NFD: Using Behavior Models to Develop Cross-Platform Network Functions

Hongyi Huang[1], Wenfei Wu[1], Yongchao He[1], Bangwen Deng[1], Ying Zhang[3], Yongqiang Xiong[2], Guo Chen[4], Yong Cui[1], and Peng Cheng[2]
1 Tsinghua University, 2 Microsoft Research, 3 Facebook, 4 Hunan University

IEEE INFOCOM 2021 - IEEE Conference on Computer Communications | 978-1-6654-0325-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/INFOCOM42981.2021.9488734

***Abstract*—NFV ecosystem is flourishing and more and more NF platforms appear, but this makes NF vendors difficult to deliver NFs rapidly to diverse platforms. We propose an NF development framework named NFD for cross-platform NF development. NFD's main idea is to decouple the functional logic from the platform logic —it provides a platform-independent language to program NFs' behavior models, and a compiler with interfaces to develop platform-specific plugins. By enabling a plugin on the compiler, various NF models would be compiled to 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.**

I. Introduction

The ecosystem of network function virtualization (NFV) has gradually matured over the past few years. Network clients propose the requirement of in-network functionalities; network operators would adopt one of the various NF platforms as the runtime environment or management framework for NFs, e.g., Azure VFP [1], OpenNF [2], E2 [3], LeanNFV [4], AWS Nitro [5], etc.; *NF vendors*[1] would provide various software NFs; and integration of the platform and NFs would be established to serve network clients (e.g., cloud tenants or enterprise network users).

However, the diversity in NF platforms and NF logic and the long business cycle of NF software development could slow down the NF developers to deliver NFs. On the one hand, there is a huge number of combinations of environments and NFs — an increasing number of NF platforms are proposed for different reasons, e.g., acceleration (DPDK, SR-IOV, AWS Nitro [5]), security (SGX), scalability (Azure VFP [1], OpenNF [2]), and manageability (E2 [3], LeanNFV [4]); and NFs can be highly customized for different network users, e.g., load balancer with blacklisting, or unbalanced load balancer for heterogeneous backends.

On the other hand, developing or porting an NF to a specific platform involves a non-trivial business cycle — developers need to spend efforts 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.

In this paper, we would explore the possibility to build *an NF development framework which can rapidly build NFs for diverse platforms*. We make an empirical study on several existing NF platforms, and categorize them to two classes — execution environments and management platforms. The first class requires NFs to be developed with a certain piece of logic to be explicitly declared (i.e., programming abstractions), and the second class needs both declared a piece of logic and instrument logic to the NF program structure.

Therefore, we design an NF development framework named NFD. NFD consists of a domain-specific language (DSL) and a compiler. The NFD language is platform-independent and has several built-in programming abstractions (we summarize them from existing frameworks and also add our own abstractions). The compiler backend has interfaces to operate on the NF program syntax tree so that programmer can instrument the program structure. Such a design decouples NF's functional logic and environmental logic — developing m NF models and n platform plugins could achieve mn NF implementations ($m+n<mn$ when $m>1$ and $n>1$).

Scope. NFD is constructed based on the empirical study of existing platforms. It can support these existing platforms, but has no guarantee to support possible platforms in the future. But as long as the platform has the same development requirements to NFs — specific programming abstractions and certain program structures — new platforms can be integrated to NFD following the same methodology in this paper.

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

Wenfei Wu is the corresponding author.

[1]They can be the NF developers in Azure or AWS [1], or traditional device vendors who provide software version, e.g., Palo Alto Networks or Cisco or Juniper, or new NFV startups [4].

Authorized licensed use limited to: Tsinghua University. Downloaded on June 16,2022 at 06:15:54 UTC from IEEE Xplore. Restrictions apply.

development workload.

II. Problem Analysis

We elaborate the empirical study on existing NF platforms and the intuition from them. And then we give the overview of the development framework.

A. *Study of NF Platforms*

NF platforms fall into two categories. Some of them focus on improve NF performance or security in the data plane, and the others focus on the interaction of NFs with the control plane.

1) Execution Environments: A class of NF frameworks provide new execution environments to NFs [5]–[11], and they usually replace a certain piece of logic in NFs with optimized implementation.

Example of accelerating NF I/O. 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 [6]. 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 — replace the `char* pkt` and `pcap_loop()` in libpcap by `struct rte_mbuff` and `rte_eth_rx_burst()` in DPDK.

Example of accelerating pattern match in NFs. 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 [7]–[9]). When applying GPU acceleration, NF developers need to identify the location of the operators[2], build the GPU-based implementation, and conduct the replacement. Similarly as the example above, this replacement needs to be performed on NFs one by one.

Example of securing NF states. 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 [10], [11]. 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. For example, when Han et al. port an IDS to Intel SGX, it brings about 2.5k extra lines of code in the modification [11].

Intuition. We observe that the development (or porting) of NFs for the platforms usually focus on a specific *piece of logic* and implement it in an optimized way. Thus, the intuition to build NFs universal to these platforms is to use high-level *programming abstractions* to replace the code in the programs, and use the compiler to link the programming abstractions to platform-dependent implementation at link time.

[2]"Operator" (alone) means a language element that operates on operands; "network operator" means a role in production networks who manages the network infrastructure.

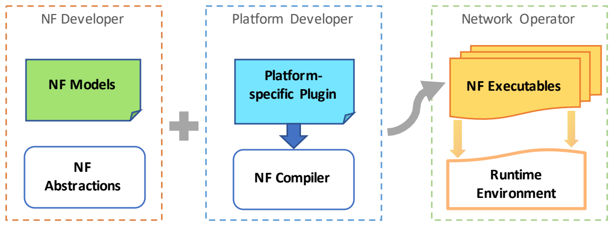


Fig. 1: NFD architecture and workflow

2) Management Platforms: The second class of NF platforms [1]–[4]integrate NFs with the control plane from better management. They still operate on a certain piece of logic, but they further need to instrument NFs to achieve the interaction.

Example of integrating NFs with state management framework. 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 [2], [12], modifying PRADS and Snort takes more than 100 man-hours respectively.

Intuition. We observe that these platforms not only operate on a piece of logic in NFs but also need to instrument the program to interact with that logic in the NF program *structure*. Thus, the intuition to build NFs for these platforms is to further build a compiler backend, which can traverse the NF program structure and apply changes to all NFs.

B. *Solution Overview*

Towards the goal of building rapid development framework, we take two techniques to build our solution — domain specific language and program compilation time optimization (more precisely, it should be transformation).

We propose an NFD language to program NFs. The language first contains common language elements such as basic types, expressions, statements, and control flows in high-level language (e.g., C/C++). More importantly, it declares some certain elements as programming abstractions. The programming abstractions are summarized from the individual NF porting cases [2], [11], current NF development frameworks (e.g., Netbricks [13]), and some network management solutions (e.g., NF placement, verification). NF developers use the language to write NF (behavior) models.

As Fig.1 shows, NFD has a compiler to translate the model to an executable and integrate platform specific features. The compilation process first translate an NF model to a syntax tree (compiler frontend parser), and then traverse the syntax tree to generate runnable code (compiler backend). NFD compiler's backend would generate C/C++ code in Linux in preliminary, and also provides interfaces of (1) the syntax tree of the NF model and (2) a tree traversal API, which can overwrite the tree-to-code translate or instrument other logic. Thus, a platform developer could use the interfaces to build platform "plugins". By combining an NF model and a platform plugin,

Authorized licensed use limited to: Tsinghua University. Downloaded on June 16,2022 at 06:15:54 UTC from IEEE Xplore. Restrictions apply.

NFD would generate executable that both has the NF model functionality and adapts to the platform.

III. NFD Language and NF Models

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

A. NF Modeling Language

General Program Elements			
const	c	$::=$	$(0\|1)^+$
variable	var		
expression	e	$::=$	$c\|var\|e\|Expr\_Op(e_1, e_2, ...)$
condition	x	$::=$	$Rel\_Op(e, ..)\|Log\_Op(x, ..)\|(x)$
model	$model$	$::=$	$stmts$
statements	$stmts$	$::=$	$stmt\|stmt; stmts$
statement	$stmt$	$::=$	$var = e\|if\|loop$
if statement	if	$::=$	if $(x)\{(stmts\}$ else $\{stmts\}$
loop statement	$loop$	$::=$	while(x) then $\{stmts\}$
NFD Specific Extension			
header field	h	$::=$	$sip\|dip\|sport\|dport\|proto\|...$
state	st	$::=$	declare state s

Fig. 2: NFD language for NF models

Figure 2 shows the NFD language syntax. Note that this language is a summary of several existing solutions [14]–[17] — it could equivalently express the same semantics as existing solutions, but is also enriched with several new abstractions (§ III-B). Alternative language designs are also acceptable as long as they can express NF logic.

NFD language contains basic language elements in general high-level programming languages (e.g., C++, Rust), including basic types, expressions, statements, and control flows[3], so that the language can be *semantically complete* to express all existing NFs developed in high-level languages. The semantics of these elements are the same as that in the high-level language, and we omit it here for space. A formal definition of the language semantics is in [18].

We further introduce a few NF specific extension. The extension does not change the program language syntax, but it explicitly declares some elements as NF specific logic. We reserve a few keywords (sip, dip, etc.) to represent packet header fields, they refer to the current packet at runtime. We explicitly declare NF state as "state variable".

We further define a few derived symbols and notations in Table I to simplify the 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 §III-C), the operator would look up or modify the map structure [19]; (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.

[3]The $Expr\_Op(e_1, e_2, ...)$ operator is an abbreviation of various operators: arithmetic $(+, -, \times, /, ++, --)$, set $(\cap, \cup, \setminus)$, index ([]) , and user-defined ($Encrypt$, $Hash$, NAT) operators. $Rel\_Op(e, ...)$ stands for relations: equality($=, \neq$), scalar($>, <$), set $(\subset, \in)$, and user-defined ($PatternMatch$) relation operators. $Log\_Op(x, ...)$ stands for logical operators: and, or, not.

TABLE I: Derived symbols in NFD language

symbols	meaning
$f[h]$	h is a header field (Figure 2 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 [19], 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: $h_1 = v_1 \wedge h_2 = v_2 \wedge ...$
$f \sqsubseteq r$	Abbr. for a flow-rule match: $f[h_1] = v_1 \wedge f[h_2] = v_2 \wedge ...$
R	Abbreviation for a rule set: $\{r_1, r_2, ...\}$
$f \sqsubseteq R$	Abbreviation for a flow-ruleset match (f match one of rules in R): $f \sqsubseteq r_1 \vee f \sqsubseteq r_2 \vee ...$

B. Representing NF Programming Abstractions

The syntax and its semantics can express common programming abstractions in NF programming frameworks [3], [13], [15]–[17], [20]–[23], including packet parsing ($f[h]$), deparsing (f), and transformation ($f[h] = e$), drop ($f[output] = \epsilon$), and filter ($f \sqsubseteq R$).

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)`. They are in the class of "Expr_Op". In NFD, we implement "Encrypt" (encrypting a byte stream), "PatternMatch" (searching a pattern in a byte stream), "hash"(in hash-based load balancer), and "NAT" (a non-replacement IP sampling algorithm for IP translation). We also create two long-neglected abstractions in NFD— state abstraction and time-driven logic abstraction.

State abstraction. 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.

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

Authorized licensed use limited to: Tsinghua University. Downloaded on June 16,2022 at 06:15:54 UTC from IEEE Xplore. Restrictions apply.

```
class State_Counter{
  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]++;
} };
```

Fig. 3: The counter state class of a per-flow monitor: member variables and an overridden operator

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 3 shows the implementation of the state class in the per-flow monitor example including attributes, instances, and an overridden operator $++$.

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.

Advantage. Using programming abstractions instead of self-development has two benefits. (1) It helps the developer to reuse the code and avoid making mistakes. For example, using the state abstraction above can save development effort to rebuild it, and avoid mistakenly declaring a state in Figure 3 as a single variable). (2) It helps NFD to locate the "platform specific target piece of logic" in the future platform integration (§ V). For example, the state abstraction helps to find the state and integrate to state management system [2].

C. A Model and Use Cases

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

Fig. 4: SMAT syntax

We show a few cases where NFs are developed using NFD language. As discussed a few existing works [24] and exercised by industry [1], a wide range of NFs can be implemented as a stateful match-action table (SMAT), whose structure can be expressed as Fig. 4 in NFD. We visualize the programs in tables for a better view and space limitation. SMAT's semantics is that each entry first match flows and states, and if the match result is true, the action is taken and stop, otherwise, proceed to the next row (i.e., the first match applies).

Figure 5 shows the example of a stateful Firewall, a stateful NAT, and a load balancer that stores the consistent mapping of a flow to a backend server. In addition, we design a rate limiter

	Match		Action	
	Flow	**State**	**Flow**	**State**
Stateful Firewall	**Configuration:** OK={r1, r2, ...}			
	f$\sqsubseteq$OK	-	f[output]:=IFACE	seen:=seen$\cup${f}
	f	f$\in$seen	f[output]:=IFACE	-
	f$\not\sqsubseteq$ OK	f$\notin$seen	f[output]:=ϵ	-
NAT	f	f$\notin$map	f:=NAT(f); f[output]:=IFACE	map[f]:=NAT(f)
	f	f$\in$map	f:=map[f]; f[output]:=IFACE	-
LB	**Configuration:** mode = ROUND_ROBIN			
	f	f$\in$ map	f[dip]:=srv[map[f]]	-
	f	*	f[dip]:=srv[idx]	map[f]:=idx; idx:=(idx+1)%N

Fig. 5: 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]$\leq$tkn	f[output]:=IFACE	tkn:=tkn-f[size]
*	f[size]$>$tkn	f[output]:=ϵ	-

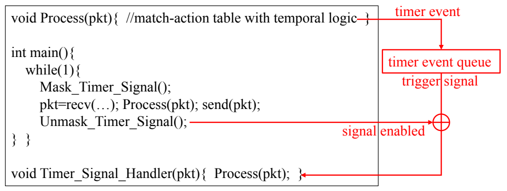
Fig. 6: The model of a rate limiter

to validate the *timer* operator— a rate limiter in Figure 6. 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.

IV. NF Compilation

NFD compiles NF models to NF programs and also provides the syntax tree of the NF model.

NF Code Generation. NFD compiles an NF model to a C++ NF program by the following transformation. (1) Most basic elements (e.g., control flows, expressions, predicates, and policies) in NFD language can be implemented in C++ directly. (2) States are declared and initialized as global variables at the beginning of the program. (3) Time-driven logic is incorporated as Fig. 7 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. (4) 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);  }
```

Fig. 7: NF program structure

Authorized licensed use limited to: Tsinghua University. Downloaded on June 16,2022 at 06:15:54 UTC from IEEE Xplore. Restrictions apply.

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., "PatternMatch", "Encrypt"), "flow" class with "[]" as in § III-A and "state" class as in § III-B. In the final compilation from a C++ program to a binary executable, they would be linked together.

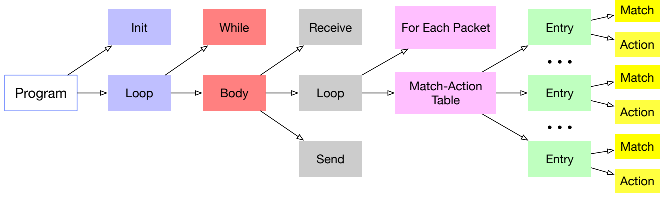


Fig. 8: A part of the syntax tree of a SMAT

NF Syntax Tree. An NF model built on the NFD language syntax would follow a tree structure: deriving a program to composing sections (i.e., I/O, initialization, model), deriving each section to statements (i.e., match and action), and deriving each statement to basic symbols (variables, constants, and variables).

For example, Fig. 8 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.

V. NF-Environment Integration

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

A. Programming Interfaces

The programmable interfaces for platform integration are a tree traversal function and per-symbol callback functions. (1) According to the NFD syntax, NFD compiler would generate one callback function for each symbol. For example, it would generate a `visitSMAT()` for the symbol SMAT and a `visitEntry()` for Entry in Fig.8. These callback functions are initially empty, but can be overridden by programmer to add their logic. (2) NFD provide a tree-traversal function for syntax trees. The function is input with a syntax tree, it traverses the tree with a depth-first-search (DFS) order. When visiting each node on the syntax tree, the function would check the type of the node (e.g., SMAT or Entry or Init) and invoke the corresponding callback function.

Note that the preliminary compilation in § IV is also implemented by this interface. The preliminary compiler traverses the NF model's syntax tree and translate each node to corresponding C/C++ implementation. Integrating an NF with new platform needs the programmer to inherit the preliminary

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

Fig. 9: Code of the OpenNF plugin

```
public static void main(...){
    new SGXVisitor.visit(syntax_tree);}
public class SGXVisitor implements NFDCompiler{
  List<String> sensitiveFunc;
  List<String> sensitiveData;
  @Override public T visitStateDeclaration(...){
    stateName = ...
    sensitiveData.add(stateName);}
  @Override public T visitStateMatch(...){
    FuncName = ...
    sensitiveFunc.add(FuncName);}
  @Override public T visitStateAction(...){
    FuncName = ...
    sensitiveFunc.add(FuncName);}
}
```

Fig. 10: Code of the SGX plugin

compiler and override the per-symbol callback function. Programmers can add new logic in the callback function or even replace the original one. We call the class that inheriting the compiler a "platform plugin".

B. Use Cases

We walk through the examples in §II to show how NFD performs the integration.

Example of IO acceleration with DPDK. We first identify the abstraction of IO is "Receive" symbol. Then we inherit the preliminary compiler and create a DPDK platform plugin named `DPDKVisitor`. In the plugin we override the callback function named `visitReceive()`. In the callback function `visitReceive()`, we do not call the super class (i.e. the compiler), but add DPDK implementation (`rte_eth_rx_burst()`). In other callback functions, we just call the super class's method. We execute the `DPDKVisitor.visit(syntax_tree)` to traverse the syntax tree again to generate new programs; the IO logic is replaced and others are not changed.

Example of GPU acceleration. Similarly to the DPDK acceleration, GPU acceleration is to override the operator `PatternMatch()`. It use an GPU implementation to replace the CPU one.

Example of integrating with OpenNF. Integrating to OpenNF is a bit complicated, but the workflow is the same. Each NF needs to make three modifications: (1) adding the agent code which starts the agent thread in the initialization, (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).

Authorized licensed use limited to: Tsinghua University. Downloaded on June 16,2022 at 06:15:54 UTC from IEEE Xplore. Restrictions apply.

We build an OpenNF plugin for (1) and (2), and build an external library for (3). Fig. 9 shows part of plugin. The plugin override `visitInit()` and add the logic to start the OpenNF agent and declare `allStates` variable (line 6); it then overrides the `visitStateDeclaration()`, where the name of each state varialbe is added to `allStates`. By calling `OpenNFVisitor.visit(syntax_tree)`, the code about (1) and (2) are instrumented to the final NF code.

In the external libaray, when get/put/delete a `flow` is called, the "List of all states" is iterated, each state would use `[flow]` to operate on the corresponding state instance. NFD links the external library with the OpenNF plugin generated code, and achieves an executable integrated with OpenNF.

Example of adding SGX protection. SGX protection needs to find out all NF state variables and state related functions, and seal them in a specially protected memory region. Fig. 10 shows the plugin for this. It overrides the `visitStateDeclaration()`, `visitStateMatch()`, and `visitStateAction()`. In each overriding function, the plugin collect the variable names and function names. Finally, the plugin outputs the list of state variables and functions. NFD use the list to generate an SGX configuration and compile the code with configuration, and outputs an SGX-enhanced executable.

VI. Implementation

TABLE II: 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++)

NFD Implementation. We write the syntax of NFD language in g4, and use Java Antlr4 to build the NFD compiler frontend (i.e., the parser) and the platform integration interfaces (syntax tree traversal function and callback functions). Then we implemented the preliminary compiler and plugins for different platforms. The lines of code of some components are listed in Table II.

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 [25] 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 environmental plugins (OpenNF, Intel SGX, GPU, DPDK), the NFD compiler has arguments to decide whether to add the plugins to NF programs.[4]

NF Development. 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 Fig. 5 are used for representing results in this section. A complete list and testing results are in [18]. 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 [26]–[30].

Experiment Settings. 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 [31] 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) [32]; (3) **VM-1host**: the NF is wrapped up by a VM (using KVM [33]), 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.

VII. Evaluation

We show that NFD can save development workload, its NFs are functionally valid, the integration of NFs with platforms works correctly, and NFD can be used to develop complex NFs that equivalent to commodity ones.

A. 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 $O(n+m)$.

We use the lines of code (LoC) to quantify the development workload. As in TABLE II, 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

[4]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 releases a SGX compiler supporting C++ STL.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 16,2022 at 06:15:54 UTC from IEEE Xplore. Restrictions apply.

of 14 NFs and 4 platforms cost totally about 700k 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 (700k v.s. 1877).

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 § VII-C). For one of the authors, 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 with the SGX expert). While using NFD, the graduate student spent one hour writing a network monitor SMAT model and less than one day building a SGX plugin (following the requirement from SGX), which only takes 8 man-hours for the student. Last but not least, that SGX plugin can be applied to any NF SMAT model in the future. NFD shows good potential to improve productivity.

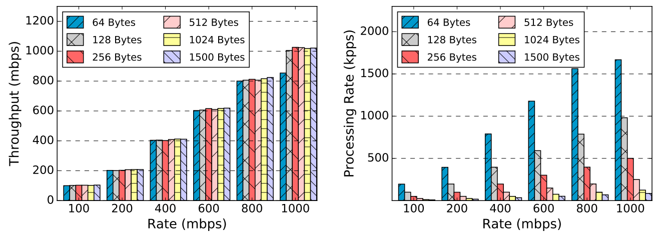


Fig. 11: NFD rate limiter performance

B. Individual NF Validation

We show that NFD outputs logically correct NFs.

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 [18]), 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 Fig. 11. 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 exceed 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.

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. Fig. 12 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. Hence, our firewall has better performance. [5] (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 Fig. 13. The experiments are under Unit-1host (using the socket for inter-process 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 execute 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 [18]. Unit performance tests show that NFD-based NFs have acceptable performance, and in a lot of cases they can be viewed as micro-services 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). Fig. 14 shows the throughput when these NFs use DPDK or libpcap. We observe that libpcap

[5]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.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 16,2022 at 06:15:54 UTC from IEEE Xplore. Restrictions apply.

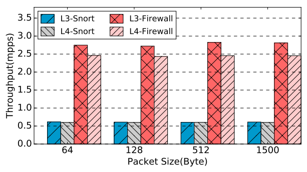


Fig. 12: Unit test of NFD FW v.s. Snort

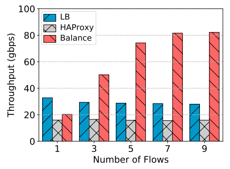


Fig. 13: Load balancers

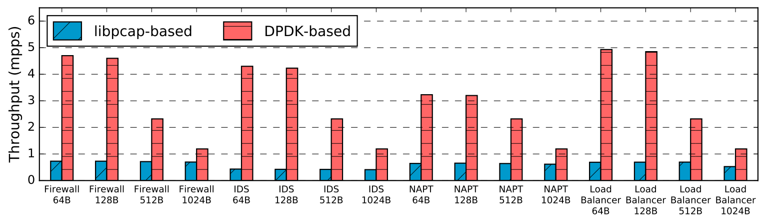
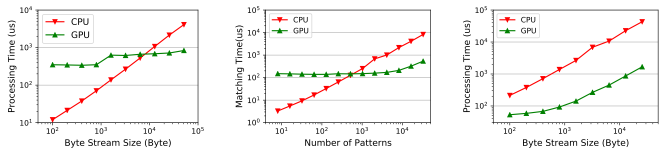


Fig. 14: Performance of 4 NFs, tuning packet size



(a) Encryption scaling up byte stream size (b) Pattern matching scaling up number of patterns (c) Pattern matching scaling up byte stream size

Fig. 15: GPU acceleration (Unit-1host)

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.

C. NF-Platform Integration

We show the use cases of integrating NFs with environments, and measure their performance.

Accelerate processing with GPU. Atop CUDA Toolkit [34], the two operators — encryption and pattern matching — are integrated with bytestream NFs. The performance result is in Fig. 15. 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 Fig. 15a, GPU is slower in encrypting <6KB bytestream, but faster in large bytestreams; because the encryption chunks a bytestream and encrypts blocks in parallel. Similarly (Fig. 15b), GPU could match >5K patterns at a faster speed than CPU, but slower for <5k patterns, because these patterns are matched in parallel. GPU operators perform worse than CPU when preparation time dominates during the process.

Alternative packet I/O using DPDK. NFD provisions NFs with different packet I/O drivers (DPDK and Libpcap) and deploys them in the path of two end-hosts. Fig. 16 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 Fig.10 in [2]) and repeat the experiment. We observe NFD-based Firewall successfully interacts with the OpenNF controller, and the experiment result is in [18]. We draw the similar conclusion as in OpenNF [2]: (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 Fig. 17. We observe that NFD NFs can achieve 1mpps in SGX environment, which is acceptable. But compared with the same setting without SGX, where the throughput is usually >10 mpps[6], we conclude that SGX environment is the bottleneck.

D. Case Study: Complex NF Development

Replace NF chains by consolidating them. In current NFV systems, NFs are fixed and chained to get complex functionality. NFD could provide an alternative solution — consolidating NF models and generate one executable.

We use the example in § I, where a network client needs a load balancer with blacklisting. This can either be implemented by chaining a firewall and a load balancer (denoted as "FW→LB") or by consolidating a firewall model with a load balancer model using NFD (denoted as "FW+LB"). Fig. 18 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 for NF chaining.

Build a complex NF equivalent to a commodity NF. pfSense [35] from Netgate has now become a prevalent NF for in-network security. It embraces several core features in the data plane, including firewall, NAT, LB, and rate limiter[7]. We make an equivalent implementation in NFD by concatenating

[6]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.

[7]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.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 16,2022 at 06:15:54 UTC from IEEE Xplore. Restrictions apply.

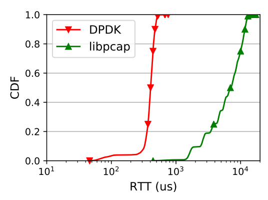


Fig. 16: Packet I/O: DPDK versus libpcap (VM-1host)

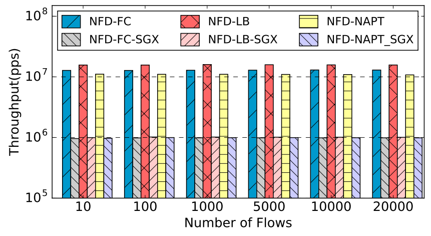


Fig. 17: NF performance with and without SGX

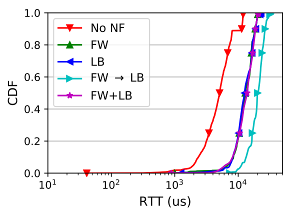


Fig. 18: RTT of NF chains (VM-1host)

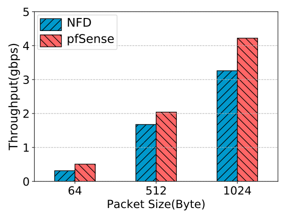


Fig. 19: NF performance compared with pfSense

the SMATs of these four NF models and compiling the merged model to a synthesized NF program.

Evaluation shows that the NFD-based NF has equivalent logic compared with pfSense, but a small performance degradation (Fig. 19, e.g., 1.68 v.s. 2.08 gbps when packet size is 512B). The degradation is caused by the non-optimal (but universal) data structure and redundant parsing in NFD, which is the tradeoff between the agile development and performance and can possibly be improved using compilation optimization techniques.

VIII. Discussion and Related Work

Design choice of NFD Language. NFD "summarizes" the programming abstractions from existing development and modeling frameworks (with two own new abstractions). The language is interchangeable with most existing languages, e.g., SNAP [17], VFP [1], P4 [36]. As long as the the compiler of other frameworks could provide the same syntax tree modification interfaces, the methodology in NFD can be applied similarly.

Deployment progress. NFD is currently anonymously open sourced in [18]. Its model-based NFs have recently be released on OpenNetVM [37].

NF development frameworks Most recent NF development frameworks target one environment or platform, and summarize NF programming abstractions, such as packet parsing, filtering, transformation [12], [13], and NFD complements the part for cross-platform integration. Modular NF development [21], [38] eases the composition of NFs, but they do not target cross-platform deployment, because intra-module logic are still ad hoc and not designed to be platform independent.

NF management frameworks Traditional NF management frameworks [3], [37], [39]–[41] 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], [6], [7], [9]–[11], [34], [42], [43], 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.

NF Frameworks for other purposes. A few recent NFV frameworks are proposed for various requirements [44], [45], into which NFD would be a great help to port NFs. SNF [38] proposed DAG-based NF chains can be synthesized to eliminate cross-NF redundancy, where NFD NF models can contribute to the synthesizing. Like DPDK, other IO acceleration solutions (e.g., mTCP [46]) can also override NFD I/O. CHC requires NF states to be identified for integration [47]; Metron [41] requires to anatomize NFs and offloads stateless part to hardware; VFP and Eden [1], [22] also implement NFs on both software and hardware. For these frameworks, NFD would be beneficial by automating NF program analysis and porting using its compiler plugin.

Other Inspirations. NFD is inspired by several works. (1) Packet processing operators (e.g., "resubmit") are also used in OVS and P4 [32], [36]. (2) Devoflow [48] 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 [49] and stateful table [24], and also NF modeling language [17]).

IX. Conclusion

We built 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. Our evaluation demonstrates NFD's feasibility by developing NFs with less workload, valid logic and performance, platform compatibility, and commodity-equivalent complex logic.

Acknowledgment

This project is supported by National Natural Science Foundation of China Grant No. 61802225 and the Collaborative Research Award from Microsoft Research Asia.

References

[1] D. Firestone, "VFP: A virtual switch platform for host SDN in the public cloud," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 315–328. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/firestone

[2] 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*, ser. SIGCOMM '14, 2014.

[3] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A framework for nfv applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 121–136. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815423

[4] "https://leannfv.org."

Authorized licensed use limited to: Tsinghua University. Downloaded on June 16,2022 at 06:15:54 UTC from IEEE Xplore. Restrictions apply.

[5] "http://www.brendangregg.com/blog/2017-11-29/aws-ec2-virtualization-2017.html."

[6] "https://www.dpdk.org."

[7] 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: USENIX Association, 2014, pp. 321–332. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/vasiliadis

[8] X. Yi, J. Duan, and C. Wu, "Gpunfv: a gpu-accelerated nfv system," in *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 2017, pp. 85–91.

[9] 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: USENIX Association, 2018, pp. 187–200. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/zhang-kai

[10] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, "Safebricks: Shielding network functions in the cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 201–216. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/poddar

[11] J. Han, S. Kim, J. Ha, and D. Han, "Sgx-box: Enabling visibility on encrypted traffic using a secure middlebox module," in *Proceedings of the First Asia-Pacific Workshop on Networking*. ACM, 2017, pp. 99–105.

[12] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for nfv: Simplifying middlebox modifications using statealyzr." in *NSDI*, 2016, pp. 239–253.

[13] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of nfv." in *OSDI*, 2016, pp. 203–216.

[14] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 1–13.

[15] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," in *ACM SIGPLAN Notices*, vol. 49, no. 1. ACM, 2014, pp. 113–126.

[16] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *ACM SIGPLAN Notices*, vol. 47, no. 1. ACM, 2012, pp. 217–230.

[17] 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 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 29–43.

[18] "https://github.com/netfuncdev/nfd."

[19] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "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*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 533–546. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616497

[20] 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*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 511–524. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934875

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.

[22] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling end-host network functions," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 493–507.

[23] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2014, pp. 459–473.

[24] 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*. ACM, 2016, pp. 29–35.

[25] V. Jacobson, C. Leres, and S. McCanne, "Tcpdump/libpcap," *http://www.tcpdump.org*, 2005.

[26] "https://www.snort.org/."

[27] "https://github.com/gamelinux/prads."

[28] "https://www.inlab.de/balance.html."

[29] "http://www.haproxy.org."

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

[31] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879175

[32] "http://openvswitch.org."

[33] "https://www.linux-kvm.org/page/main_page."

[34] "https://developer.nvidia.com/cuda-toolkit/."

[35] "https://www.pfsense.org/."

[36] "https://p4.org/."

[37] W. Zhang, G. Liu, W. Zhang, N. Shah, P. Lopreiato, G. Todeschi, K. K. Ramakrishnan, and T. Wood, "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*, D. Han and D. Raz, Eds. ACM, 2016, pp. 26–31. [Online]. Available: http://doi.acm.org/10.1145/2940147.2940155

[38] 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.

[39] O. F. Report, "Accelerating nfv delivery with openstack," *white paper*.

[40] J. Hwang, K. K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[41] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. M. Jr., "Metron: NFV service chains at the true speed of the underlying hardware," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 171–186. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/katsikas

[42] "https://software.intel.com/en-us/sgx."

[43] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "Softnic: A software nic to augment hardware," *Dept. EECS, Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155*, 2015.

[44] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing." in *NSDI*, 2017, pp. 97–112.

[45] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda *et al.*, "Flowblaze: Stateful packet processing in hardware," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[46] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level {TCP} stack for multicore systems," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 489–502.

[47] J. Khalid and A. Akella, "Correctness and performance for stateful chained network functions," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/khalid

[48] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: Scaling flow management for high-performance networks," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 254–265, 2011.

[49] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J. M. Kang, "Sfc-checker: Checking the correct forwarding behavior of service function chaining," in *Network Function Virtualization and Software Defined Networks*, 2017, pp. 134–140.

Authorized licensed use limited to: Tsinghua University. Downloaded on June 16,2022 at 06:15:54 UTC from IEEE Xplore. Restrictions apply.