NFReducer: Redundant Logic Elimination for Network Functions with Runtime Configurations

Bangwen Deng and Wenfei Wu
Tsinghua 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 individual NF and NF chains when they are deployed with concrete configured rules. We use program analysis techniques to optimize away the redundancy where we also overcome the NF specific challenges — we combine symbolic execution and dead code elimination to eliminate unused logic, we customize the common sub-expression elimination to eliminate duplicated logic, and we add network semantics to the dead code elimination to eliminate overwritten logic. We implement a prototype named NFReducer using LLVM. Our evaluation on both legacy and platform NFs shows that after eliminating the redundant logic, the packet processing rate of the NFs can be significantly improved and the operational overhead is small.**

I. Introduction

Virtualized network functions (NFs) in the network data plane would process all traversing network traffic. Thus, their efficiency significantly affects the whole network's end-to-end performance (*e.g.*, latency accumulation, throughput bottleneck). And a lot of works recognize this critical efficiency issue and propose the corresponding optimization, such as accelerating the NF execution [1]–[6], parallelizing NF (chains) [7], [8], and consolidating NFs [9]–[11].

A recent trend of DevOps inspires us to propose an orthogonal approach — *using the operation-time configurations to optimize NF programs*. "DevOps is a set of practices that combine software development and IT operations", whose original target is to provide agile development and deployment. With this trend, the barrier between the NF developers and operators vanishes, meaning that the same NF developers are able to get the operation-time configuration (either in advance or in the development-deployment-operation life cycle).

In NF runtime operation, there are two kinds of configurations — individual NF rules and NF chaining policy[1]. With these configurations, we identify that a part of the NF program code would become *redundancy logic*, which is defined as the piece of code whose execution does not influence the correctness of the packet processing of an NF or NF chain. We summarize three types of them — unused logic, duplicated logic, and overwritten logic.

Wenfei Wu is the corresponding author.

[1]In this paper, we consider the NFs are in a chain. They may be a complicated topology like a directed acyclic graph (DAG), then the method in this paper still works but needs to be combined with routing and switching logic.

We refer to a few compiler optimization techniques to optimize them, including dead code elimination, common sub-expression elimination, etc. We also face and overcome the network-specific challenges. (1) Unused logic can generally be eliminated by dead code elimination, but network protocols may share variables (e.g., TCP port and UDP port), which invalidates the dead code elimination. We overcome this challenge by applying symbolic execution to extract individual paths and run dead code elimination on each path (§IV-A). (2) Duplicated logic is from multiple NFs' parsing. It is supposed to be eliminated by common sub-expression elimination and copy propagation. We overcome an engineering challenge that the current tool only conducts the optimization on a single variable, not for global data structure; we flatten the data structure in NFs to apply the optimization (§IV-B). (3) Overwritten logic can be presumably eliminated by dead code elimination, but the method itself depends on judge whether a variable in an instruction is a lefthand symbol (LHS, whose value depends on other variables) or a righthand symbol (RHS, whose value decides other variables' values). In network software, `send(pkt)` and `drop(pkt)` have complicated cases to decide whether `pkt` is LHS or RHS. We add a heuristic before dead code elimination and avoid complicated judgment (§IV-C).

We build a tool chain named NFReducer with these three optimizations on LLVM. We measure the performance gain of applying each optimization and all to two commodity NFs, one platform NF, and a few NF chaining cases. We also analyze the manual labor to the code so as to apply NFReducer, and measure the execution time and re-compilation time as the overhead. The result shows that NFReducer can obviously improve NF's processing speed (from 7% to $2.5\times$ depending on the situation) and has limited overhead (one-time manual labeling and less than 10s rebuilt time).

Scope. NFReducer has the assumption that operation-time configurations can be fed to the development process. In DevOps, NF developers and operators are the same groups of people (e.g., NF infrastructure vendors like cloud providers and enterprise network constructors delivering solutions), and NFReducer should be applied between the phases of development and deployment. In the case where the developer does not have the operation-time configurations, we do not regard that they 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

```
 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 }
```

Fig. 1: Snort code (simplified)

features helps an NF to take over a larger market share. The contribution of this paper is as follows.

- We show the existence of three types of redundant logic in NFs in the scenario of DevOps, and devise compiler-based solutions to eliminate the redundant logic.
- We implement the methodologies as a tool chain named NFReducer. We evaluate it on commodity NFs and platform NFs and demonstrate the performance gain and acceptable overhead when applying NFReducer to these NFs.

II. Background

A. NFs and the Operations

NFs are software in the network data plane. In their operation, there are two kinds of configurations. Each NFs need to be configured to process the traffic with its own logic, and multiple NFs need to be chained for synthetic functionalities.

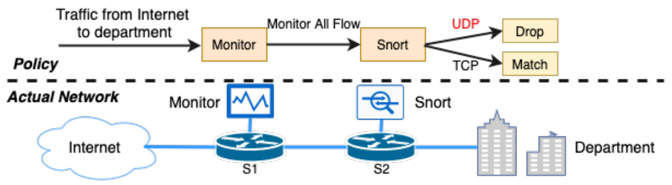

Fig. 2: NF chain: Monitor + Snort

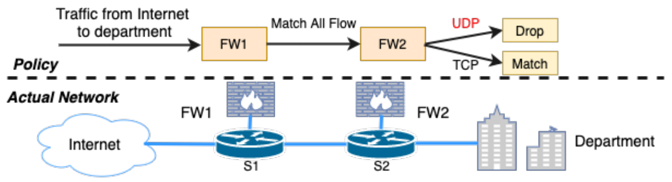

Fig. 3: NF instances: FW1 + FW2

Figure 1 shows an example of an Intrusion Detection System (IDS) named Snort [12]. The configuration is about which flow should get through (line 2). We observe that NF typically has two stages — packet parsing and NF specific logic[2] [13]. Snort parses a packet (line 13) and saves the extracted header information in the global structure `net`; it then iterates configured rules (line 14) to find the first match (line 41-45), and applies corresponding action (drop or pass, line 43).

Figure 2 and 3 show configurations about NF chains, where several NFs are placed in a sequence with traffic routed to traverse them in the same sequence; each NF in the chain are configured as above. Figure 2 shows a case where traversing traffic is monitored (i.e., counting flow statistics) and then processed by the IDS. Figure 3 shows a case where two firewall instances are chained together: this is possible if two NF operators (e.g., a company level and a department level) want to manage their rules independently or with priority.

B. Three Types of Redundant Logic

We find that both kinds of configurations would cause redundant logic.

Unused Logic. In Snort, if all rules use "any" to match arbitrary port numbers (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.

We call this a piece of *unused logic*. Note that "unused" means "executed but not used", not "non-executed". The essential reason is from 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 [14]. However, in the runtime deployment, NF operators might only configure a

[2]There exist NFs whose two stages are mixed, i.e., processing as parsing, but they are minority ones. We do not discuss them in this paper.

subspace of the entire protocol space due to the requirements (*e.g.*, cloud tenant filtering away some traffic [15]). If the incoming packets exercise extra protocols in the NF code than the configuration, the redundant processing will happen.

Duplicated Logic. In Figure 2 and Figure 3, both NFs need to parse packets and then perform their specific packet processing logic. The double "parsing" is redundant logic.

We call this a piece of *duplicated logic*. The essential reason is that the class of NF software usually share the same operation — parsing packets. When they are developed independently, each of them needs this piece of logic (to proceed to the next step); but in DevOps, an NF chain is viewed as one piece of software without isolation, and thus, the same logic of difference NFs become redundant.

Overwritten Logic. In Figure 3, in addition to the duplicated logic, the two firewalls may have other redundancy. If firewall 1 lets all UDP packets get through, but firewall 2 blocks all UDP packets, all work in firewall 1 that is done to UDP (e.g., statistics, TTL decremental, or modification) would become redundant.

We call this a piece of *overwirtten logic*. The essential reason is that the NF actions have semantic priority — packet drop can invalidate packet pass. Once NF's execution order (i.e., the order on the chain) differs from the semantic order, low semantic priority actions are first executed and cannot be eliminated.

C. Goal and Preliminary Compiler Techniques

Goal. Our goal in this paper is to find a solution that *identifies and eliminates redundant logic in NFs and NF chains caused by operation-time configurations*, and the solution has better be automated.

We take inspirations from the way how a compiler optimizes a program during compilation, and build our own solution NFReducer. In NFReducer, we also overcome a few specific challenges in NFs. We use a few preliminary program analysis methods and a compiler framework, and we introduce them below.

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. For example, `x=1+1` can be optimized as `x=2`.

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[3]), the instruction can be eliminated. For example, if a variable `x` is never used after `x=1`, then `x=1` itself can be eliminated.

Common Sub-expression Elimination. If instances of identical expressions(i.e., they are all equal to the same value) happens in the program, they can be replaced by a single variable holding the computed value. For example, `a=x+y; b=x+y;` can be optimized to `a=x+y; b=a;`

[3]"Being used" is formally described as "appearing on the right side of an assignment statement", i.e., being a right-hand symbol or RHS for short.

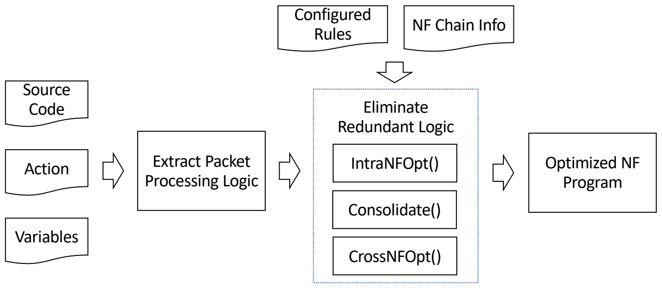

Fig. 4: The Architecture of NFReducer

Copy Propagation. If a local variable is acting as an intermediate variable, the occurrences of the variables can be replaced with their values. Copy propagation often runs afterward the optimization of common sub-expression elimination to achieve higher efficiency. For example, If `b` is not used after `b=a; n=b+c;`, then the sequence can be optimized as `n=a+c`.

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) [16].

Symbolic Execution. After marking inputs as symbolic variables, symbolic execution collects execution conditions for each program path and then generates concrete inputs that can lead the path to be executed or validate the path is infeasible [17]–[19].

Compiler Infrastructure. LLVM is a compiler infrastructure widely used in programming languages, software engineering, and systems communities to build different techniques [20]. All the previously mentioned program analysis techniques have existing implementations under LLVM. Thus, we also build NFReducer using LLVM to fast prototype our idea.

III. Synthetic Overall Workflow

Eliminating redundant logic is not as simple as run the compiler optimization once to the NF program. It has the following procedures: (1) the NF program should be preprocessed to extract the packet processing logic; (2) the operation-time configuration should be injected to the program; (3) then redundant logic elimination can be applied; (4) after elimination, the program needs to (re)deployed to the system; (5) once the configuration changes, the optimization needs to be redone from step (2) to step (4). Figure 4 shows the overall workflow of NFReducer.

A. Packet Processing Logic Extraction

The core functionality of an NF is to process the packet, but as a general piece of software, it may be developed with packet processing irrelevant features, e.g., logging, standard output warning. While they are useful in practice, in this paper,

Algorithm 1 Identify Critical Variables (VarsIdentify())

```
Input: NF Source Code $S$
Output: StateVariables $SV$, ConfigVariables $CV$
 1: function VARSIDENTIFY($S$)
 2:     initialize variables set $VS = \{\}$, $SV = \{\}$, $CV = \{\}$
 3:     $VS = VS \cup$ GlobalVarDecls($S$)
 4:     $VS = VS \cup$ StaticVarDecls($S$)
 5:     $VS = VS \cup$ LocalVarDecls($LoopProc$)
 6:     for each stmt in AssignmentStmts($loopProc$) do
 7:         for each $var$ in $VS$ do
 8:             if $var == stmt.LHS$
 9:            or $var$ in $PointsTo(stmt.LHS)$
10:            or $PointsTo(var) \cap PointsTo(stmt.LHS)$
11:             then
12:                 $SV = VS \cup var$
13:             end if
14:         end for
15:     end for
16:     $CV = VS - SV$
17:     for each $var$ in $CV$ do
18:         if $!Used(LoopProc)$ then
19:             $Remove$(CV,var)
20:         end if
21:     end for
22:     return $SV, CV$
23: end function
```

we exclude these packet processing irrelevant logic due to the following reasons.

First, eliminating the packet irrelevant logic can reduce the total program size, which helps to accelerate the program analysis; without this elimination, some program analysis methods (e.g., symbolic execution) have scalability issues. Second, the irrelevant logic consumes CPU cycles in the NF runtime, and eliminating it helps to evaluate the performance gain better. Third, in practice, these features usually can be tuned enabled/disabled by macros during the compilation time, and are often eliminated in the release version. We extract packet processing logic by first labeling its critical variable and then find the program slice related to these variables.

Step 1: Labeling variables. Intuitively, an NF program usually relies on a loop structure to process the packet stream. We name the loop structure as the packet processing loop (e.g., lines 11–15 in Figure 1). Packet processing loop may have four classes of variables involved — packet variables, state variables, config variables, and temp variables. Packet and state variables are important to identify the NF actions, while config variables are the key to eliminate the redundancy.

An NF usually relies on commonly used API to receive and send packets, e.g., *pcap_loop()* in libpcap and *rte_eth_rx_burst, rte_eth_tx_burst* in DPDK. At the beginning of the packet processing loop, the packet receiving function would assign the received value to a variable, which is the packet variable (e.g., `pkt` at line 12 in Figure 1). NFReducer users can search the network I/O function, and label the packet receiving function's return value or referenced arguments.

State variables maintain cross-packet information and affect the packet processing result [21], [22]. For example, a SYN flood detector needs a packet counter (and a threshold) to decide whether to allow the current packet; a TCP-connection aware firewall has a variable to record whether the firewall has seen an SYN, SYNACK, or ACK, and decides whether the current packet is valid. State variables have the following properties. First, it is not a local variable of the packet processing loop. Second, its value is modified in the packet processing loop (typical appear on the left side of an assignment, i.e., a left-hand symbol or LHS). Third, its value influences how to handle incoming packets (typical appear on the right side of an assignment, i.e., a right-hand symbol or RHS).

Config variables are initialized before the packet processing loop, and their values affect the packet processing result (e.g., all fields of parameter `r` of function `MatchRule()` in Figure 1). A config variable has the following properties. Its value is generated during parsing configuration files (or command line), it is used in the packet processing loop, and its value does not change in the loop.

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.

NFReducer uses Algorithm 1 to catch the state and config variables. It first searches all candidate variables whose lifetime is longer than the packet processing loop, including the global variables, static variables and local variables declared before the packet processing loop (Line 3-5). Then the algorithm refines the candidate variables according to the properties: for each assignment statement in the packet processing loop, it checks whether the left-hand symbol (LHS) is equal or point to the candidate variables, and put the candidate variables acting as LHS into the class of state variables (Line 6-15). The remaining set is a superset of all config variables (line 16). The remaining set is refined by removing all variables that are not used within the loop (not affecting packet processing, line 17-20), and the final remaining variables are config variables.

After receiving a packet, an NF can take *flow actions* by replying or forwarding packets. It can also take *state actions* by updating its state variables. NFReducer identifies the flow 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 state actions by inspecting where the state variables are updated (assignment statement).

Step 2: Extract packet processing program slice. Algorithm 2 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 inspect to decide whether to eliminate this kind of unrelated program

Algorithm 2 Packet Processing Logic (PktProcLogic())

```
Input: NF Action Set $acS$ and NF Source Code $S$
Output: 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
```

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.

B. Config Variable Propagation

With packet processing logic ready, the config variables are assigned by the operation-time configuration. Note that loops usually cause difficulty to program analysis because in a loop, a variable may stand for different instances in different iterations (e.g., the rule `r` in Snort in line 42). But for the convenience of redundant logic elimination, the code needs to transformed by unfolding loops.

Step 3: Unrolling loops. NFReducer unrolls a loop by cloning its loop body and functions called from the loop n times, with n equal to the number of entries of the loop data structure. For example, if the Snort in Figure 1 is only configured with the rule in 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.

After packet processing logic is prepared, the redundant logic elimination in § IV is applied to get an optimized piece of code. The redundant logic elimination is elaborated in § IV and we proceed to the (re)deployment after the elimination.

C. (Re)deployment Optimized NF

NF chains have two execution models — the run-to-completion model (RTC, i.e., multiple NFs consolidated and running as one single process) and the pipeline model (i.e.,multiple NFs running as independent processes and chained by inter-procedure I/O). As NFReducer consolidates NFs and eliminates duplicated parsing (not necessary to add them back), the optimized NF would run in an RTC model.

The RTC model has the advantage of no virtualization and inter-procedure I/O than the pipeline model. The traditional RTC [23], [24] model has the disadvantage of no inter-NF memory isolation, but this can be complemented by compilation time memory check. RTC and pipeline models have different parallelization methods — RTC would divide flow spaces and assign a core to an NF chain instance to process a subspace, and pipeline would assign each NF a separate core.

Algorithm 3 Individual NF Optimization (IntraNFOpt())

```
Input: Packet Processing Logic $S$, Configuration $conf$
Output: Optimized NF Code $S$'
 1: function INTRANFOPT($config, S$)
 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
```

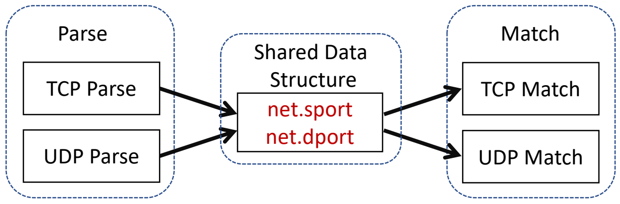

Fig. 5: Data Dependence in Snort

IV. Redundant Logic Elimination

A. Unused Logic Elimination

Dead code elimination naturally suits the unused logic elimination in NFs. In the example in §II, the port number is parsed in the packet parsing phase. But in the later stage, it is matched to a wildcard (always true), which can be eliminated by constant folding (line 50-51). Then the port number parsing becomes dead code and can be eliminated (line 30-31, 36-37). We call the *unused layer parsing* as a case of *unused logic*.

The main idea of dead code elimination is that if a variable is updated (LHS) but not used afterward (RHS), its assignment is a piece of dead code. In NF specific software, we meet with a new challenge — network packet data structure is shared between the execution path, causing the usage on one path to mislead the usage judgment on another path.

Using Snort as an example, both protocol TCP and UDP have the same semantics of "port number" and in NF TCP packets and UDP packets would share the same variable to store the port number. Assuming Snort 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

Algorithm 4 Consolidation (Consolidate())

```
Input: NF1 Source Code S1 and NF2 Source Code S2
Output: Consolidated Program S'
1: function CONSOLIDATE($S1, S2$)
2:     RENAMEVARFUNC($S1, S2$)
3:     $S' \leftarrow$ SPLICE($S1, S2$)
4:     $S' \leftarrow$ COMMON_SUBEXPRESSION_ELIMINATION($S'$)
5:     $S' \leftarrow$ COPY_PROPAGATION($S'$)
6:     $S' \leftarrow$ DEAD_CODE_ELIMINATION($S'$)
7:     return $S'$
8: end function
```

logic to decode port numbers for UDP packets inside function `DecodeUDPPkt()` is unnecessary and should be removed.

But in fact, the data structure "port number" is used in TCP matching, and the dead code elimination algorithm would falsely recognize it as being used after the UDP parsing. Thus, the UDP parsing would not be eliminated. Figure 5 illustrates the dependency between the two phases and the data structure, and we name it *unused protocol parsing*.

Solution. We leverage symbolic execution to search the paths sharing data structure. We perform dead code elimination on each path, and then merge each path back to a program.

The whole algorithm is illustrated in Algorithm 3. We extract all execution paths (line 4, they may contain unsatisfiable one). 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, do this after line 6 can be faster). 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.

B. Duplicated Logic Elimination

Duplicated logic is eliminated by splicing NFs as one program and remove repeated code. Common sub-expression elimination and copy propagation are commonly used methods to remove repeated code.In the program splicing, variables and functions must be renamed to avoid conflicts (so as to be compilable). Use the Snort as an example, if two Snort programs are spliced (assuming their variables and functions are renamed), a packet would be parsed twice and store in two `net` (assume they are renamed `net1` and `net2`). The latter one should be removed.

Algorithm 4 shows the workflow of consolidating two NFs. It does program renaming, program splicing, common sub-expression elimination, copy propagation, and dead code elimination sequential and completes the optimization.

In the implementation, we overcome an engineering challenge. The default version of LLVM CSE optimization can only handle single variables, not global data structures. But a packet is usually a data structure (e.g., `net` in Snort), which cannot be handled. The essential reason is a data structure field is represented by a base address plus an offset, which has no name and has the same format with a pointer plus an offset (may cause mistakes). We parse the code and mark the packet

Algorithm 5 Cross-NF Optimization (CrossNFOpt())

```
Input: Consolidated Program S
Output: Optimized Program S'
1: function CROSSNFOPT($S$)
2:     initialize code set $CS$ = {}
3:     $paths$ = EXTRACTEXECUTIONPATH(S)
4:     for each path $p$ in $paths$ do
5:         TRUNCATEATFIRSTDROP($p$)
6:         CHECKCHAINEDACTIONS($p$)
7:         LABELACTIONPKT($p$)
8:         $code \leftarrow$DEAD_CODE_ELIMINATION($p$)
9:         $CS = CS \cup code$
10:    end for
11:    $S' \leftarrow$ MERGECODE($CS$)
12:    return $S'$
13: end function
```

specific data structure fields (its base address plus the offset) as a single variable, and re-implement CSE.

C. Overwritten Logic Elimination

Intuitively, overwritten logic can be eliminated by dead code elimination. For example, for a sequence of instructions `x=1; x=2;`, the first instruction `x=1` is an assignment (LHS) that is never used (RHS) afterward (before `x=2`), and thus can be eliminated.

But in NF software, there is a difficulty in deciding whether the variable is RHS or LHS in NF actions. That is, should the variable `pkt` in `send(pkt)` and `drop(pkt)`[4] be regarded as RHS or LHS? For `drop(pkt)`, `pkt` should be LHS (or neither), so that all previous packet header modification is regarded as dead code. But all the following code should not participate in optimization.

For `send(pkt)`, `pkt` should be partially RHS. That is, `pkt` should be unrolled to packet header fields, for fields that have been LHS before, they are RHS now; for fields that have not been LHS, they are neither. This complicated decision can avoid the unused fields (e.g., port number) become RHS so that they cannot be eliminated.

In addition, this overwritten logic elimination should be done on each execution path because we do not want TCP drop to eliminate UDP parsing.

Algorithm 5 shows the whole process. The program is first symbolically executed to get each path. Then each path is truncated at the first drop, and packet fields are labeled according to the action. And NFReducer runs dead code elimination on each path. Finally, all optimized paths are merged, and the result is returned.

Overall, the three algorithms should be applied in the following order — duplicated logic elimination, then overwritten logic elimination, and finally unused logic elimination.

[4]Some papers say three actions — pass, drop, and modify. We regard the modify as "an assignment plus pass".

V. Implementation

In packet processing logic extraction, We rely on several existing implementations of program analysis tools in LLVM [20] to build NFReducer. We use the DG library [25] of Symbiotic [26] as the static program slicing discussed in §III-A. The slicing takes the tuples of (instruction, instruction operand) as input. We use all the combinations between the labeled NF actions and their operands as input tuples.

We implement a pass to clone the code when it is necessary and replace the identified config variables with corresponding constant values in configured rules (§IV-A). The existing implementation of constant propagation only processes variables in primitive types and does not consider structs and class. We enhance the existing implementation to enable constant propagation on struct and class fields when they are used as config variables. We also enhance the Common Sub-expression Elimination pass and Copy Propagation pass in the LLVM platform to finish the corresponding optimization.

We use KLEE [17] as the symbolic execution engine. By default, KLEE also explores commonly-used library functions along an execution path, which can significantly increase the number of paths to be analyzed without providing benefits to our results. Therefore, We configure KLEE to only analyze NF code, without inspecting functions in standard libraries.

We reuse most optimization pass of LLVM, including dead code elimination, constant propagation and folding. We implement common sub-expression elimination and copy propagation by ourselves as described in § IV-B.

VI. Evaluation

We collect NFs and apply NFReducer. We demonstrate that NFReducer can improve NF performance with acceptable operation overhead. We make a study about the potential gain of applying NFReducer in a campus network.

A. 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 legacy IDSes (Snort-1.0 and Suricata-3.1) and two platform NFs (a firewall and a monitor) on OpenNetVM [27] as our benchmark programs to perform the individual NF optimization. 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. We also set up the NF chain applications depicted in Section II in the OpenNetVM platform [27] as the cross-NF optimization benchmarks. In addition to the performance metric, we also measure the labeling effort and NFReducer's processing time as the metric of operation overhead.

Preparation: Packet Processing Logic. Figure 6 shows the performance gain after eliminating program logic irrelevant to packet processing for Snort (§III-A). 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 and OpenNetVM-Firewall, but the performance gain is very small (*i.e.*, less than 1%).

As we discussed in §III, 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 all the benchmarks.

B. Invidual NF Performance

Unused Layer Redundancy. Figure 7-9 show the performance improvement after eliminating unused layer redundancy for Snort, Suricata, and OpenNetVM-Firewall, respectively. The test NFs are only configured with layer-3 rules. Snort and OpenNetVM-Firewall only have single-thread modes. Suricata has both single-thread and multi-thread modes.

Two figures show that the throughput of the two IDSes and the firewall increase significantly. (*e.g.*, 15% for Snort, 21% for OpenNetVM-Firewall, 15%-10× and 40% to 3× for Suricata in single-thread mode and multi-thread mode, respectively).

As packet size increases, the performance gain is constant for Snort and OpenNetVM-Firewall, 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 the other two so that Suricata conducts more redundant computation after configured with layer-3 rules only. Suricata in the multi-thread mode shows the same trend except that in the small packet case (64B), the throughput is lower than that in 128B. Because multi-threading usually cannot handle well the small packet in high packet per second.

In summary, unused layer redundancy exists in both the legacy IDSes and the platform firewall, and we can achieve a much better performance after eliminating it.

Unused Protocol Redundancy. Figure 10-12 show the performance gain after eliminating unused protocol redundancy for Snort , Suricata in the single-thread mode, and OpenNetVM-Firewall, respectively. The benchmark NFs are configured with TCP rules only. 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. In OpenNetVM-Firewall, configurations are embedded in code, in which the compiler might apply some optimizations before we apply NFReducer, so the performance gain is moderated. But we can also achieve 6.8% performance gain for the firewall when the proportion of UDP packets reaches to 50%.

These results show that unused protocol redundancy can significantly impact NFs' performance, and NFReducer can effectively eliminate it.

C. NF Chain Optimization

To evaluate the performance gain after eliminating cross-NF redundancy, we set up the NF chains as mentioned in §II and

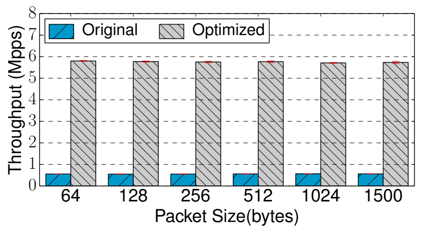

Fig. 6: Throughput of Snort after removing irrelevant.

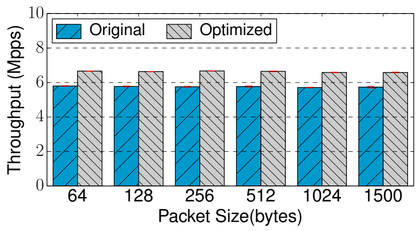

Fig. 7: Throughput of Snort after eliminating *unused layer* redundancy.

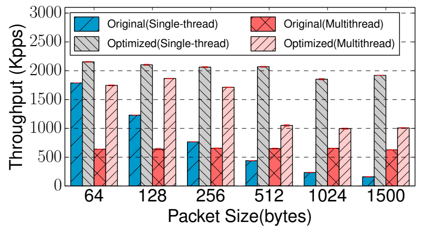

Fig. 8: Throughput of Suricata after eliminating *unused layer* redundancy.

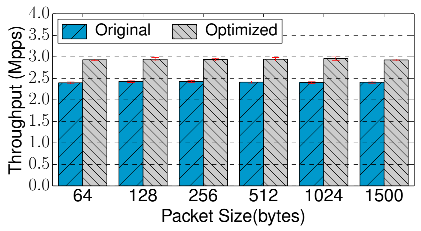

Fig. 9: Throughput of OpenNetVM-FW after eliminating *unused layer* redundancy.

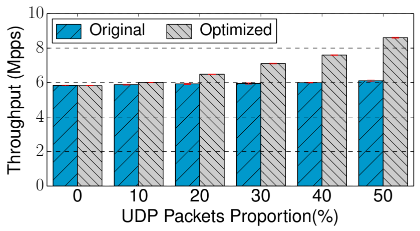

Fig. 10: Throughput of Snort after eliminating *unused protocol* redundancy.

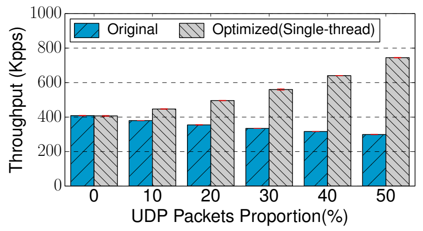

Fig. 11: Throughput of Suricata after eliminating *unused protocol* redundancy.

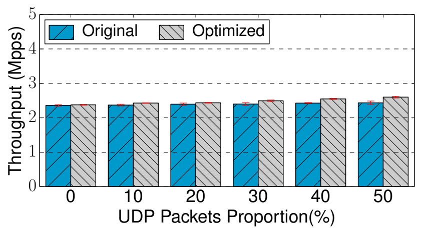

Fig. 12: Throughput of OpenNetVM-FW after eliminating *unused protocol* redundancy.

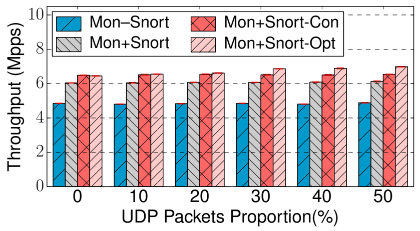

Fig. 13: Throughput of Snort after eliminating cross-NF redundancy.

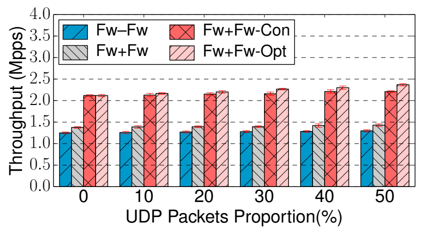

Fig. 14: Throughput of OpenNetVM-FW after eliminating cross-NF redundancy.

perform optimization on them. For the applications in §II, we configure the latter NF instance with TCP rules only.

We compare the throughput under two settings. First, the two NFs execute in two different processes, and they are chained in a pipeline ("Mon–Snort" in Figure 13, "Fw–Fw" in Figure 14). Second, the two NF instances are spliced directly ("Mon+Snort", "Fw+Fw"), consolidated together ("Mon+Snort-Con", "Fw+Fw-Con") and further optimized by NFReducer ("Mon+Snort-Opt", "Fw+Fw-Opt"). They are all deployed in one process. We measure the performance gain of cross-NF optimization from the following aspects.

Duplicated Logic Redundancy. As shown in Figure 13 and Figure 14, comparing the results of splicing and consolidating NFs, the consolidation can help improve throughout by more than 25% for monitor and Snort IDS chain, and improve throughput by nearly 55% for the two OpenNetVM-Firewall instances. Since the parsing overhead in OpenNetVM-Firewall contributes more to the whole overhead than that in Snort IDS, the performance gain of the firewall is larger.

This result shows that duplicated logic redundancy can impact NF chains' performance, and NFReducer can help eliminate this kind of redundancy effectively.

Overwritten Logic Redundancy. We compare the performance of the optimized program with consolidated one to explain the performance gain when eliminating overwritten logic redundancy. As shown in Figure 13 and Figure 14, the optimized program can achieve about 7% performance gain in both cases when UDP proportion reaches 50%. When increasing the proportion of UDP packets, we can also get the increasing performance gain in both cases.

Thus, overwritten logic redundancy also exists when multiple NFs are deployed together and NFReducer can help eliminate it and improve the performance of multiple NFs.

D. Overhead

The overhead of NFReducer comes from two aspects. First, we need 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.

Following the methods described in § III-A, we identify the critical variables for the benchmark NFs and show them in Table I. The three NFs have the same number of packet variables for sharing the similar network I/O functions. As for the state variables, Snort has 4 false positives related to the *log* operation among 8 detected state variables, while Suricata has one among 5. Due to direct rules declaration instead of using linked list in the source code, the OpenNetVM-Firewall is detected more config variables than the other two NFs. One

TABLE I: # of Identified Critical Variables in Benchmarks

	# of *Packet Variables*	# of *State Variables*	# of *Config Variables*
Snort IDS	2	8 (4 FPs)	6
Suricata IDS	2	5 (1 FP)	10
OpenNetVM-Firewall	2	4	22

TABLE II: Overhead

	Extracting packet processing logic	Optimization	Rebuilding
Snort IDS	7.6s	26.8s	0.126s
Suricata IDS	1.2s	83.6s	2.753s
OpenNetVM-Firewall	0.146s	1.606s	1.571s

author identifies all the variables following the methods in half an 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.

The optimizing time overhead is shown in Table II. 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, respectively. To build the optimized version, we need 2.753s. As for OpenNetVM-Firewall, NFReducer spends 0.146s to extract the packet processing logic and 1.606s to remove redundancy. We need 1.571s to rebuild the optimized version into the OpenNetVM platform.

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.

E. Study of Production Network

We collect and analyze the open-source Snort IDS rules [28] from the research community and the firewall rules from a campus cluster to show the practical rules deployment. Table III shows the statistics. In the campus network, the internal network rules target access control between regions, and the operator configures layer-3 rules. The external network rules target security, and the operator configures layer-4 rules. And they are on different firewall instances. The one with layer-3 rules is expected to benefit from NFReducer.

VII. Related Work and Discussion

NFV frameworks. 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. [12], [29]–[34]) or in a development framework (e.g., libVNF [35], NetBricks [23]). In the NF deployment, NFs are managed by control plane systems such as OpenNetVM [27], ONOS [36], BESS [24], ClickOS [37], OpenNF [38], libVNF [35], NEWS [39], OpenBox [10], and NetBricks [23]. And NFReducer can improve the NF performance in these development and deployment frameworks.

TABLE III: The Statistics of Rules

	Network Layer			Network Protocol			
	Layer-3	Layer-4	Others	TCP	UDP	IP/ICMP	Others
Rule Set of Snort	4.2%	95.8%	95.8%	87.1%	8.6%	4.2%	0
Rule Set of Campus Cluster	12.0%	88%	0	78.4%	9.6%	11.4%	0.6%

NF acceleration. Existing works on NF performance acceleration fall into two categories. They either accelerate the processing speed (*e.g.*, using FPGA or GPU [1]–[6]) or parallel the processing [7], [8]. NFReducer is orthogonal to these solutions. It refines the NF internal algorithm and reduces complexity. Microboxes [40] is a framework that takes redundancy as inputs and eliminates redundancy, while NFReducer can also identify redundancy.

Modular NF. Works like SNF [41], CoMB [11], OpenBox [10] also propose the idea of cross-NF redundant logic elimination. They work on NF composing modules (e.g., parser, filter). NFReducer works on the instruction level, and has more potential optimization space.

Other NF Consolidation. SpeedyBox [9] provide APIs to instrument NFs so as to collect and consolidate actions of an NF chain at runtime. NFReducer does the consolidation after development, avoiding changing the code. NFReducer [42] uses program analysis to eliminate intra-NF redundancy, and NFReducer extends the optimization space to NF chains.

Other Inspirations. Works like StatelessNF, StateAlyzr, and NFactor [21], [22], [43], [44] inspire the packet processing logic identification in NFReducer.

Scope of Usage. For individual NF optimization, as discussed in § II-B, the redundancy is caused by the mismatch of the configurations and the parsing 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. For NF chain optimization, all NFs benefit from the two optimizations. Because all NFs parse packets and have actions on packets.

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. For example, both Snort [12] and Suricata [30] are IDSes, and both PAN [45] and pfSense [31] are firewalls. NFReducer will improve their runtime performance. (2) Many NFs synthesize several functionalities (*e.g.*, PAN and pfSense [31], [45] with NAT and firewall) and 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 [46] to filter traffic) customized towards the server user. Using NFReducer to automate the optimization of each instance is more applicable than conducting manual optimization.

VIII. Conclusion

We built NFReducer that leverages program analysis techniques to eliminate redundancy logic in NFs and NF chains. By combining typical program analysis techniques, NF specific domain knowledge, and customized implementation, NFReducer can eliminate unused logic in individual NFs and duplicated and overwritten logic in NF chains. Our prototype and evaluation show that NFReducer can improve NF (chain) performance much with limited operational overhead.

References

[1] "DPDK," https://www.dpdk.org.

[2] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*, Florianópolis, Brazil, 2016.

[3] W. Sun and R. Ricci, "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.

[4] X. Yi, J. Duan, and C. Wu, "Gpunfv: a gpu-accelerated nfv system," in *Proceedings of the First Asia-Pacific Workshop on Networking (APNet '17)*, Hong Kong, China, 2017.

[5] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-net: Effective GPU sharing in NFV systems," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, Renton, WA, 2018.

[6] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, "GASPP: A gpu-accelerated stateful packet processing framework," in *2014 USENIX Annual Technical Conference (USENIX ATC '14)*, Philadelphia, PA, 2014.

[7] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM '17)*, Los Angeles, CA, 2017.

[8] Y. Zhang, B. Anwer, V. Gopalakrishnan, B. Han, J. Reich, A. Shaikh, and Z.-L. Zhang, "Parabox: Exploiting parallelism for virtual network functions in service chaining," in *Proceedings of the Symposium on SDN Research (SOSR '17)*, Santa Clara, CA, 2017.

[9] Y. Jiang, Y. Cui, W. Wu, Z. Xu, J. Gu, K. K. Ramakrishnan, Y. He, and X. Qian, "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.

[10] A. Bremler-Barr, Y. Harchol, and D. Hay, "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.

[11] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, 2012.

[12] "Snort IDS," https://www.snort.org/.

[13] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 699–718.

[14] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993.

[15] S. Devarajan, V. Stepanenko, R. Verma, and J. Kawamoto, "Multi-tenant cloud-based firewall systems and methods," May 18 2017, uS Patent App. 14/943,579.

[16] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering (ICSE '81)*, 1981.

[17] C. Cadar, D. Dunbar, and D. Engler, "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.

[18] P. Godefroid, N. Klarlund, and K. Sen, "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.

[19] K. Sen, D. Marinov, and G. Agha, "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)*, ser. ESEC/FSE-13, Lisbon, Portugal, 2005.

[20] C. Lattner and V. Adve, "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.

[21] 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)*, Santa Clara, CA, 2016.

[22] M. Kablan, A. Alsudais, E. Keller, and F. Le, "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] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, 2016.

[24] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, 2015.

[25] "DG Static Slicer," https://github.com/mchalupa/dg.

[26] "Symbiotic," http://staticafi.github.io/symbiotic.

[27] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. Ramakrishnan, and T. Wood, "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.

[28] "Snort IDS Rules," https://www.snort.org/downloads#rules.

[29] "PRADS," https://github.com/gamelinux/prads.

[30] "Suricata IDS/IPS," https://suricata-ids.org/.

[31] "pfsense," https://www.pfsense.org/.

[32] "Balance," https://www.inlab.de/balance.html.

[33] "Haproxy," http://www.haproxy.org.

[34] "clicknat," https://github.com/kohler/click/blob/master/conf/thomer-nat.click.

[35] P. Naik, A. Kanase, T. Patel, and M. Vutukuru, "libvnf: Building virtual network functions made easy," in *Proceedings of the ACM Symposium on Cloud Computing (SOCC '18)*, Carlsbad, CA, 2018.

[36] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow *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.

[37] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, MA, 2014.

[38] 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)*, Chicago, Illinois, 2014.

[39] H. Mekky, F. Hao, S. Mukherjee, T. Lakshman, and Z.-L. Zhang, "Network function virtualization enablement within sdn data plane," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, Atlanta, GA, 2017.

[40] G. Liu, Y. Ren, M. Yurchenko, K. Ramakrishnan, and T. Wood, "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.

[41] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr, and D. Kostić, "Snf: Synthesizing high performance nfv service chains," *PeerJ Computer Science*, vol. 2, p. e98, 2016.

[42] B. Deng, W. Wu, and L. Song, "Redundant logic elimination in network functions," in *Proceedings of the Symposium on SDN Research*, 2020, pp. 34–40.

[43] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*, Florianópolis, Brazil, 2016.

[44] 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)*, Atlanta, Georgia, 2016.
[45] "PAN," https://www.paloaltonetworks.com/.
[46] "iptables," https://www.netfilter.org/projects/iptables/index.html.