NF's configured rules are changed, operators need to apply NFReducer to analyze the NF again. We want to measure how much time the application of NFReducer costs to understand whether NFReducer can be used in a production environment. Specifically, our experiments are designed to answer the following two research questions.

RQ1: what the performance improvement we can achieve after applying NFReducer?

RQ2: how much operation overhead NFReducer incurs?

5.2 Experimental Results

We discuss how our experimental results answer the previous two research questions as follows.

RQ1. Effectiveness. We measure how much performance improvement we can gain on Snort and Suricata after eliminating logic irrelevant to packet processing, *type-I* redundancy, *type-II* redundancy, and *type-III* redundancy, respectively.

Packet Processing Logic. Figure 3 shows the performance gain after eliminating program logic unrelated to packet processing for Snort (§3.2). We can achieve around 10× performance improvement (*i.e.*, from 0.56 Mpps to 5.75 Mpps). Among the unrelated logic, logging each packet's statistics is the most time-consuming. We do not recommend removing all logging functionalities in NFs since they are important for testing and debugging. However, our results still confirm that NFReducer can precisely identify logic unrelated to packet processing, and removing the logic sometimes can significantly improve NFs' performance. We also conduct the same experiment for Suricata, but the performance gain is very small (*i.e.*, less than 1%).

As we discussed in §3.3, our redundancy elimination is applied to packet processing logic so that all the following evaluations and comparisons are conducted on the extracted packet processing logic for Snort and Suricata.

Type-I Redundancy. Figure 4 and Figure 5 show the performance improvement after eliminating *type-I* redundancy for Snort and Suricata, respectively. The two NFs are only configured with layer-3 rules. Snort only has a single-thread mode. Suricata has both single-thread and multi-thread modes.

Two figures show that packet processing rates of the two IDSes increase significantly. As packet size increases, the performance gain is constant for Snort, but more performance gain can be achieved for Suricata in the single-thread mode. The reason is that Suricata inspects packets deeper in payload than Snort so that Suricata conducts more redundant computation after configured with layer-3 rules only. For Suricata in the multi-thread mode, the performance gain increases firstly and then decreases. We will identify the reason in the future. In summary, *type-I* redundancy exists in both Snort and Suricata, and we can achieve a much better performance after eliminating it.

Type-II Redundancy. Figure 6 and Figure 7 show the performance gain after eliminating *type-II* redundancy for Snort and Suricata in the single-thread mode. The two IDSes are configured with TCP rules only. The results are straightforward — as the number of UDP packets increases, the algorithm shows a larger performance gain. When the proportion of UDP packets increases to 50%, removing the redundancy can achieve 40% and 2.5× performance gain for Snort and Suricata, respectively. These results show that *type-II* redundancy can significantly impact NFs' performance, and NFReducer can effectively eliminate it.

Type-III Redundancy. To evaluate the performance gain after eliminating *type-III* redundancy, we add a monitor before a Snort instance and configure the Snort with TCP rules only. We compare the throughput under two settings. First, the monitor and the Snort execute in two different processes, and they are chained in a pipeline ("Mon–Snort" in Figure 8). Second, the monitor and the Snort are consolidated together and optimized by NFReducer ("Mon+Snort" in Figure 8). As shown in Figure 8, the consolidation and redundancy elimination can help improve throughout by more than 30%, and increasing the proportion of UDP packets can increase the performance gain. Thus, *type-III* redundancy also exists when multiple NFs are deployed together.

RQ2. Overhead. The overhead of NFReducer comes from two aspects. First, we need users to label the critical variables and NF actions. Second, when the configured rules of an NF are changed, NFReducer needs to analyze the NF and rebuild the NF.

In total, Snort has 2 packet variables, 8 state variables, and 14 config variables. The numbers of the three variable types for Suricata are 2, 4, and 19, respectively. One author of this paper identifies all the variables in around one hour. An NF program only needs this manual labeling once. When the NF's configuration changes, the labeling results can be reused by NFReducer. To sum up, we don't think the labeling process can incur a large operation overhead.

For Snort, the execution time to extract the packet processing logic is 7.6s, and the execution time to eliminate redundancy in one Snort instance is 26.8s. The optimized version needs 0.126s to be built into an executable. NFReducer spends 1.2s and 83.6s to extract the packet processing logic and remove redundancy for Suricata,

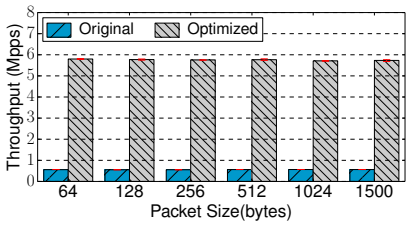

Figure 3: Throughput of Snort after removing irrelevant.

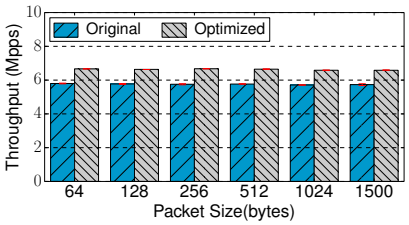

Figure 4: Throughput of Snort after eliminating *type-I* redundancy.

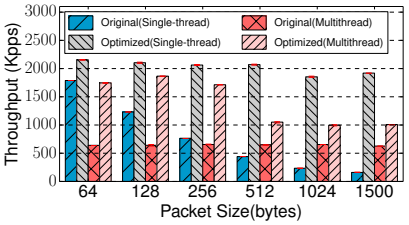

Figure 5: Throughput of Suricata after eliminating *type-I* redundancy.

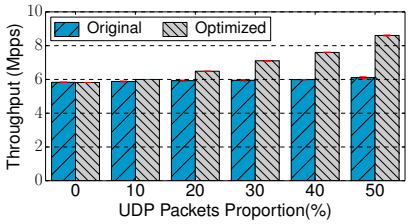

Figure 6: Throughput of Snort after eliminating *type-II* redundancy.

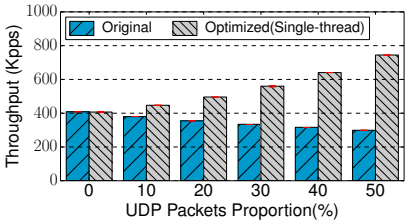

Figure 7: Throughput of Suricata after eliminating *type-II* redundancy.

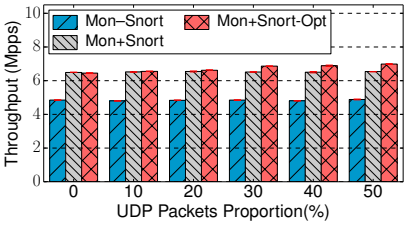

Figure 8: Throughput of Snort after eliminating *type-III* redundancy.

respectively. To build the optimized version, we need 2.753s. Since an NF's configured rules tend to be used for a long time (*e.g.*, several days), we think the execution time of NFReducer and the rebuilding time are tolerable in real-world operation.

6 SCOPE OF USAGE

While we use two IDSes as examples to show the usage of NFReducer, we discuss the scope of NFReducer— what categories of NFs can benefit from NFReducer. As discussed in §1, the redundancy is caused by the runtime configuration exercising only a subspace of network protocols in the NF code. Thus, NFReducer could improve NFs that process a larger protocol space significantly, *e.g.*, IDSes, firewalls, and Deep Packet Inspectors (DPI). NFs that process a single protocol could benefit less from NFReducer, *e.g.*, TCP load balancer, HTTP cache.

Even if NFReducer cannot benefit all NF categories, such a tool is non-trivial in many DevOps scenarios. (1) Each NF category contains many different NF variants (*i.e.*, products and implementations). For example, both Snort [10] and Suricata [11] are IDSes, and both PAN [7] and pfSense [8] are firewalls. NFReducer will improve their runtime performance. (2) Many NFs synthesize several functionalities (*e.g.*, PAN and pfSense [7, 8] with NAT and firewall), and use runtime configurations to turn them on/off. NFReducer could be applied to them. (3) In a network, one NF would be deployed as many instances, each of which has customized configurations. For example, a public cloud may have each physical server installed its own security rules (*e.g.*, iptables [6] to filter traffic), and the rules are customized by the tenant of each server. Using NFReducer to automate the optimization of each instance is more applicable than conducting manual optimization.

7 RELATED WORK

NFReducer provides an approach to jointly considering NF development and operation for better performance. In the current NF development, NFs are developed either as individual legacy software (e.g., load balancers, firewalls, NATs, caches, etc. [1, 2, 5, 8–11]) or in a development framework (e.g., libVNF [31], NetBricks [32]). In the NF deployment, NFs are managed by control plane systems such as OpenNetVM [42], ONOS [14], BESS [20], ClickOS [28], OpenNF [18], libVNF [31], NEWS [30], OpenBox [15], and NetBricks [32]. And NFReducer can improve the NF performance in these development and deployment frameworks.

Existing works on NF performance acceleration fall into two categories. They either accelerate the processing speed (*e.g.*, using FPGA or GPU [4, 26, 36, 37, 40, 41]) or parallel the processing [35, 43]. NFReducer is orthogonal to these solutions. It refines the NF internal algorithm and reduces complexity. Microboxes [27] is a framework that takes redundancy as inputs and eliminates redundancy, while NFReducer can also identify redundancy.

Works like SNF [23], CoMB [33], OpenBox [15], and SpeedyBox [21] also propose the idea of cross-NF redundant logic elimination. NFReducer is different from them by converting redundancy across multiple NFs into redundancy within one NF. Works like StatelessNF, StateAlyzr, and NFactor [13, 22, 24, 39] inspire the packet processing logic identification in NFReducer.

8 CONCLUSION AND FUTURE WORK

We discover the problem that existing legacy NFs can be over-engineered with runtime redundant logic. We propose a framework named NFReducer to eliminate the redundancy in individual NFs and cross NFs. NFReducer is built on program analysis techniques to eliminate such redundancy. And the preliminary implementation and evaluation show that NFReducer can improve the performance of the two example NFs significantly.

We will enhance NFReducer from the following aspects. First, we will complete and automate the whole workflow process. Second, we will apply NFReducer to more NFs (for legacy individual NFs or NFs in a development framework). Third, we will make complete tests on NFReducer, including throughput, latency, optimization overhead, etc.

REFERENCES

[1] Balance. https://www.inlab.de/balance.html.
[2] clicknat. https://github.com/kohler/click/blob/master/conf/thomer-nat.click.

[3] DG Static Slicer. https://github.com/mchalupa/dg.
[4] DPDK. https://www.dpdk.org.
[5] Haproxy. http://www.haproxy.org.
[6] iptables. https://www.netfilter.org/projects/iptables/index.html.
[7] PAN. https://www.paloaltonetworks.com/.
[8] pfsense. https://www.pfsense.org/.
[9] PRADS. https://github.com/gamelinux/prads.
[10] Snort IDS. https://www.snort.org/.
[11] Suricata IDS/IPS. https://suricata-ids.org/.
[12] Symbiotic. http://staticafi.github.io/symbiotic.
[13] Arashloo, M. T., Koral, Y., Greenberg, M., Rexford, J., and Walker, D. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)* (Florianópolis, Brazil, 2016).
[14] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking (HotSDN '14)* (Chicago, Illinois, 2014).
[15] Bremler-Barr, A., Harchol, Y., and Hay, D. Openbox: a software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)* (Florianópolis, Brazil, 2016).
[16] Cadar, C., Dunbar, D., and Engler, D. 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 (OSDI '08)* (San Diego, California, 2008).
[17] Devarajan, S., Stepanenko, V., Verma, R., and Kawamoto, J. Multi-tenant cloud-based firewall systems and methods, May 18 2017. US Patent App. 14/943,579.
[18] Gember-Jacobson, A., Viswanathan, R., Prakash, C., Grandl, R., Khalid, J., Das, S., and Akella, A. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)* (Chicago, Illinois, 2014).
[19] Godefroid, P., Klarlund, N., and Sen, K. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)* (Chicago, IL, USA, 2005).
[20] Han, S., Jang, K., Panda, A., Palkar, S., Han, D., and Ratnasamy, S. Softnic: A software nic to augment hardware. Tech. Rep. UCB/EECS-2015-155, EECS Department, University of California, Berkeley, 2015.
[21] Jiang, Y., Cui, Y., Wu, W., Xu, Z., Gu, J., Ramakrishnan, K. K., He, Y., and Qian, X. Speedybox: Low-latency nfv service chains with cross-nf runtime consolidation. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems (ICDCS '19)* (Dallas, Texas, 2019).
[22] Kablan, M., Alsudais, A., Keller, E., and Le, F. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17)* (BOSTON, MA, 2017).
[23] Katsikas, G. P., Enguehard, M., Kuźniar, M., Maguire Jr, G. Q., and Kostić, D. Snf: Synthesizing high performance nfv service chains. *PeerJ Computer Science 2* (2016), e98.
[24] Khalid, J., Gember-Jacobson, A., Michael, R., Abhashkumar, A., and Akella, A. Paving the way for {NFV}: Simplifying middlebox modifications using statealyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)* (Santa Clara, CA, 2016).
[25] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '14)* (Palo Alto, California, 2004).
[26] Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)* (Florianópolis, Brazil, 2016).
[27] Liu, G., Ren, Y., Yurchenko, M., Ramakrishnan, K., and Wood, T. Microboxes: high performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions. In *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM '18)* (Budapest, Hungary, 2018).
[28] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. Clickos and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)* (Seattle, MA, 2014).
[29] McCanne, S., and Jacobson, V. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter* (1993), vol. 46.
[30] Mekky, H., Hao, F., Mukherjee, S., Lakshman, T., and Zhang, Z.-L. Network function virtualization enablement within sdn data plane. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications* (Atlanta, GA, 2017).
[31] Naik, P., Kanase, A., Patel, T., and Vutukuru, M. libvnf: Building virtual network functions made easy. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '18)* (Carlsbad, CA, 2018).
[32] Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S., and Shenker, S. Netbricks: Taking the v out of nfv. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)* (Savannah, GA, 2016).
[33] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. Design and implementation of a consolidated middlebox architecture. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)* (San Jose, CA, 2012).
[34] Sen, K., Marinov, D., and Agha, G. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '05)* (Lisbon, Portugal, 2005), ESEC/FSE-13.
[35] Sun, C., Bi, J., Zheng, Z., Yu, H., and Hu, H. Nfp: Enabling network function parallelism in nfv. In *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM '17)* (Los Angeles, CA, 2017).
[36] Sun, W., and Ricci, R. Fast and flexible: parallel packet processing with gpus and click. In *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems (ANCS '13)* (San Jose, CA, 2013).
[37] Vasiliadis, G., Koromilas, L., Polychronakis, M., and Ioannidis, S. GASPP: A gpu-accelerated stateful packet processing framework. In *2014 USENIX Annual Technical Conference (USENIX ATC '14)* (Philadelphia, PA, 2014).
[38] Weiser, M. Program slicing. In *Proceedings of the 5th international conference on Software engineering (ICSE '81)* (1981).
[39] Wu, W., Zhang, Y., and Banerjee, S. Automatic synthesis of nf models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)* (Atlanta, Georgia, 2016).
[40] Yi, X., Duan, J., and Wu, C. Gpunfv: a gpu-accelerated nfv system. In *Proceedings of the First Asia-Pacific Workshop on Networking (APNet '17)* (Hong Kong, China, 2017).
[41] Zhang, K., He, B., Hu, J., Wang, Z., Hua, B., Meng, J., and Yang, L. G-net: Effective GPU sharing in NFV systems. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)* (Renton, WA, 2018).
[42] Zhang, W., Liu, G., Zhang, W., Shah, N., Lopreiato, P., Todeschi, G., Ramakrishnan, K., and Wood, T. Opennetvm: A platform for high performance network service chains. In *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMIddlebox '16)* (Florianopolis, Brazil, 2016).
[43] Zhang, Y., Anwer, B., Gopalakrishnan, V., Han, B., Reich, J., Shaikh, A., and Zhang, Z.-L. Parabox: Exploiting parallelism for virtual network functions in service chaining. In *Proceedings of the Symposium on SDN Research (SOSR '17)* (Santa Clara, CA, 2017).