NFD: Using Behavior Models to Develop Cross-Platform NFs

Paper # 135, 12 pages

Abstract

The flourishing of NFV ecosystem drives more and more NF runtime platforms to appear, making NF vendors difficult to rapidly deliver many NFs for different environments. We propose an NF development framework named NFD for cross-platform NF development. NFD's main idea is to decouple the packet processing logic from the environmental adaptation logic —it provides a platform independent language to program NFs' behavior models (i.e., the flow processing logic), and a compiler with interfaces to develop platform-specific plugins (i.e., the environmental adaption logic). By enabling a plugin on the compiler, various NF models would be compiled to runnable executables integrated with the target platform. We prototype NFD, build 14 NFs, and support 6 platforms (standard Linux, OpenNetVM, GPU, SGX, DPDK, OpenNF). Our evaluation shows that NFD can save development workload for cross-platform NFs and output valid and performant NFs.

1 Introduction

With network functions' (NFs) capability to enhance network performance and security, the ecosystem of network function virtualization (NFV) has gradually matured over the past few years. Various NF platforms have emerged to improve NFV performance (DPDK, SR-IOV, AWS Nitro[15]), security (SGX), scalability (Azure VFP[31], OpenNF[33]), and manageability (E2[60], LeanNFV[6]). In the runtime, network users such as cloud tenants usually require NF to perform certain functionalities, and network operators could require NFs to leverage platform features. The task of delivering NFs satisfying both functional and environmental[1] requirements is usually undertaken by NF vendors, which process, however, is not trivial.

On the one hand, an increasing number of NF platforms are appearing recently, and NF functional requirements can be highly customized (e.g., load balancer with blacklisting, or unbalanced load balancer for heterogeneous backends), which leads to a large number of combinations. On the other hand, porting an NF to a specific platform involves a non-trivial business cycle — developers need to spend effort understanding NF logic, decoupling and rewriting the environmental logic, developing, and testing. Such a contradiction would potentially slow down NF vendors to deliver NFs and become an obstacle to the prosperity of the NFV technology. Thus, our goal in this paper is to explore the possibility *to rapidly develop NFs for diverse platforms.*

Our intuition to solve this problem is to use a platform independent language to develop NFs (namely NF models) and use a compiler to adapt their implementation to the platform. Such intuition is inspired by the following study and observations. (1) (Methodology) In software engineering and programming language areas, programmers use machine independent languages to develop programs, and the compiler would translate the programs against target heterogeneous machines. One famous example is Java's "write once, run everywhere". (2) (NF modeling) It is possible to build a domain specific language for NFs. Several NF models or modeling languages[18, 32, 42, 50, 56, 59, 62, 71] have already been proposed for the purpose of network management (e.g., verification), and most NF development frameworks usually propose NF programming abstractions (e.g., APIs) to represent common semantics in NF logic (e.g., parsing, streaming). (3) (NF-Environment Adaptation) According to many experiences, when NFs are integrated with one platform, the developers usually follow a common process — anatomizing a specific piece of logic and refining or rewriting that piece with the environment specific programming abstractions/libraries(§2).

Therefore, we design an NF development framework named NFD, which *decouples the NF packet processing logic from environmental adaptation logic*. NFD consists of a domain specific language and a compiler; the NFD language is platform-independent and used to develop NFs, namely NF models; the compiler has open interfaces to build environmental plugins which operate on NF models to integrate various environmental features. In a practical workflow, NF developers would develop NF models, and platform providers[2] would develop platform specific plugins for the compiler. By enabling a plugin and compiling the NF model, the output NF program (i.e., code) would integrate both the NF functional logic and the environmental logic. Thus, both pieces of logic can be developed and evolve simultaneously and independently.

The NFD language is designed with the following features. (1) It inherits several programming abstractions (e.g., flow abstraction, state abstraction, user-defined functions[61]) in existing development frameworks, which simplifies the NF model programming. (2) We enrich the language with several new programming abstractions (i.e., time-driven logic

[1]Platform and environment are interchangeable in this paper

[2]They can be network operators who integrate various hardware/software in the platform; or they can be NF vendors who would like to sell NFs to a new platform; or they can be platform hardware/software vendors such as SGX, GPU, etc.

and state abstraction) to make the language more expressive in semantics and more friendly in environmental integration. (3) It integrates complete control flow logic (i.e., if-else branch, and while loop) from high-level languages, which guarantees the language capable to express equivalent logic as general programs.

The NFD compiler parses an NF model and outputs its syntax tree, namely NF intermediate representation (IR). The compiler provides interfaces to develop environmental plugins. The interfaces include a syntax tree traversing function and the function prototype of basic symbols (i.e., leaf nodes of the tree). The tree traversing function can modify the IR or collect information, and the prototype can be overridden by a different implementation. By these means, the platform developer can integrate environment specific features. An NF model is finally transformed into a C++ program, which would be further compiled to an executable by platform specific compilers (e.g., GNU g++, GPU NVCC, or SGX compiler).

NFD overcomes a challenge in designing a suitable structure for NF models. There is a potential contradiction in designing such a structure — NF models had better follow a universal structure so that one plugin targeting the structure can be applied to many NFs, saving development efforts; but actual production NFs are developed in a more free format in high-level languages. We formally propose the stateful match-action table (SMAT) as the universal structure. We show the following facts of SMAT: (1) the SMAT structure is able to express the control flow logic in general high-level languages, (2) many NFs in the practice can be developed using SMAT, and (3) one plugin targeting SMAT structure can be applied to all SMAT NFs.

We prototype NFD, and develop 14 NFs on 6 platforms. The platforms are standard Linux, DPDK, GPU, SGX, OpenNF, and OpenNetVM. The evaluation shows that NFD can be used to develop NFs with environmental adaptation, correct logic, and satisfactory performance. In this paper, we make the following contributions.

- Design and prototype NFD, the first solution for cross-platform NF development. NFD leverages domain specific language and compiler technologies to decouple packet processing logic and environment adaption logic, which can significantly reduce NF delivery cycle.
- Implement and contribute 14 NFs on 6 platforms as well as a commodity equivalent complex NF to the community. These NFs are validated to have correct functional logic and satisfactory performance compared with commodity NFs, and the development process shows the NFD can reduce development workload.

The remaining part of the paper is organized as follows. §2 gives the motivation and goal to design NFD; §3 is an overview including NFD language, model, compiler, and platform plugins. §4 elaborates the language and model design; §5 elaborates the compiler architecture and NF intermediate representation; §6 shows the compiler interfaces and use cases to build environment adaption plugins. §7, §8, and §9 describe the implementation, evaluation, and use cases. We finally discuss the scope, progress, and possible future improvements in §10 and related work in §11. §12 concludes the whole paper.

2 Motivation and Goal

Except for undertaking various network services, NFs must be integrated into a runtime production environment, which includes both performance/security enhancements in the data plane and the management interactions from the control plane. We summarize a few practices where NFs are integrated with existing network infrastructure, and point out that this integration process is usually tedious, taking too much manual effort from developers.

Example #1: Accelerating NF I/O using DPDK. Data Plane Development Kit (DPDK) allows an application to send/receive packets directly to/from NICs, which bypasses the protocol stack in OS kernels. DPDK can significantly accelerate packet I/O in NFs, and thus draws wide attention[9]. To apply DPDK in many existing NFs, the NF developers need to identify the packet I/O logic in NF programs and replace the I/O function as well as the corresponding data structure. In Figure 1, the top code is the original packet forwarding IO code extracted from Snort and the bottom code is ported to DPDK platform by using the IO functions prefixed with *rte_eth* as well as using DPDK's way to loop on packets. While the process seems straightforward (276 changes in nDPI[12]), it would take a lot of effort to port many existing legacy NFs to a DPDK environment.

Example #2: Accelerating NF processing using GPU. GPU naturally supports parallel processing, which is applied in NFs to process multiple packets or multiple chunks in one packet in parallel (e.g., pattern match, parallel encryption[34, 37, 51, 75–77, 80, 81]). When applying GPU acceleration, NF developers need to identify the location of the operators, build the GPU-based implementation, and make the replacement. Similarly as Example 1, this replacement needs to be performed on NFs one by one.

Example #3: Integrating NFs to OpenNF. OpenNF is a network controller which jointly controls flow routing and NF placement in a network. It could flexibly scale NFs out and in. To integrate an NF with OpenNF, the NF developer needs to add a local agent in an NF, which communicates with the OpenNF controller and operates on NF local states (add/remove/modify). This is usually not a trivial process; as described by [33, 49], modifying Prads and Snort takes more than 100 man-hours respectively. In Figure 2, the top code is also the packet I/O from Snort, the bottom part shows several modifications: starting the agent and initializing of the flow state handling functions, and the function needs to be implemented for each NF individually to define the ways of operating flow states.

```
//=== Snort.c with libpcap ===
int main(int argc, char* argv[]){
 ... // initialization
 pcap_loop(phandle,−1,pcap_handle,NULL);
}
```

```
//=== Snort.c with DPDK ===
int main(int argc, char* argv[]){
 ... // initialization
 for (;;) {
  struct rte_mbuf *bufs[SIZE];
  rte_eth_rx_burst(port, 0, bufs, SIZE);
  ...
}}
```

Figure 1. DPDK for Snort

```
//=== Snort.c without OpenNF ===
int main(int argc, char* argv[]){
 ... // initialization
 pcap_loop(phandle,−1,pcap_handle,NULL);
}
```

```
//=== Snort.c with OpenNF ===

int main(int argc, char* argv[]){
 ... // initialization
 locals.put_allflows = &put_allflows;
 sdmbn_init(&locals); // start agent
 pcap_loop(phandle,−1,pcap_handle,NULL);
}
```

Figure 2. OpenNF for Snort

```
//=== PRADS.c ===
void check_vlan (packetinfo *pi) {
 config.pr_s.vlan_recv++; // a state
 ... }
void prepare_ip4 (packetinfo *pi){
 config.pr_s.ip4_recv++;  // a state
 ... }
```

```
//=== SGX Config ===
enclave {
 ...
 trusted{ public void check_vlan(...);
          public void prepare_ip4(...);
};};
```

Figure 3. SGX for PRADS

Example #4: Securing NF states using SGX. Outsourcing NFs into an untrusted environment (e.g., a public cloud) usually causes security concerns for NF users. A set of work proposes to apply Intel SGX to protect NF states from the untrusted underlying OS[35, 64, 70]. However, this modification is still non-trivial: the NF developer needs to identify the sensitive code and data in the NF (usually NF states) and seal them with SGX abstractions. In Figure 3, the top code is two functions from PRADS [3]; assuming the two variables on the top code are sensitive data requiring SGX protection, the developer needs to identify the functions operating on them and put those functions into the SGX config file on the bottom. For example, when Han et al. port an IDS to Intel SGX, it brings about 2.5k extra lines of code in the modification[35].

Goal. Except for these examples, more and more diverse runtime environments appear in the NFV ecosystem, and NF vendors would face a problem to quickly deliver NFs to various environments (either developing NFs from scratch or porting NFs between platforms). Thus, in this work, we aim to build *a framework which enables NF developers to build cross-platform NFs in an agile way.*

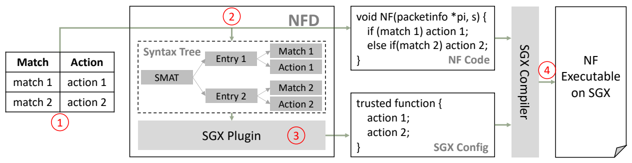
As a development framework, the following **requirements** should be satisfied. First, it should be expressive to describe the packet processing logic in various NFs. Second, the outcome NFs should be logically correct and have comparable performance compared with existing legacy NFs. Finally, as the specific requirement in this work, it should save the development workload in the NF-environment integration.

3 Design Overview and Challenge

Intuition. Summarizing the four environmental adaptation cases above, we get the following intuitions to design the development framework. First, most NF-environment integration targets *a specific piece of logic* (e.g., IO in DPDK, states in OpenNF), and thus the NF development framework had better incorporate corresponding programming abstractions to express these NF specific semantics.

Second, in NF-environment integration, the developer entails a means to *identify the location* of the target piece of logic and make the environmental adaptation. Thus, the NF development framework had better provide such interfaces for logic identification in NFs.

Third, the *diverse NF-environment integration operations* include override (Example 1 and 2), modification (Example 3), and retrieval (Example 4), but one environment usually uses one operation for all NFs in it. The development framework had better provide the programming interfaces to automate these operations to many NF programs.



```
void NF(packetinfo *pi, s) {
  if (match 1) action 1;
  else if(match 2) action 2;
}
```

```
trusted function {
  action 1;
  action 2;
}
```

Figure 4. Example of NFD workflow

Framework Architecture. Based on the intuitions, we design an NF development framework named NFD. Its main idea is to *decouple the NF packet processing logic from environmental adaption logic*, where both pieces of logic can be developed and evolved independently and composed in a flexible way. The whole framework consists of a platform independent language to express NF logic and a compiler to integrate environmental logic.

NF developers use NFD domain specific **language** to describe NF behaviors (i.e., the flow processing logic). The language consists of basic elements such as constants, variables, operators[3], statements, and control flows. Specifically for NF programming, NFD explicitly defines programming abstractions such as flows, headers, states (they are variables), flow/state predicates (they are boolean expressions), and flow/state policies (they are assignment statements). These language elements (both operators and operands) are organized by the language syntax. The code developed in NFD language to describe NF behaviors is also named as *an NF model.*

The NFD **compiler** parses the NF model, provides the NF model's syntax tree as NF intermediate representation (IR) and interfaces of a tree traversal function and operator function prototypes. Platform developers can implement a *plugin* using the traversal function, which adds platform enhancing logic into the IR (modification) or collect information from IR for platform configuration (retrieval). Platform developers can also replace the function prototype by a platform

[3]"Operator" (alone) means a language element that operates on operand; "network operator" means a role in production networks which manage the network infrastructure.

customized implementation (override). Combining plugins and NF models, the final executable both keeps the original packet processing logic and adapts to a specific runtime environment.

The **workflow** of porting an NF to SGX platform is shown in Figure 4, where the NF internal states (e.g., flow statistics) should be identified and sealed in the SGX memory. (1) The NF vendor develops the NF model in NFD language. (2) Then the NFD compiler would analyze the model and output the program (C++ code) and its syntax tree. (3) The developer writes a compiler plugin which traverses the syntax tree and finds the functions that operate on states, and the plugin outputs an SGX configuration (describing the state variables and functions to be sealed in SGX). (4) The SGX compiler compiles the program with the SGX configuration and generates the binary executable.

Challenge. In the design of NFD, a challenge is to design *a suitable structure* of NF behavior models. On the one hand, having many NFs follow a universal structure would ease the plugin development. Because a plugin targeting the universal structure would be applicable to all NFs following that structure. On the other hand, practical legacy NFs from various existing platforms and vendors are seldom developed in the same standard or structure, and many of them are developed in a free format in high-level languages (e.g., C/C++ or Rust[52, 61, 82]). The structural requirement for cross-platform development contradicts the state-of-the-art NF development practice.

We propose the universal structure to be *a stateful match-action table, i.e., SMAT*, which is a tradeoff between development flexibility and environmental portability.[4] (Practice) Many NFs are developed under the SMAT structure [31](§ 5); (expressiveness) jointly using SMAT, "resubmission" operator, and flow tags can express typical control flows (if-else-then branch and while-loop); (portability) and a plugin targeting SMAT can modify all SMAT NFs.

4 NF Modeling

We introduce the NF modeling language and the corresponding NF programming abstractions. And then, we show the representative SMAT model structure and several NF examples.

4.1 NF Modeling Language

Figure 5 shows the NFD language syntax. NFD language first contains basic language elements in general high-level programming languages (e.g., C++, Rust), including basic types, expressions, predicates, policies, and control flows, so that the language can be *semantically complete* to express all existing NFs developed in high-level languages. The basic types

[4]Using the structure is not mandatory but could benefit environment adaption in later stages. Even an NF is not developed using SMAT, the NFD language is expressive and semantic complete for NFs — it contains the high-level language elements and programming abstractions.

Basic types and expression

const	c	::=	$(0\|1)^+$
header field	h	::=	$sip\|dip\|sport\|dport\|proto\|\ldots$
state	s		
variable	var		
expression	e	::=	$c\|h\|s\|var\|e\|Expr\_Op(e_1, e_2, \ldots)$

Predicates

flow predicate	x_f, y_f	::=	$\epsilon\| * \|h = c\|\neg x_f\|x_f \wedge y_f\|x_f \vee y_f$
state predicate	x_s, y_s	::=	$*\|Rel\_Op(s, e)\|\neg x_s\|x_s \wedge y_s\|x_s \vee y_s$
general predicate	x, y	::=	$Rel\_Op(e_1, e_2, \ldots)\|\neg x\|x \wedge y\|x \vee y$

Policies and Statements

flow policy	p_f, q_f	::=	$h := e\|p_f; q_f$
state policy	p_s, q_s	::=	$s := e\|p_s; q_s$
general policy	p, q	::=	$q := e\|p; q$

Model

model	$model$	::=	$stmts$
statements	$stmts$	::=	$stmt\|stmt; stmts$
statement	$stmt$	::=	$p\|if\|loop$
if statement	if	::=	if $(x)\{(stmts\}$ else $\{stmts\}$
loop statement	$loop$	::=	while(x) then $\{stmts\}$

Figure 5. NFD language for NF models

and expressions include variables, constants, and expressions whose semantics follow their mathematical definition[5]. A predicate is a boolean expression with a true/false value. A policy is an assignment which transfers the value of the left-hand expression to the right-hand variable. A control flow (execution path) can be *if-statement* or *loop-statement*. An if-statement decides the execution branch according to the boolean expression in the condition field; a loop-statement would repeatedly execute the body as long as the boolean expression is true. Lastly, an NF model is a sequential execution of simple statements (i.e., policy) or compound statements. A formal definition of the language semantics is in [5].

NFD also contains elements with NF semantics, including the header field variables, the state variables, the flow/state predicates and policies. In an NF model, these NF specific elements are explicitly declared, and there are two benefits. First, NF specific operations such as packet parsing can be directly expressed (instead of writing the parsing logic in raw code); second, the explicit definition could help to identify NF related logic in NF-environment integration.

Table 1. Derived symbols in NFD language

symbols	meaning
$f[h]$	h is a header field (Figure 5 does not list all fields), and $f[h]$ is the field h in packet f.
$f[TAG]$	We append tags to each packet for flexible processing[29], which can be viewed fields of a packet.
$f[output]$	Record the output ports of a packet. $f[output] := \{p_1, p_2\}$ means sending packet f to port p_1 and p_2. $f[output] := \epsilon$ means dropping the packet.
r	Abbreviation for A rule: $h_1 = v_1 \wedge h_2 = v_2 \wedge \ldots$
$f \sqsubseteq r$	Abbr. for a flow-rule match: $f[h_1] = v_1 \wedge f[h_2] = v_2 \wedge \ldots$
R	Abbreviation for a rule set: $\{r_1, r_2, \ldots\}$
$f \sqsubseteq R$	Abbreviation for a flow-ruleset match (f match one of rules in R): $f \sqsubseteq r_1 \vee f \sqsubseteq r_2 \vee \ldots$

[5]The $Expr\_Op(e_1, e_2, \ldots)$ operator in NFD language is an abbreviation of various operators: currently, we implement arithmetic (+, −, ×, /, ++, −−), set (∩, ∪, \), index ([]) , and user-defined ($Encrypt$, $Hash$, NAT) operators. NFD uses $Rel\_Op(s, e)$ to stand for rich relations: currently, we implement equality(=, ≠), scalar(>, <), set (⊂, ∈), and user-defined ($PatternMatch$) relation operators.

We further define a few derived symbols and notations of the language in Table 1 to simplify later text description. The "[]" operator has multiple meanings: (1) if the input is a packet header field (e.g., *sip*, *dip*), it would parse the packet to the corresponding layer and further fetch or modify the field value; (2) if the input is a tag (e.g., *BR*, *output* in §4.3), the operator would look up or modify the map structure; (3) if the input is an attribute (e.g., *size* in the rate limiter example below), the operator would compute and return corresponding value; (4) $f[output] := resubmit$ means the packet is resubmitted to the table; and $f[output] := timer(t)$ means the packet is resubmitted to the table after time t, which is used to develop time-driven logic. The "$\sqsubseteq$" denotes as a flow predicate, which means whether a flow matches a rule (i.e., values in multiple fields) or a rule set.

4.2 Representing NF Programming Abstractions

The NFD language is able to express a large variety of programming abstractions in existing NF programs and NF programming frameworks [17–19, 21, 52, 57, 58, 60, 61]. We list them as follows.

Packet processing abstraction. We inherit packet processing abstractions from existing development frameworks (e.g., NetBricks, OpenFlow). "parse", "deparse", and "transformation" can be expressed by the expression $f[h]$, f, and $f \sqsubseteq R$, and "filter" is $if(f \sqsubseteq R)\{f[output] := IFace|\epsilon\}$.

Bytestream processing abstractions. f could also represent a bytestream.[6] Whether f stands for packets or bytestreams depends on the compiler configuration, which decides to use packet I/O or socket I/O; the compilation would also check the semantic correctness of the operators on f (whether an operator is applicable on packets or flows).

User-defined abstraction. We allow users to implement their own programming abstractions. All these abstractions are denoted as $UD\_Op(\text{"}Func\_Name\text{"}, *args)$, but the user should also provide a corresponding implementation like `void Func_Name(*arg)`. Then NFD compiler would link them with the NF program. As examples, we implement the byte stream operators "Encrypt" (encrypting a byte stream) and "PatternMatch" (searching a pattern in a byte stream), and we also implement packet processing operators such as "hash"(in hash-based load balancer) and "NAT" (a non-replacement IP sampling algorithm for IP translation).

NFD also augments two new programming abstractions for NF development, which either eases NF programming and porting or is able to express new kind of logic.

State abstraction. We observe that an NF state is usually associated with a flow of certain granularity, and all operations on the state should fall on one specific instance of that granularity. For example, a 5-tuple "per-flow packet counter" is actually not one counter, but stands for a set of instances with each instance counting one 5-tuple flow's packets. Such states are usually declared as a group of variables in existing frameworks (e.g., array "int counter[1000]" or "map<flow, int> counter"), each state update needs to be accompanied with a lookup operation in the group of variables.

```
  string type="int";  Value value=0;
  int granularity=sip&dip&sport&dport&proto;
 map<unsigned, Value> instances;
  State_Counter& operator++(){
    key=hash(pkt&MaskOf(granularity));
    if(instances.find(key)==instances.end())
      instances.put(key, value);
    instances[key]++;
} };
```

Figure 6. The counter state class of a per-flow monitor: member variables and an overridden operator.

NFD uses a class to abstract the NF state. The state class has an attribute describing its granularity (i.e., a list of header fields, e.g., 5-tuple). The class maintains concrete instances of the same granularity internally, and all operations upon the state class would fall on an instance (by default of the current flow in processing). In the per-flow counter example above, the counter should be declared as
`int counter <sip, dip, sport, dport, proto> = 0.`
The instances of a state class are allocated on demand: NFD overrides all operators to the state class; once a state is operated on, the operator function would first check whether "current flow" has an corresponding instance of that state; if no, a new instance of the flow would be created and added to the state instance map; and then the operator proceeds with the instance. Figure 6 shows the implementation of the state class in the per-flow monitor example including attributes, instances, and an overridden operator ++.

This class abstraction could prevent programmers from making mistakes on states (e.g., declare a state in Figure 6 as a single variable) and its uniform implementation would also benefit later environment adaptation about NF states[33].

Time-driven logic abstractions. Some NFs contain time-driven logic; for example, a rate limiter "periodically" refreshes tokens for packet dispatching. This abstraction was not proposed in existing NF development frameworks. NFD captures it by adding an operator $timer(flow, \Delta t)$ to describe resubmitting a $flow$ after time Δt, which complements the NF time-driven logic.

4.3 A Representative Model and Use Cases

Stateful Match Action Table

entry	$entry$	::=	if $(x_f \wedge x_s)$ then $(p_f; p_s)$ else $\perp$
SMAT	$smat$	::=	$entry \mid entry; model$

Figure 7. SMAT syntax

While NF models can be developed in a free format as long as they follow the language syntax, we suggest a representative structure for NF model. The structure is a stateful match action table (SMAT) whose syntax is in Figure 7. Its semantics is that each entry has a predicate to match flows and states, and if the match result is true, the policy (for flow

[6]In the current implementation, NFD only handles bytestreams sent to/received from TCP socket; thus, window control, packetization, and packet resembling are handed over to the socket library, not in NFD language.

and states) is being taken to action. A table with multiple entries complies with the rule of "the first match applies".

We elaborate SMAT for the following reasons: (1) several existing practices choose SMAT as the behavior model for many NFs; for example, VFP in Microsoft Azure uses this model to develop FPGA accelerated NFs [31], and there are also efforts to reduce NF code to SMAT[79]. (2) If an NF is stateless (e.g., rule-based IDS), its SMAT would degenerate to an ordinary match-action table; which represents a large variety of switch policies. (3) Most importantly for NFD, if many NFs are developed using the structure of SMAT, a plugin can be developed targeting the structure instead of each individual NF. Applying one plugin to the structure can save the total development workload compared with developing separate plugins for each NFs.

We emphasize two facts of SMAT. First, if the "resubmission" operator and flow tags are allowed in SMAT, typical control flows in high-level languages (i.e., the if-else-then branch and the while loop) can be expressed in SMAT. The main idea to prove this claim is like using conditional statements and "goto" (in the C language) to represent branches and loops, and the formal proof is in [5]. Second, NFD is semantically complete for NF development by inheriting high-level language elements and programming abstractions. Even if SMAT is not convenient to develop a specific NF, the development can fall back to using elements in high-level languages, with only a sacrifice of sparing extra effort of developing environmental plugins for that NF.

	Match		Action	
	Flow	**State**	**Flow**	**State**
Stateful Firewall	**Configuration:** OK={r1, r2, ...}			
	f∈OK	-	f[output]:=IFACE	seen:=seen∪{f}
	f	f∈seen	f[output]:=IFACE	-
	f∉ OK	f∉seen	f[output]:=ϵ	-
Stateless Firewall	**Configuration:** DENY = {r1, r2, ...}			
	f∈DENY	-	f[output]:=ϵ	-
	f∉DENY	-	f[output]:=IFACE	-
NAT	f	f∉map	f:=NAT(f) f[output]:=IFACE	map[f]:=NAT(f)
	f	f∈map	f:=map[f] f[output]:=IFACE	-
LB	**Configuration:** mode = ROUND_ROBIN			
	f	f∈ map	f[dip]:=srv[map[f]]	-
	f	*	f[dip]:=srv[idx]	map[f]:=idx idx:=(idx+1)%N
LB	**Configuration:** mode = HASH			
	f	-	f[output]:=hash(f)	-

Figure 8. Examples of NF models

Init: tkn:=TOKEN, f_{dmy}[BR]:=REF, f_{dmy}[output]:=timer(Δt)

Match		Action	
flow	**state**	**flow**	**state**
f[BR]=REF	*	f[output]:=timer(Δt)	tkn:=TOKEN
*	f[size]≤tkn	f[output]:=IFACE	tkn:=tkn-f[size]
*	f[size]>tkn	f[output]:=ϵ	-

Figure 9. The model of a rate limiter

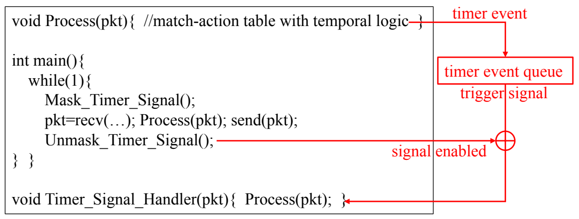
A few typical SMAT NFs are listed in Figure 8, and they are visualized as tables with each entry in the model as a row in the table. They are a stateless Firewall with a blacklist, the stateful Firewall, a stateful NAT and load balancer that both store the consistent mapping of a flow to a backend server, and a stateless load balancer that uses consistent hashing. In addition, we design a rate limiter to validate the *timer* operator— a rate limiter in Figure 9. It uses the leaky bucket algorithm: the rate limiter refreshes tokens periodically; and for each traversing packet, if there are enough tokens left, it is sent by consuming them; otherwise, it is discarded.

5 NF Compilation

NFD compiles NF models to NF programs and also provides the syntax tree of the NF model as its intermediate representation (IR).

NF Code Generation. NFD compiles an NF model to a C++ NF program by the following transformation.

- Most basic elements (e.g., control flows, expressions, predicates, and policies) in NFD language can be implemented in C++ directly.
- States are declared and initialized as global variables at the beginning of the program.
- In SMAT, each entry is translated to an if-branch with the match as the condition and the action as the body. Entries are connected sequentially.
- Time-driven logic is incorporated as Figure 10 depicts. The program initialization and the flow processing can add time events to the timer event queue; the timer signal handler calls the flow processing logic recursively. The timer signal is masked at the beginning of each pass of flow processing and unmasked at the end. Thus, timer events would not interleave with the flow processing iteration, preventing timer events from preempting flow processing and mistakenly polluting states in use.
- All NFs share a common program skeleton for packet I/O: the compiler declares and initializes states at the beginning of the program, wraps up NF model code in an infinite loop and adds a flow receiving/sending function at the beginning/end of the loop. Thus, the NF program would repeatedly fetch and process flows.



```
void Process(pkt){  //match-action table with temporal logic }

int main(){
   while(1){
      Mask_Timer_Signal();
      pkt=recv(…); Process(pkt); send(pkt);
      Unmask_Timer_Signal();
}  }

void Timer_Signal_Handler(pkt){  Process(pkt);  }
```

Figure 10. NF program structure

After compilation, most basic language elements are naturally supported by C++ (e.g., arithmetic operator, control flows). Remaining operations are supported by the NFD library including some complicated operators (e.g., "Pattern-Match", "Encrypt"), "flow" class with "[]" as in § 4.1 and "state" class as in § 4.2. In the final compilation from a C++ program to a binary executable, they would be linked together.

NF Intermediate Representation. An NF model built on the NFD language syntax would follow a tree structure:

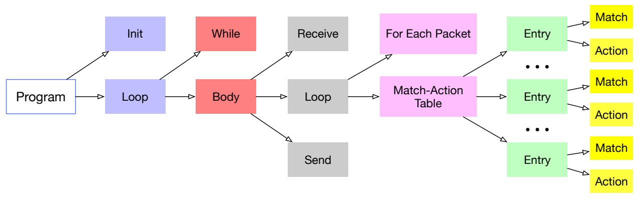


Figure 11. A part of the syntax tree of a SMAT

deriving a program to composing sections (i.e., I/O, initialization, model), deriving each section to statements (i.e., policy and predicate), and deriving each statement to basic symbols (variables, constants, and variables).

The syntax tree is also named *intermediate representation (IR)*. In a syntax tree, the leaf nodes are symbols that cannot derive other symbols, and they are named as basic symbols (e.g., constant, var, flow, state); the intermediate nodes can derive other intermediate nodes or leaf nodes, and they are named as deriving symbols.

For example, Figure 11 shows the syntax tree of a SMAT, where the root "program" derives an "init" block and a "loop" block, the "Match-Action Table" block derives multiple "entry" blocks, and each entry can derive the predicate and policy statements (omitted in the figure). The variables, constants, and operators in predicate and policy statements are basic symbols, and the remaining nodes are deriving symbols.

6 NF-Environment Integration

NFD provides interfaces to operate on IR, by which environmental features can be added to the NF program. We show the cases in § 9, where the integration is performed automatically using NFD.

6.1 Programming Interfaces

The NF-environment integration is achieved by operating on the IR, and the NFD compiler provides two kinds of interfaces for this operation: the prototype of basic symbol functions and a syntax tree traversal function. A function prototype includes its function name and input/output argument types; NFD provides the prototypes of basic symbols (mainly operators in expressions) and allows developers to customize the internal implementation of a prototype.

The tree traversal function is a class with many callback functions, and each callback function corresponds to a kind of symbol in the syntax. A tree traversal function can traverse a syntax tree, and when a node on the tree is visited, the callback function corresponding to the kind of the current node symbol would be triggered. Developers can inherit the class and implement their own logic in the callback functions. In addition, the preliminary compilation(§ 5) is also a tree traversal, which traverses the NF model and generates the preliminary NF program.

We summarize three common operations from existing NF-environment integration experiences, and use NFD compiler interfaces to automate these operations. We use the examples in § 2 to illustrate these operations.

```
    new OpenNFVisitor.visit(syntax_tree);
  }
public class OpenNFVisitor implements NFDCompiler{
  @Override public T visitInit(...){
    AddAgentCode(...)
    InsertCode("List<State>␣allStates")
    super.visitInit(...) // orig. compilation
  }
  @Override public T visitStateDeclaration(...){
    super.visitStateDeclaration(...)
    stateName = ... // get the state name
    InsertCode(String.format("allStates.add(%s)",
         stateName))
} }
```

Figure 12. Code of the OpenNF plugin

6.2 Operation 1: Override

The override operation does not change the structure of an NF IR, instead, it replaces a piece of logic by a platform enhanced implementation. The platform developers can refer to the function prototypes and wrap up platform dependent implementation with the same input/output types, and NFD compiler would link the new implementation with the original program to build a new executable.

Example 1: Accelerating I/O with DPDK. "Receive" is a leaf node in the syntax tree (Figure 11), meaning to receive a flow's packets/bytestream from NIC or OS. When it stands for packet receiving, NFD has a default implementation using libpcap[40]. To integrate DPDK with NFD-based NFs to accelerate IO (i.e., example 1 in §2) [36], the DPDK provider could just wrap up the DPDK packet I/O into the Receive's prototype "`size_t Receive(*args)`", and NFD compiler would apply this DPDK acceleration to all NFs.

Example 2: Accelerating flow processing with GPU. In § 4.2, we define two stream operators — Encrypt and PatternMatch. As user-defined operators, we initially make a CPU-based implementation. We then build the GPU accelerated version of the two bytestream operators above [75] (example 2 in §2). "Encrypt's" prototype is `f Encrypt(f)`; in the GPU implementation, the bytestream is chunked into block and blocks are encrypted in parallel by the GPU (AES counter mode). "PatternMatch's" prototype is `bool PatternMatch(f, patterns)`; in the GPU implementation, multiple `patterns` are matched by GPU threads simultaneously, which significantly speeds up the processing rate. NFD similarly applies these GPU implementations to NF programs.

6.3 Operation 2: Modification

The modification operation changes the structure of the NF IR; it refers to inserting/deleting/modifying a node on the syntax tree, and these modifications would correspond to the logic change in NFs. The platform developer can leverage the tree traversal function to automate this operation: override the callback functions in the tree traversal function where the callback functions modify nodes, run a pass of the tree traversal function and apply the modification to the IR, and finally the modified IR could be compiled to an executable with the platform enhancement.

```
14     new SGXVisitor.visit(syntax_tree);
15   }
16 public class SGXVisitor implements NFDCompiler{
17   List<String> sensitiveFunc;
18   List<String> sensitiveData;
19   @Override public T visitStateDeclaration(...){
20     stateName = ...
21     sensitiveData.add(stateName);
22   }
23   @Override public T visitStateMatch(...){
24     FuncName = ...
25     sensitiveFunc.add(FuncName);
26   }
27   @Override public T visitStateAction(...){
28     FuncName = ...
29     sensitiveFunc.add(FuncName);
30 } }
```

Figure 13. Code of the SGX plugin

Example 3: Modifying IR for OpenNF. To integrate NFs with OpenNF as the example 3 in §3, each NF needs to make three non-trivial modifications: (1) adding the agent code which starts the agent thread in the runtime, (2) adding a collection of all states in the NF so that they are retrievable in state operations, and (3) implementing the interfaces of state operations (get/put/delete).

Using the state abstraction in § 4.2, we build an OpenNF plugin for the compiler to perform (1) and (2), and build an external library linkable to all NF programs to achieve (3). Part of the code of the syntax tree traversal library is in Figure 12. When the NF developer uses this library to traverse the original NF syntax tree, the code generation is modified: when the "init" node is visited (line 4), the OpenNF extension inserts the code that starts an agent thread into the output NF program (line 5); and then it adds a declaration of "List of all states" to collect all states (line 6); when a state declaration node in the syntax tree is visited, a piece of code, where the state is added to the "List of all states" (line 12), is inserted to the NF program. Thus, the OpenNF plugin adds the agent logic and state collection logic.

The platform provider then provides an external library for state operation. In each of the interface about get/put/delete `flows`, the "List of all states" is iterated and each state is looked up to find the instance of the target `flow` (using the operator []); if an instance is found, it is operated; otherwise, the state is passed. Since the data structure of "List of all states" and the operator [] have the same implementation across all OpenNF pass optimized NFs, this external library can be linked with any of these NFs seamlessly; and thus, the goal of rapid NF porting to OpenNF is achieved.

6.4 Operation 3: Retrieval

The retrieval operation does not change the IR itself, instead, it collects extra information and uses the information for platform integration. The developer can leverage the tree traversal function to visit each node on IR and collect corresponding information, and finally, the collected information can be used to generate extra platform dependent configurations.

Example 4: Retrieving States for SGX Protection. Assume an NF developer would like to apply Trusted Execution Environment like SGX [8] to protect sensitive states information from unauthorized OS operators (e.g., in the public cloud), they need to first identify the state related code snippet (including the variables and the functions processing them), then specify to seal them in SGX enclaves in a configuration file, and finally compile the code with the new configuration to get the NF executable on SGX platform.

Figure 13 shows the compiler plugin to perform the state related logic identification. As the SGX plugin traverses the syntax tree, when state declaration node is visited, the state name is recorded in a list named "sensitiveData" (line 21); and similarly when any state operation nodes are visited (state match and state action), the wrapping-up function's names are recorded in "sensitiveFunc" (line 25 and 29). After the execution of the plugin, the state related information is used to generate an extra SGX configuration file, where state variables are declared as protected data segment and state processing functions are declared as a protected code segment. With this SGX plugin, all NFs in NFD could be automatically ported to SGX supported network environments.

7 Implementation

Table 2. Lines of code in NFD partial implementation

Component of NFD	Lines of Code
NFD model grammar	234 (g4)
compiler frontend (automatically derived by Antlr)	4.3k (Java)
compiler backend (generate C++ NF programs)	1137 (Java)
C++ template (program structure, operators) for NFs	752 (C++)
extension for OpenNF	489 (C++)
extension for GPU	668 (C++)
extension for DPDK	167 (C++)
extension for SGX	273 (C++)

We use Java Antlr4 to build the NFD compiler and implement C++ templates for NF programs; Antlr4 also supports to generate the syntax tree traversal function; and we further generate prototypes of basic symbols (§ 4.1). The lines of code of some components are listed in Table 2.

The NFD compiler has a few tunable parameters. (1) It can configure whether to generate a packet NF or a bytestream NF. A packet NF operates on each packet using pcap library[40] and a bytestream one operates on each flow using socket. For example, a packet LB modifies the destination IP and port for each packet, while a stream-level LB terminates incoming TCP connection and relays byte streams to the next TCP connection. As the NF types is configured, the compiler would also perform a semantic check: packet operators cannot be applied to bytestreams and vise versa. (2) For environment enhancements (OpenNF, Intel SGX, GPU, DPDK), the NFD compiler has arguments to decide whether to add the enhancements to NF programs.[7]

[7]By the time of this project, Intel SGX compiler does not support C++ STL. We replace C++ STL classes that are used in NFD by self-developed code. This change does not affect any steps of NFD. It only requires the SGX compiler to compile NF programs with the self-developed code. It would be resolved when Intel release SGX compiler supporting C++ STL.

8 Evaluation

NFD can be used to build NFs with less development workload, and the outcome NFs are valid in logic and performance.

8.1 NFs and Experiment Settings

We developed 14 NFs using NFD, spanning security-featured NFs (e.g., Firewall, heavy hitter detector, and flood detector), LBs (layer-3 and layer-4), NAT, monitors, and rate limiters. The typical NFs in Figure 8 are used for representing results in this section. A complete list and testing results are in [5]. In addition, we also collect several commodity NFs to compare with NFD-based NFs (for logic and performance): they are Snort, Prads, Balance, HAProxy, and Click NAT[3, 4, 10, 14, 16].

All NF tests are on three servers connected to one switch, each server with Intel i9 CPU (10-core, 20-thread), 128GB memory, 10Gbps NIC, three NVIDIA GTX1080 Ti graphics cards, and 1TB SSD. And we collect the network traces in [20] to test our NFs. An NF experiment is performed in one of the following four ways: (1) **Unit-1host**: the network I/O is removed, and a prepared trace file is injected directly into the NF's processing logic, and the NF runs merely on one host; (2) **NS-1host**: an NF runs as a native process on one physical host, and it is chained to a sender and a receiver on the same host using Linux Network Namespace and Open vSwitch (OVS) [1]; (3) **VM-1host**: the NF is wrapped up by a VM (using KVM[11]), and the NF-residing VM is chained to a receiver VM and a sender VM by OVS on the same host; (4) **Native-2hosts**: the NF runs on one host as a native (non-virtualized) process, and it is chained with a sender and a receiver on another physical host.

8.2 Saving Development Workload

Comparing the LoC. Theoretically if we want to build n NFs in m environments, the traditional development method would cause a workload of $O(nm)$, while NFD can presumably reduce the workload to be $O(n + n)$.

We use the lines of code (LoC) to quantify the development workload. As in Table 2, building the NFD framework needs 2123 LoC (234 for language grammar, 1137 for compiler backend, and 752 for NF template; the 4.3k LoC of derived frontend parser is not counted). With this platform established, each of the NF models costs 20 LoC on average. And the four environments cost 1597 LoC in total (489 for OpenNF, 668 for GPU, 167 for DPDK, and 273 for SGX). Thus the total development workload is 1877 LoC.

Among all final NF programs, their platform independent logic is usually 750+ LoC (from the template and SMAT). GPU platform works only for bytestream NFs (IDS, encryption), but the other three works for all. The combination of 14 NFs and 4 platforms cost totally about 70k LoC ($750 \times (489 + 167 + 273) + 668 \times 2$). Without NFD, these workloads would be undertaken by human programmers, which is a significant burden compared with the NFD approach (70k v.s. 1877).

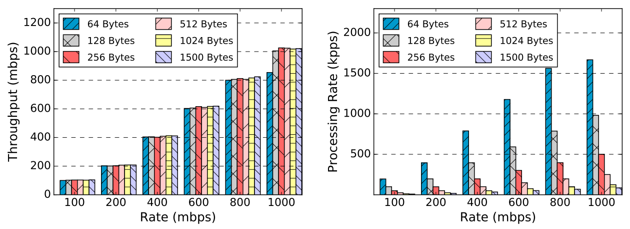


Figure 14. NFD rate limiter performance

Case study of SGX. In another empirical study, we compare the development man-hour of a non-NFD SGX-enhanced network monitor with that of a NFD-based one (details in § 9.1). For a graduate student in our NF developer team, without NFD, it takes one week to learn SGX programming from an SGX expert, it takes another two days to build an SGX-enhanced network monitor (totally 72 man-hours for the student and some consulting time from the expert). While using NFD, the graduate student spent one hour to write a network monitor SMAT model and less than one day to build a SGX plugin (following the requirement from the SGX expert), which only takes 8 man-hours for the student. And that SGX plugin can be applied to any NF SMAT model in the future. NFD shows good potential to improve productivity.

8.3 Individual NF Validation

Logical correctness. Our basic methodology to validate an NF's logic is to use traces to test whether the NFD-based NF has the same behaviors (to packets) as expected (either a commodity NF or pre-computed results). We compare the following pairs of NFs: (1) NFD-based Firewall v.s. Snort (using first 1M packets from the trace and tuning alert rules), (2) NFD-based bytestream LB v.s. Balance and HAProxy (tuning round-robin or hash mode), (3) NFD-based NAT v.s. Click NAT (tuning internal and external address pools). In the test result (in [5]), if NFs perform deterministic behaviors (e.g., IDSes, round-robin LBs), NFD-based NFs have the same behavior with the commodity NFs; if NFs have random behaviors (e.g. hash-based LB, NATs), the behaviors for each individual flow are not exactly the same between the NFD-based NFs and the commodity ones, but flows' total behaviors for a pair (of the NFD-based NF and the commodity one) follow the same distribution (e.g., uniform distribution from frontends to backends in hash-based LBs, non-collision mapping from an internal address pool to an external one in NATs).

We then test whether NFD-based rate limiter has the expected rate control for flows. We set up a VM-1host experiment for the rate limiter, and tune the sending rate and the packet size. We draw the actual packet processing rate and throughput in Figure 14. We conclude that there is an upper bound of the packet processing rate, which is about 1.67 mpps (the 64B bar of the rightmost group). And if the target rate is not too large to violate this packet processing rate (i.e., Control_Rate/Packet_Size < 1.67 mpps), the rate limiter can control the sending rate accurately as the configuration.

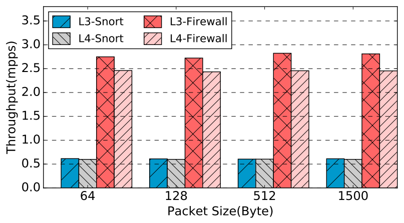


Figure 15. Unit test of NFD FW v.s. Snort

Performance (Unit-1host). We test whether NFs' performance is acceptable. We repeat the unit test on Firewall, tune the number of rules and granularity of rules, and measure the throughput and packet processing rate. Figure 15 shows the performance in the case of 10 rules which deny traffic with a few IP (layer-3) or IP+Port (layer-4). We observe that (1) NFD-based firewall performs significantly better than Snort (2.5mpps v.s. <0.5mpps). By looking into the code, we find that the performance gap is from the implementation of flow-rule match: Snort uses linked list to store rules from the config and matches a packet one by one; but the rules in the NFD model are finally embedded into the code, our firewall has higher performance. [8] (2) The NFD-based firewall configured with layer-3 rules has better performance than that with layer-4, but Snort does not show this trend. The reason is that Snort blindly parses any packets to layer 4, but NFD firewall would adapt the parsing depth to the configuration.

The performance of bytestream LBs lies in Figure 16. The experiments are under unit-1host (using the socket for interprocess communications between the sender, the NF, and the receiver). LBs are in round-robin mode and there are five backend servers in each experiment. We tune the number of incoming flows from the frontend. We observe that (1) NFD-based LB always has higher throughput than HAProxy, and it also outperforms Balance when there is only one flow. The reason is that Balance and HAProxy are commodity NFs with a lot of extra features (e.g., group-based round-robin in Balance, consumer-producer based I/O model in HAProxy). Although we carefully turn unused features off to make the comparison fair, the Balance and HAProxy program still silently executes some unused features, wasting CPU cycles. (2) Balance would outperform the other two LBs when there is more than one flow. The reason is that Balance would create a process (`fork()`) for each new connection, and thus leverage the multiple cores on the machine. But this advantage fails to increase when the server side is fully loaded (i.e., >5 flows for 5 backend servers).

We make a complete test for all NFs and list their performance in [5]. Unit performance tests show that NFD-based NFs have acceptable performance, and in a lot of cases they can be viewed as micro-services [30] without redundant features, which gives them better performance.

Performance (Native-2hosts). We put NFs into a synthesized environment to see whether they would become the bottleneck of the system. We choose various NFs (stateful Firewall, stateless Firewall, NAPT, layer-3 LB) and tune the packet size (64B-1500B). Figure 17 shows the throughput when these NFs uses DPDK or libpcap. We observe that libpcap usually achieves <1mpps throughput, while DPDK NFs can achieve 1-5mpps throughput. For DPDK NFs, the throughput is constrained by the total bandwidth 10Gbps. Thus, NFD NFs could process packets at a line rate of the NIC.

[8]We use Snort 1.0 which only contains layer-3 and layer-4 parsing, and thus we can exclude the possibility that Snort has other CPU-consuming logic.

9 Development Use Cases

We show the use cases of integrating NFs with environments and build a functionally complex NF (comparable with commodity NF) based on existing micro services.

9.1 NF-Environment Integration

Accelerate processing with GPU. Atop CUDA Toolkit[2], the two operators — encryption and pattern matching — are integrated with bytestream NFs. The performance result is in Figure 18. We have the following observations: GPU operators need more preparation time (e.g., copy data from memory to GPU), but accelerate performance by parallel computation. In Figure 18a, GPU is slower in encrypting <6KB bytestream, but faster in large bytestreams; because the encryption chunks a bytestream and encrypts blocks in parallel. Similarly (Figure 18b), GPU could match >5K patterns at a faster speed than CPU, but slower for <5k patterns, because these patterns are matched in parallel.

Alternative packet I/O using DPDK. NFD provision NFs with different packet I/O drivers (DPDK and Libpcap) and deploy them in the path of two end-hosts. Figure 19 shows that end-to-end RTT in VM-based test. Benefiting from the kernel bypass technology, DPDK has about 10X smaller RTT than libpcap (405us v.s. 6952us).

NF state management with OpenNF. We port NFD-based NFs to an OpenNF platform. We use NFD-based Firewall to replace the NFs in the state move experiment in the OpenNF report (§8.1.1 and Figure10 in [33]) and repeat the experiment. We observe NFD-based Firewall successfully interacts with the OpenNF controller, and the experiment result is in [5]. We draw the similar conclusion as in OpenNF[33]: (1) the stricter state migration requirement (no guarantee (NG) > loss-free (LF) > order-preserving (OP)) makes the state move time and packet latency longer; (2) the optimizations (parallelizing (PL) and late-locking-early-release (LLER)) in OpenNF improve the state move time and packet latency.

Enhance NF security with SGX. We use NFD to generate three pairs of NFs —flow counter (FC), packet load balancer (LB), and NAPT. Each pair has one NF without SGX protection and one with it to protect states. We set up Unit-1host for these NFs, tune the number of flows, and measure their performance in Figure 20. We observe that NFD NFs can achieve 1mpps in SGX environment, which is acceptable. But compared with the same setting without SGX, where

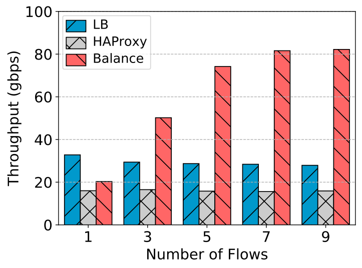


Figure 16. Load balancers

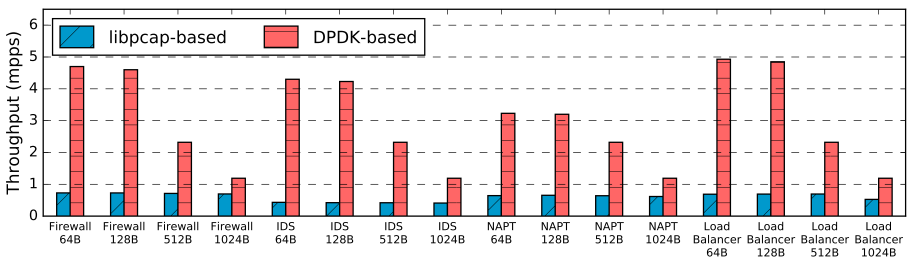
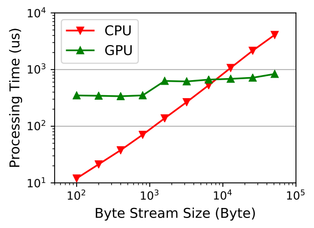
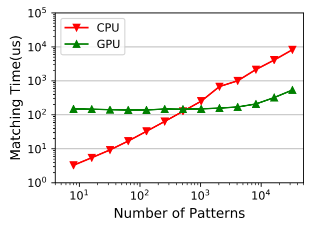


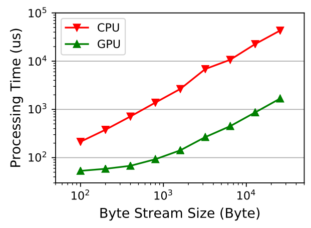
Figure 17. Performance of 4 NFs, tuning packet size



(a) Encryption scaling up byte stream size



(b) Pattern matching scaling up number of patterns



(c) Pattern matching scaling up byte stream size

Figure 18. GPU acceleration (Unit-1host)

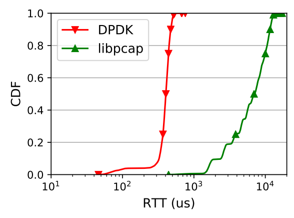


Figure 19. Packet I/O: DPDK versus libpcap (VM-1host)

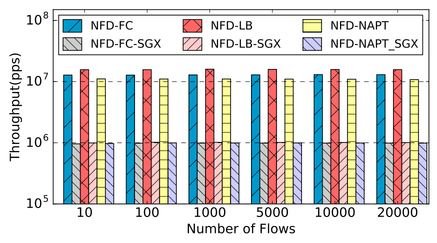
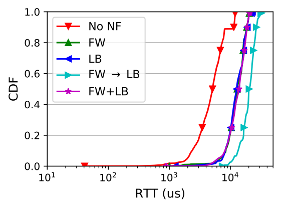
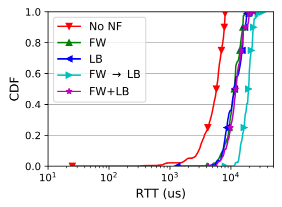


Figure 20. NF performance with and without SGX



(a) VM-1host



(b) NS-1host

Figure 21. RTT of NF chains

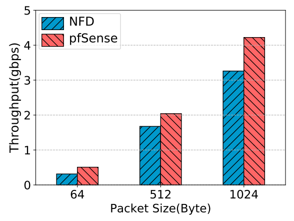


Figure 22. NF performance compared with pfSense

the throughput is usually >10 mpps[9], we conclude that SGX environment is the bottleneck.

9.2 NF Logic Customization

Replace NF chains by merging them. Network operators may require multiple functionalities for a network. The current typical solution is to set up individual NFs and chain them by routing traffic traversing them. This approach usually increases network management complexity. NFD could provide an alternative solution — merging NF models in the development instead of chaining them in the runtime management, and handling the development complexity by the NFD compiler. In addition, the merged NF has lower latency (fewer hops) than chaining NFs.

For example, a cloud network requires a load balancer with blacklisting; this can either be implemented by chaining a firewall with a load balancer (denoted as "FW⟶LB") or by merging a firewall model with a load balancer model and being compiled as a whole program (denoted as "FW+LB").

[9]This number is large. Because SGX does not support C++ STL, and we replace the map data structure by an array in all four NFs.

Figure 21 shows the CDF of the time of delivering packets from the sender to the backend server on KVM testbed with these two approaches.

We observe that each NF (FW, LB) increases the network latency (median) from 5087us (baseline, "No NF") to 12895us (FW) or 12637us (LB), and Chaining NFs doubles the latency (20331us in "FW⟶LB"), but merging them does not increase more latency compared with a single NF (13831us). Thus, NFD provides a more appealing alternative approach like NF consolidation for NF chaining.

Build an NF equivalent to a commodity NF. pfSense[13] from Netgate has now become a prevalent NF for in-network security. It embraces several core features[10] in the data plane, including firewall, NAT, LB, and rate limiter. We make an equivalent implementation in NFD by concatenating the SMATs of these four NF models and compiling the merged model to a synthesized NF program.

[10]For other features, pfSense can provide other off-path data plane services such as DHCP and DNS, which should be provided independently instead of being synthesized with the four on-path functionalities. The web GUI of pfSense is in the control plane, and we do not consider it in NFD.

Evaluation shows that the NFD-based NF has equivalent logic compared with pfSense, but small performance degradation (Figure 22). For example, when the packet size is tuned to 512 bytes, NFD NF achieves the throughput of 1.676 gbps, which approaches pfSense of 2.08 gbps.

10 Discussion

Future Improvements and Complements. NFD is insufficient in several aspects but we aim to improve them. (1) The current implementation of NFD compiles NF models to C++ language. Thus currently it does not work with a platform that does not support C++, e.g., FPGA and P4, which we plan to extend in the future. (2) The NF model only captures the NF logical behavior, without defining the software robustness aspects such as failure handling (e.g., `malloc()` and `connect()` failure). Thus, the NF program auto-generated may need to be further improved to add those handling. In the future, we plan to extend the NF model to cover those software robustness behaviors. (3) Many of the NFs we covered are micro services. Except for pfSense, we would go on working with vendors on commercial complex models. (4) We plan to add multi-core support for NFD compilation, which includes parallel executing of the NF model, dispatching packet to threads, and concurrency control on states; we would refer to existing experiences such as Kargus[41].

Deployment progress. NFD is currently anonymously open sourced in [5]. Its model-based NFs have recently be released on OpenNetVM[82].

Other benefits. Model-centric NF development is one step towards the standardization of NFV. Many management solutions use behavior models to represent the data plane components and further perform their management. Examples are symbolic execution in network verification[71] and model checking in NF software testing[26]. Model-centric NF development can give operators the confidence that the model represents the actual behavior without semantic gaps, laying the foundation of all model-based management solutions.

11 Related Works

In recent NF development frameworks, one significant progress is to summarize NF programming abstractions, such as packet parsing, filtering, transformation[49, 61]. As for many cases in NF building and porting, an NF level behavior model (which organizes these programming abstractions) would reduce the development complexity significantly. NFD's behavior model is a structure in granularity between programming abstractions[17, 18, 52, 58, 59, 61, 65] and monolithic piece of software which helps environmental integration. Other operators or interfaces that are currently not covered by NFD can also be incorporated.

Some frameworks propose modular NF programming[51, 52, 55, 57, 69], which eases the composition of modules; but how to define the granularity of each module's logic and how to organize intra-module logic are still ad hoc and depend on programmer's mind, which complicates the later NF modification and porting. NFD's representative structure can alleviate this complexity.

Traditional NF management frameworks[27, 38, 44, 45, 54, 60, 66, 67, 72, 78, 82] view NF as monolithic logical unit, which does not help logic design inside NFs. And several platform specific development/porting solutions are bound with the environment related features (e.g., SGX, GPU, OpenNF, DPDK)[2, 8, 9, 33–36, 44, 55, 56, 64, 70, 75, 80, 81], which lack cross-platform abstractions. Thus, we believe NFD would complement the insufficiency of existing development frameworks/methodologies by NF logic (re-)design and cross-platform adaptation.

A few recent NFV frameworks are proposed for various requirements[39, 44, 53, 63, 65, 68, 73], into which NFD would be a great help to port NFs. SNF[46] proposed DAG-based NF chains can be synthesized to eliminate cross-NF redundancy, where NFD NF models can contribute to the synthesizing. SCC[47] is a profiling-based NF chain acceleration solution, while NFD can help instrument NF models for NF performance profiling. NFV-Throttle[24] uses an agent to observe NF's resource utilization (for rate throttling), NFD can even instrument agent inside an NF for more accurate monitoring. In NF benchmarking systems[22, 23], NFD can integrate a profiler inside of NFs to collect more accurate information. Like DPDK, other IO acceleration solutions (e.g., mTCP[43]) can also override NFD I/O. CHC requires NF states to be identified for integration[48]; S6 requires states to be shared among NF instances[78]; Metron[45] requires to anatomize NFs and offloads stateless part to hardware; VFP and Eden[19, 31] also implements NFs on both software and hardware. For these frameworks, NFD would be beneficial by automating NF program analysis and porting using its compiler plugin.

NFD is inspired by several works. (1) Packet processing operators (e.g., "resubmit") are also used in OVS and P4[1, 7]. (2) Devoflow[25] proposes "rule clone" to control switch rule explosion, and NFD adopts this idea in the state abstraction. (3) The model language is inspired by several NF modeling works (like DFA[28, 74] and stateful table[79], and also NF modeling language[18, 62, 71]).

12 Conclusion

We designed a cross-platform NF development framework named NFD. It has a platform-independent language to develop NF models; its compiler provides interfaces to operate on the model and integrate platform-specific features. We show the cases to develop 14 NFs with 6 platforms of standard Linux, GPU, DPDK, OpenNF, SGX, and OpenNetVM. Our evaluation demonstrates NFD's feasibility by developing NFs with less workload, valid logic and performance, environmental compatibility, and logic customization.

References

[1] [n. d.]. http://openvswitch.org. ([n. d.]).
[2] [n. d.]. https://developer.nvidia.com/cuda-toolkit/. ([n. d.]).
[3] [n. d.]. https://github.com/gamelinux/prads. ([n. d.]).
[4] [n. d.]. https://github.com/kohler/click/blob/master/conf/thomer-nat.click. ([n. d.]).
[5] [n. d.]. https://github.com/NetFuncDev/nfd. ([n. d.]).
[6] [n. d.]. https://leannfv.org. ([n. d.]).
[7] [n. d.]. https://p4.org/. ([n. d.]).
[8] [n. d.]. https://software.intel.com/en-us/sgx. ([n. d.]).
[9] [n. d.]. https://www.dpdk.org. ([n. d.]).
[10] [n. d.]. https://www.inlab.de/balance.html. ([n. d.]).
[11] [n. d.]. https://www.linux-kvm.org/page/Main_Page. ([n. d.]).
[12] [n. d.]. https://www.ntop.org/products/deep-packet-inspection/ndpi/. ([n. d.]).
[13] [n. d.]. https://www.pfsense.org/. ([n. d.]).
[14] [n. d.]. https://www.snort.org/. ([n. d.]).
[15] [n. d.]. http://www.brendangregg.com/blog/2017-11-29/aws-ec2-virtualization-2017.html. ([n. d.]).
[16] [n. d.]. http://www.haproxy.org. ([n. d.]).
[17] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 113–126.
[18] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 29–43.
[19] Hitesh Ballani, Paolo Costa, Christos Gkantsidis, Matthew P Grosvenor, Thomas Karagiannis, Lazaros Koromilas, and Greg O'Shea. 2015. Enabling end-host network functions. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 493–507.
[20] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10)*. ACM, New York, NY, USA, 267–280. https://doi.org/10.1145/1879141.1879175
[21] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2016. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 511–524. https://doi.org/10.1145/2934872.2934875
[22] Domenico Cotroneo, Luigi De Simone, Antonio Ken Iannillo, Anna Lanzaro, and Roberto Natella. 2015. Dependability evaluation and benchmarking of network function virtualization infrastructures. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 1–9.
[23] Domenico Cotroneo, Luigi De Simone, and Roberto Natella. 2017. Nfv-bench: A dependability benchmark for network function virtualization systems. *IEEE Transactions on Network and Service Management* 14, 4 (2017), 934–948.
[24] Domenico Cotroneo, Roberto Natella, and Stefano Rosiello. 2017. NFV-throttle: An overload control framework for network function virtualization. *IEEE Transactions on Network and Service Management* 14, 4 (2017), 949–963.
[25] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling flow management for high-performance networks. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 254–265.
[26] Mihai Dobrescu and Katerina Argyraki. 2014. Software Dataplane Verification. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 101–114. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/dobrescu
[27] GSNFV ETSI. 2013. Network functions virtualisation (nfv): Architectural framework. *ETsI Gs NFV* 2, 2 (2013), V1.
[28] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. 2016. BUZZ: Testing Context-dependent Policies in Stateful Networks. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 275–289. http://dl.acm.org/citation.cfm?id=2930611.2930630
[29] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. 2014. Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 533–546. http://dl.acm.org/citation.cfm?id=2616448.2616497
[30] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari. 2016. Open Issues in Scheduling Microservices in the Cloud. *IEEE Cloud Computing* 3, 5 (Sept 2016), 81–88. https://doi.org/10.1109/MCC.2016.112
[31] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone
[32] Kai Gao, Taishi Nojima, and Y Richard Yang. 2018. Trident: toward a unified SDN programming framework with automatic updates. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 386–401.
[33] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*.
[34] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: Revitalizing GPU as Packet Processing Accelerator.. In *NSDI*. 83–96.
[35] Juhyeng Han, Seongmin Kim, Jaehyeong Ha, and Dongsu Han. 2017. SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module. In *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 99–105.
[36] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. SoftNIC: A software NIC to augment hardware. *Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155* (2015).
[37] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2011. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 195–206.
[38] Jinho Hwang, K K_ Ramakrishnan, and Timothy Wood. 2015. NetVM: high performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management* 12, 1 (2015), 34–47.
[39] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance contracts for software network functions. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 517–530.
[40] Van Jacobson, Craig Leres, and Steven McCanne. 2005. Tcpdump/libpcap. *http://www.tcpdump.org* (2005).
[41] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. 2012. Kargus: a highly-scalable software-based intrusion detection system. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 317–328.
[42] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. 2017. mos: A reusable networking stack for flow monitoring middleboxes. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*.

113–129.

[43] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level {TCP} Stack for Multicore Systems. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 489–502.

[44] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing.. In *NSDI*. 97–112.

[45] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 171–186. https://www.usenix.org/conference/nsdi18/presentation/katsikas

[46] Georgios P Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q Maguire Jr, and Dejan Kostić. 2016. SNF: synthesizing high performance NFV service chains. *PeerJ Computer Science* 2 (2016), e98.

[47] Georgios P Katsikas, Gerald Q Maguire Jr, and Dejan Kostić. 2017. Profiling and accelerating commodity NFV service chains with SCC. *Journal of Systems and Software* 127 (2017), 12–27.

[48] Junaid Khalid and Aditya Akella. 2019. Correctness and Performance for Stateful Chained Network Functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi19/presentation/khalid

[49] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. 2016. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr.. In *NSDI*. 239–253.

[50] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. 2015. Kinetic: Verifiable dynamic network control. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 59–72.

[51] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. 2015. NBA (Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 22, 14 pages. https://doi.org/10.1145/2741948.2741969

[52] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. *ACM Transactions on Computer Systems (TOCS)* 18, 3 (2000), 263–297.

[53] Sameer G Kulkarni, Guyue Liu, KK Ramakrishnan, Mayutan Arumaithurai, Timothy Wood, and Xiaoming Fu. 2018. REINFORCE: achieving efficient failure resiliency for network function virtualization based services. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. ACM, 41–53.

[54] Sameer G Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, KK Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. 2017. Nfvnice: Dynamic backpressure and scheduling for nfv service chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 71–84.

[55] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 1–14.

[56] Guyue Liu, Yuxin Ren, Mykola Yurchenko, KK Ramakrishnan, and Timothy Wood. 2018. Microboxes: high performance nfv with customizable, asynchronous tcp stacks and dynamic subscriptions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 504–517.

[57] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 459–473.

[58] Christopher Monsanto, Nate Foster, Rob Harrison, and David Walker. 2012. A compiler and run-time system for network programming languages. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 217–230.

[59] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. 2019. Alembic: automated model inference for stateful network functions. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 699–718.

[60] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/2815400.2815423

[61] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV.. In *OSDI*. 203–216.

[62] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Berkeley, CA, USA, 699–718. http://dl.acm.org/citation.cfm?id=3154630.3154687

[63] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. 2018. Automated synthesis of adversarial workloads for network functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 372–385.

[64] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. SafeBricks: Shielding Network Functions in the Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 201–216. https://www.usenix.org/conference/nsdi18/presentation/poddar

[65] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al. 2019. Flowblaze: Stateful packet processing in hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[66] Christofer Price, Sandra Rivera, et al. 2012. Opnfv: An open platform to accelerate nfv. *White Paper* (2012).

[67] OpenStack Foundation Report. [n. d.]. Accelerating NFV Delivery with OpenStack. *white paper* ([n. d.]).

[68] Julius Schulz-Zander, Carlos Mayer, Bogdan Ciobotaru, Raphael Lisicki, Stefan Schmid, and Anja Feldmann. 2017. Unified programmability of virtualized network functions and software-defined wireless networks. *IEEE Transactions on Network and Service Management* 14, 4 (2017), 1046–1060.

[69] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX, San Jose, CA, 323–336. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/sekar

[70] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. 2016. S-nfv: Securing nfv states by using sgx. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM, 45–48.

[71] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: Scalable Symbolic Execution for Modern Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 314–327. https://doi.org/10.1145/2934872.

2934881

[72] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 43–56.

[73] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. Resq: Enabling slos in network function virtualization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 283–297.

[74] Brendan Tschaen, Ying Zhang, Theo Benson, Sujata Banerjee, Jeongkeun Lee, and Joon Myung Kang. 2017. SFC-Checker: Checking the correct forwarding behavior of Service Function chaining. In *Network Function Virtualization and Software Defined Networks*. 134–140.

[75] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. 2014. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 321–332. https://www.usenix.org/conference/atc14/technical-sessions/presentation/vasiliadis

[76] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2011. MIDeA: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 297–308.

[77] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2011. Parallelization and characterization of pattern matching using GPUs. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 216–225.

[78] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 299–312. https://www.usenix.org/conference/nsdi18/presentation/woo

[79] Wenfei Wu, Ying Zhang, and Sujata Banerjee. 2016. Automatic synthesis of NF models by program analysis. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 29–35.

[80] Xiaodong Yi, Jingpu Duan, and Chuan Wu. 2017. GPUNFV: a GPU-Accelerated NFV System. In *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 85–91.

[81] Kai Zhang, Bingsheng He, Jiayu Hu, Zeke Wang, Bei Hua, Jiayi Meng, and Lishan Yang. 2018. G-NET: Effective GPU Sharing in NFV Systems. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 187–200. https://www.usenix.org/conference/nsdi18/presentation/zhang-kai

[82] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phil Lopreiato, Gregoire Todeschi, K. K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proceedings of the ACM SIGCOMM Workshop on Hot topics in Middleboxes and Network Function Virtualization, HotMiddlebox@SIGCOMM 2016, Florianopolis, Brazil, August, 2016*, Dongsu Han and Danny Raz (Eds.). ACM, 26–31. https://doi.org/10.1145/2940147.2940155