LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed

Hao Li and Yihan Dang, *Xi'an Jiaotong University;* Guangda Sun, *Xi'an Jiaotong University and National University of Singapore;* Guyue Liu, *New York University Shanghai;* Danfeng Shan and Peng Zhang, *Xi'an Jiaotong University*

https://www.usenix.org/conference/nsdi23/presentation/li-hao

This paper is included in the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed

Hao Li[1], Yihan Dang[1], Guangda Sun[1,2], Guyue Liu[3], Danfeng Shan[1], Peng Zhang[1]

[1]*Xi'an Jiaotong University* [2]*National University of Singapore* [3]*New York University Shanghai*

Abstract

NFV has entered into a new era that heterogeneous frameworks coexist. NFs built upon those frameworks are thus not interoperable, obstructing operators from getting the best of breed. Traditional interoperation solutions either incur large overhead, *e.g.*, virtualizing NFs into containers, or require huge code modification, *e.g.*, rewriting NFs with specific abstractions. We present LemonNFV, a novel NFV framework that can consolidate heterogeneous NFs without code modification. LemonNFV loads NFs into a single process down to the binary level, schedules them using an intercepted I/O, and isolates them with the help of a restricted memory allocator. Experiments show that LemonNFV can consolidate 5 complex NFs without modifying the native code while achieving comparable performance to the ideal and state-of-the-art pure consolidation approaches with only 0.7–4.3% overhead.

1 Introduction

The past decade has witnessed the flourish of Network Function Virtualization (NFV) research, with the goal of replacing hardware middleboxes with software network functions (NFs) running on commodity servers. Prior research efforts have led to a plethora of NFV frameworks focusing on various aspects, including performance optimization [30, 33, 35, 49], programming models [34, 41], resource management [51, 60], and more recently security [43, 52, 54]. Since there are no conventional and widely-adopted interfaces, these NFV frameworks are unsurprisingly implemented in *heterogeneous* ways, with different libraries (*e.g.*, DPDK [5], netmap [57]), languages (*e.g.*, C, C++, Rust), and abstractions (*e.g.*, Click element [37], BESS module [30]).

NFs built upon these heterogeneous NFV frameworks are not *interoperable*, which exposes two hard choices for users in actual deployment. The first choice is asking operators to learn, deploy, and maintain multiple frameworks to serve different purposes. The second choice is picking one framework and asking the developers to add extra functionalities by reinventing the wheel. The former choice dramatically increases the cost of operation, and the latter choice requires substantial engineering effort and is error-prone. This reality raises a timely and important question: *can heterogeneous NFs interoperate without modifying the code?*

To answer this question, the first natural candidate solution is using virtualization. Under this model, each NF runs in a virtual machine or a container on one core, and packets are steered across cores to chain multiple NFs. While this approach hides the heterogeneity of NFs under standardized virtualization interfaces, it incurs prohibitive performance overhead when chaining multiple NFs [33, 49]. The overhead stems from three main sources: virtualized components, cache misses, and context switches (details in §2.1). Despite the efforts of various lightweight virtualization techniques [39, 56, 68], these overheads cannot be fully eliminated. As a result, this virtualization approach fundamentally cannot achieve the line speed, one of the key requirements of NFV deployment.

The second candidate solution is using consolidation. This model implements NFs as software modules and runs them in the *same* process. NFs are chained through function calls and scheduled in a *run-to-completion (RTC)* mode, *i.e.*, once a packet is received by an NF, the NF continues processing it until finishes. By eliminating cache miss and context switch overheads, this approach ensures the line-rate processing and has been adopted by existing high-performance NFV frameworks such as Metron [35] and BESS [30]. While consolidation works well for NFs under the same framework, it seems *unlikely* to be applied to heterogeneous NFs without modifying the code due to three key challenges across loading, scheduling, and memory management.

- *How to launch heterogeneous NFs in one process?* Launching NFs written with different frameworks and languages into the same process could result in various conflicts (*e.g.*, dependencies, functions, variables). A potential solution that manually modifying the code to resolve each conflict could be tedious.
- *How to chain multiple NFs with separate control flows within a process?* Each NF has its own control flow

which often includes an infinite loop of packet processing. Chaining these separate control flows requires locating new entry points and correctly schedule them to process packets.

- *How to isolate memory of multiple NFs within a process?* NFs running in the same process share the same stack and heap, without memory isolation. This brings security concerns when chaining multiple NFs from different and possibly untrusted vendors. A previous solution that rewrites NFs with safe programming languages would incur large porting efforts.

In this work, we propose *LemonNFV*, an NFV framework that enables consolidating heterogeneous NFs without modifying the code. To address the above challenges, LemonNFV provides three abstractions for NFs: the compiled binary, the scheduling entry points, and the memory allocation interfaces. The key point of these abstractions is that they (1) are essential and sufficient for making NFs interoperable, and (2) can be easily adopted to NFs without much coding efforts by capturing the natural homogeneity that already exists in all NF implementations. By transforming NFs into such *LEast Modified network functiONs (LEMONs)*, LemonNFV then implements a set of utilities upon those abstractions that load, execute and manage NFs inside a process. Concretely, we make the following contributions in designing LemonNFV.

Loading via LEMON Binary (§4.1 and §4.4). We view each NF as a software module and leverage the standard binary format to load them. The binary hides the complexity of NFs written in different languages and/or with conflicting names. To further enable dynamic chaining and NF migration, we build a *LEMON loader* to specify memory layout and resolve dependency conflicts, both of which are not supported by existing loaders (*e.g.*, `ld.so`).

Chaining via Schedulable I/O (§4.2). We observe that packet I/O could be an ideal scheduling point where an NF's processing logic starts and ends. Based on this observation, we provide unified I/O interfaces as entry points for inter-NF scheduling. These interfaces can replace existing NFs' I/O interfaces, which are usually built on top of several common packet I/O libraries, such as DPDK, `libpcap` and netmap. To chain multiple NFs inside one process, we provide a *scheduler* to correctly switch between different NF control flows.

Isolation via Restricted Memory Interfaces (§4.3). We use a *restricted allocator* to set explicit *NF boundaries* by creating private stack and heap for each NF, which can replace the native memory allocation interfaces like `malloc` and its variants. To isolate different NFs in the process, we implement an *isolator* that leverages hardware-aided technique (Intel PKU) to realize efficient intra-process memory isolation [1].

We evaluate LemonNFV with real NFs and traffic (§6). The results show that LemonNFV can (1) consolidate 5 complex NFs without code modification – even they are implemented with different frameworks and/or languages; (2) realize comparable performance to the ideal and state-of-the-art consolidation approaches with only 0.7–4.3% overhead of isolation. *Ethics:* This work does not raise any ethical issues.

2 Motivation

In this section we explain why neither virtualization (§2.1) nor existing consolidation techniques (§2.2) can address the need of heterogeneous NF interoperation.

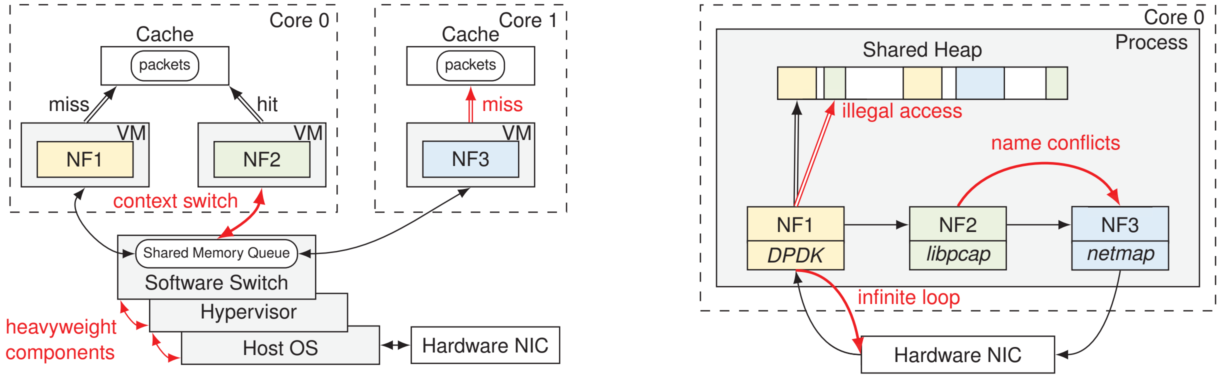
2.1 Virtualization is Slow

Figure 1a shows the three types of performance overhead incurred by virtualization approaches. *(V1) Heavyweight components:* Packets may need to go through all levels of virtualization components: host OS, VM hypervisor, guest OS and software switch. Each of the components brings a non-negligible overhead since they may run protocol stacks and perform queueing. *(V2) Cache misses:* If the NF instances are deployed on separate cores, passing packets across cores will be inevitable, in which case accessing each packet will become LLC or DRAM bounded. Moreover, passing packets across cores is often achieved by software switches, requiring frequent enqueuing and dequeuing when the NFs send and receive packets. *(V3) Context switches:* Scheduling will happen if there are multiple NF instances pinned to a single physical core, which would result in extra switching overhead.

Researchers have been leveraging the emerging lightweight virtualization techniques to improve the performance of virtualization NFV systems. By reducing full-fledged VMs into more lightweight environments, such as containers [21, 61, 63, 68], unikernels [39, 46, 69], and even processes [40, 42, 56, 76], the virtualization overhead, *i.e.*, *V1*, is greatly reduced or eliminated, yet still preserving *V2* and *V3*.

In fact, running NFs as separate instances makes it impossible to reduce *V2* and *V3* at the same time. Pinning instances to dedicated cores (*e.g.*, OpenNetVM [76], NFP [64]) eliminates the overhead of context switches (-*V3*), but forces packets to be delivered over shared memory and results in L1/L2 cache misses (+*V2*). On the other hand, compacting instances on a single core (*e.g.*, Quadrant [68], EdgeOS [56]) would reduce the cache misses on packets (-*V2*), while frequent context switches can lead to much more system calls and TLB misses (+*V3*). Therefore, we reach a conclusion that virtualization-based NFV frameworks can hardly meet line-rate (*i.e.*, 100/400Gbps) processing requirements because of its inherent overhead.

[1] Currently, our isolation model focuses on memory isolation only and does not support control flow integrity, state sharing and packet isolation as in related works [31, 52, 66]. See §4.3 and §7 for details.



(a) A virtualization approach that runs NFs as separate instances. Red arrows indicate the major source of overhead, and grey boxes are components that can be optimized out.

(b) A consolidation approach that executes NFs in a single process. Black arrows are the ideal workflow while the red ones signify the obstacles when the NFs are heterogeneous.

Figure 1: The virtualization approach is slow, while the consolidation cannot handle the conflicts between heterogeneous NFs.

2.2 Consolidating Heterogeneous NFs is Hard

Consolidation, on the other hand, avoids all the extra overhead from the virtualization approaches by eliminating the boundaries between NFs [17, 30, 35, 52, 61]. Under this model, the SFC is deployed in one process, *i.e.*, no *V1*, and NF instances are executed in an RTC manner *i.e.*, no *V2* and *V3*, as shown in Figure 1b. However, it introduces even more challenges when trying to consolidate heterogeneous NFs.

Challenge 1: NFs cannot be loaded. NFs are independently developed with different frameworks and abstractions. As such, when putting them together in the same process, they may conflict with each other in terms of dependencies, functions and variables. For example, two NFs may define global data structures with the same name, while simply linking their source code would raise a multi-definition error. Things get more complex if NFs are written in different languages.

Naive solution: A naive solution for loading NFs into a single process includes three time-consuming tasks. First, operators have to check all symbols exposed by NFs to locate the conflicts. Second, they should manually resolve the conflicting symbols, which however is not always a feasible task, considering the conflicts that may happen between closed-source libraries. Third, they need to reconstruct other code with the resolved name, which usually requires deep understanding of the whole NF codebase. Despite the above tedious efforts, manual resolution can never work for the cross-language NFs or dynamic SFC updating without halt.

Challenge 2: NFs cannot be scheduled. The workflow of each NF is driven by *an infinite loop* of receiving and sending packets, *i.e.*, processing packets after receiving them, and receiving more after sending the processed ones. As a result, an NF will take up a core forever once it starts running, and the downstream NFs in the same process will never be scheduled.

Naive solution: To break the infinite loop inside each NF, one could extract the packet processing logic of each NF, and combine them together to form a synthesized SFC [17, 35, 36]. However, the packet processing logic is closely coupled with NF-specific packet and state abstractions, and combining them results in a large amount of code modification. For example, Snort [10] leverages its unique Data AcQuisition Library (DAQ) to receive packets and fill the metadata like timestamp, protocol annotation, *etc*. Such packet abstraction is incompatible with the packet abstraction under Click [37]. As a result, composing a synthesized SFC of Snort→Click requires to transform the packets as well as all metadata from the DAQ abstraction into the Click abstraction, and vise versa for Click→Snort.

Challenge 3: NFs may affect each other. Being in the same process, all NFs share the same stack and heap. In this way, operators are not able to restrict any memory operations of allocation, read and write, and thus lose the control over illegal accesses, *i.e.*, each NF can (unintentionally) modify others' data and make their states inconsistent. For example in Figure 1b, NF1 (yellow) can silently modify the data (red arrow) allocated by NF2 (green).

Naive solution: Consolidation frameworks often choose to trust the NFs under the same process since they come from the same vendor. However, trust cannot be granted when it comes to heterogeneous NFs across vendors. Using safe language, *e.g.*, Rust in NetBricks [52], to rewrite the NFs is a feasible option, but it's not practical given the large body of existing NFs and limited popularity of the safe language. Formal verification on NFs [72, 74, 75] on the other hand requires verification expertise and sometimes also forces NFs to use certain APIs [73].

3 LemonNFV Overview

The heterogeneous NF consolidation is challenging because NFs in the same process *share* the namespace, the control flow, and the memory. In this section, we introduce the key abstractions for breaking such sharing, and explain why they can be easily adopted without code modification (§3.1). We

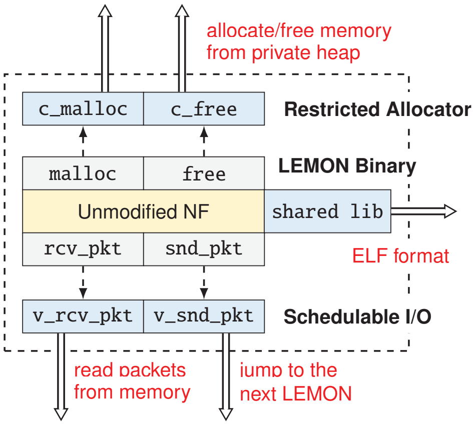


Figure 2: A LEMON with unified abstractions of binary, I/O and memory interfaces. Grey boxes are the common points existed in NFs, and the blue boxes modify/leverage their semantics for providing the unified behaviors (red annotations).

then overview the workflow of LemonNFV that implements a set of utilities upon those abstractions, including the loader, scheduler, isolator and the migration manager, to load, execute and manage LEMONs within the same process (§3.2). We finally discuss the ongoing challenges of LemonNFV (§3.3), which will be addressed in the next section.

3.1 Unified LEMON Abstractions

A LEMON carries an unmodified NF with its own namespace, control flow and memory, each of which could conflict with other LEMONs inside the process. The key challenge here is to decide the proper level to resolve the conflicts. Typical solutions either choose the lowest level for avoiding the code modification, *i.e.*, packaging NFs into OS-level containers, or operate at the highest level for ideal performance, *i.e.*, rewriting NFs' source code, neither of which can fully address our goal.

Instead, LemonNFV aims to resolve such conflicts with three middle-level abstractions: a binary that isolates the namespaces of each NF, a set of I/O interfaces that decouple the packet processing logic from infinite loop, and a set of memory interfaces that restrict the memory operations. These abstractions are high-level enough for resolving the underlying conflicts, while also low-level enough to hide the heterogeneity. In fact, there exists natural homogeneity in all NFs' implementation, and by leveraging it, LemonNFV can equip any NF with the proposed abstractions by a simple interception, *i.e.*, without code modification.

LEMON binary: wrapping the namespace. Simply putting all NFs' code together usually does not compile, because the independent-developed NFs might define variables, functions and dependencies with the same name but different semantics, which would cause name conflicts. Instead of manually wrapping an NF into an isolated namespace, *e.g.*, packing it to a C++ class with private members, we observe that the compiled binary naturally separates the namespaces of each software module, and the conflicts can be resolved through the symbol resolution process when loading the binary.

Natural homogeneity: ELF is the standard format for executables and shared libraries under Linux, which reveals a chance for LemonNFV to package each NF as a software module without modifying its native code but through a simple recompilation (right part in Figure 2).

Schedulable I/O: creating entry points. The packet processing logic of NFs are usually implemented as an infinite loop, which is not schedulable. Nevertheless, we observe that the I/O behaviors reveal the natural boundaries between NFs. To this end, we design a new set of I/O in substitution for the original, which creates explicit entry points of each NF for scheduling: for the packet receiving function, it could read packets from a shared memory region instead of the physical NIC; for packet sending, it could jump out of the infinite loop after pushing the packets back to specific queues on shared memory, according to the output port of the NF.

Natural homogeneity: NFs are built on *a handful of* I/O libraries like DPDK and `libpcap`, which means we can implement the schedulable I/O by only intercepting limited functions of these libraries (lower part in Figure 2).

Restricted allocator: separating memory domains. To realize the isolation requirement, one should create isolated memory regions for each of the NFs, even with the same process. While not all NFs have the compilation support from safe languages like Rust, the feasible solution is to create separate (instead of interleaved) memory domains for each NF, and protect those domains with bound guard [67] or privilege management [14, 53].

Natural homogeneity: All NFs rely on the native allocator, which only provides limited functions like `malloc`, `realloc` and `free`. That is, LemonNFV can override the native allocator with the restricted allocator, which enforces that the dynamic memory allocation in a LEMON takes place in its own heap (upper part in Figure 2).

3.2 LemonNFV Workflow

Having the above unified LEMON abstractions, LemonNFV implements several key components to build, execute and manage LEMONs. As shown in Figure 3, an SFC is a process (dashed rectangle) that runs two types of threads: hypervisor (red) and worker (gray). The hypervisor consists of two components: the LEMON loader that loads LEMONs and intercepts the original allocator and I/O functions, and the migration manager that migrates the LEMON to another worker, SFC, or server. The worker implements the trampolines, which process the packets from the hardware NIC, schedule a chain of LEMONs with virtualized I/O while ensuring their isolation. There is also a LemonNFV controller, relaying the user commands like LEMON loading to the hypervisors, which can be deployed on a remote server.

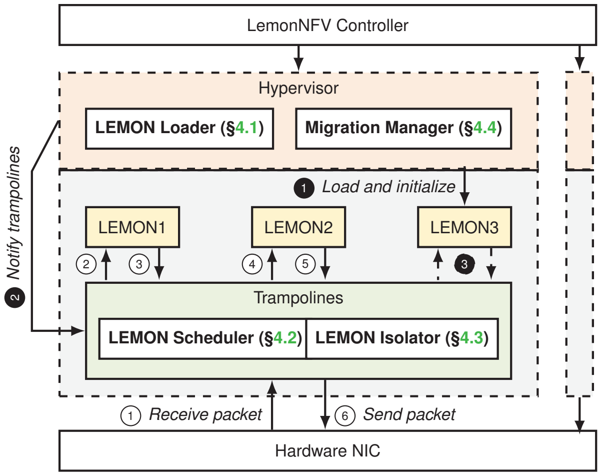


Figure 3: LemonNFV workflow, where a dashed rectangle represents a process (SFC) with two threads: a hypervisor (red) and a worker (grey). ①–⑥ illustrate how a packet being processed; and ❶–❸ are for runtime SFC changing.

Loading a LEMON. Each LEMON carries an unmodified NF, as well as its dependent libraries and configuration files. To this end, the NF developers are required to recompile the NFs into shared libraries, which only needs a few lines of modification in the `make` rules (See §6.2). This task can also be done by the operators if the target NF is open-source. Then, the operator should fetch all dependencies of the LEMON, pass their paths to the loader, and specify the total amount of memory needed. With these information, the LEMON loader would allocate the memory, load the code and variable segments, resolve the potential conflicts, and grant the corresponding privileges. The user can then consolidate an SFC by providing the path and interconnection of each LEMON in an interactive terminal.

Chaining and isolating LEMONs. Inside each worker, LemonNFV executes a chain of LEMONs in the RTC way. When receiving a (batch of) packet(s) from NIC (①), the LEMON scheduler in the trampolines will pass the packet to the first LEMON of the SFC, by calling its schedulable packet-receiving function (②). After the processing in LEMON1, the schedulable packet-sending function would transfer the control flow back to the trampolines (③), which would trigger the next LEMON, *i.e.*, LEMON2 (④). When it reaches the end of the chain (⑤), the trampolines send the packet(s) back to NIC (⑥). All of ①–⑥ are done within a single thread, thus will not produce any inter-core communication or thread context switch. When an SFC wants to scale out its performance, LemonNFV can duplicate the worker into more cores, and dispatch the flows using hardware NIC.

The trampolines also ensure the LEMON isolation through above chaining process. Specifically, the LEMON isolator would adjust the memory access privileges of each LEMON, *e.g.*, when executing LEMON2, the memory domains of LEMON1 and LEMON3 should be protected.

Managing LEMONs in runtime. Consider a simple management task that attaching LEMON3 to the end of current SFC. The hypervisor would first load and initialize LEMON3 (❶) using the LEMON loader. After that, it will notify the trampolines with the new LEMON, *i.e.*, the packet receiving function of LEMON3 (❷). Finally, the trampolines will execute LEMON3 next time a packet leaves LEMON2 (❸). Note that the hypervisor is in an individual thread, which means ❶ can be done asynchronously without halting SFC. ❷ is also lightweight, as the hypervisor only needs to notify the trampolines with the new entry point, which is a rare operation and would only halt the SFC for a negligible moment.

3.3 Ongoing Challenges

While making the LEMON interoperation a possible vision, LemonNFV is still faced with several practical challenges.

Isolated LEMON namespace. LEMON binary wraps the namespaces of each NF, but the name conflicts between multiple binaries still needs to be resolved. This is the responsibility of the LEMON loader, while the OS-default loader cannot suffice, because it tends to reuse the dependencies for all LEMONs.

Correct LEMON scheduling. Having the schedulable I/O, the LEMON scheduler in the trampolines needs to further address the following concerns for correctly scheduling the LEMONs. (1) how to efficiently switch LEMONs; (2) how to properly execute the logic other than the packet processing, *e.g.*, initialization; and (3) how to correctly handle the complex I/O behaviors like asynchronous Rx and Tx.

Efficient memory isolation. The restricted memory allocator guarantees the separate memory domains for each LEMON. However, it is still unclear how the LEMON isolator restricts the memory accesses to the legal areas. The key challenge here is to isolate those memory domains without compromising too much performance.

Flexible LEMON migration. NF migration is a critical requirement in NFV systems. Even with the help of the LEMON loader, it is still unclear for the migration manager to migrate the LEMONs to other workers, SFCs or even servers. The key challenge here is that the LEMON loader can only handle a LEMON as a whole, while the inner data structures (*i.e.*, NF states) of a LEMON might need to be *partially* migrated to another core or re-accessed in a different process for intra- or inter-server load balancing.

4 Detailed Design of LemonNFV

In this section, we discuss in details how LEMONs are loaded (§4.1), scheduled (§4.2), isolated (4.3) and migrated (§4.4) using the unified abstractions of LEMON.

4.1 Loading the LEMONs

Given an ELF binary, the loader is responsible to allocate the memory for the executable, copy the segments to the corresponding memory regions, resolve the external symbols, and finally call the constructors. While Linux has offered a mature toolchain for loading the ELF file at runtime, *i.e.*, `dl`-family functions, they are ill-suited for serving as the LEMON loader. In the following, we explain the specific requirements when loading a LEMON, and present our solution for the LEMON loader.

Dependency isolation. The OS loader, *i.e.*, `ld.so` with `dlopen`, attempts to reuse the libraries that have been already loaded for saving the memory and improving instruction cache affinity. However, this would cause dirty access of common libraries if multiple LEMONs depend on them.

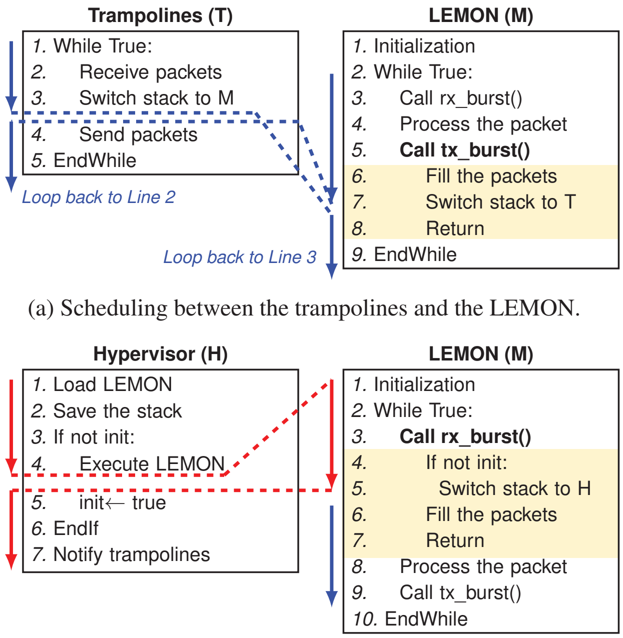
In LemonNFV, the LEMON loader views each NF and its dependencies as a sandbox, and will load the dependencies no matter whether other LEMONs have already loaded them. For example, each LEMON will load its own `libc`, such that they will not share the global variables like `optarg`. so that

Targeted symbol resolution. While loading, lots of functions in LEMONs should be intercepted to the customized versions, *e.g.*, the DPDK I/O to the schedulable I/O, the native `malloc` to the restricted `malloc`. The common solution to this task is `LD_PRELOAD`, which instructs the loader to first lookup the preload libraries when resolving every symbol of the executable. However, `LD_PRELOAD` redirects *all* symbols with the same name, while LEMONs and trampolines should use different versions. For example, trampolines call `rx_burst` to receive packets from NIC, which should not be resolved to the schedulable version as in LEMONs. Besides, LEMONs may depend on different versions of the libraries, and the intercepted functions with the same name also need vary to those semantics.

LemonNFV implements a symbol resolution mechanism tailored for loading LEMONs, which allows the trampolines and each LEMON to specify their own preload libraries, enabling different semantics for symbols of the same name.

Consistent loading address. The Linux loader cannot manually specify the loading address. Instead, the recent Linux kernels enable the random address loading, *e.g.*, ASLR [1], mostly due to the security reason. Such random loading would disable the ability of reloading a LEMON, because all pointers in the reloaded LEMON would become invalid. This feature is critical for fault recovery and LEMON migration.

To this end, the LemonNFV process reserves the same virtual address space, which is partitioned into fixed-size *slots*. the LEMON loader would load each LEMON to its *unique* slot, and allocate the fixed address regions for the private stack, heap and dependencies. As a result, all the pointers (expect packet pointers) in a LEMON snapshot would remain valid even being reloaded or migrated to another process.



```
Trampolines (T)
1. While True:
2.     Receive packets
3.     Switch stack to M
4.     Send packets
5. EndWhile
```

```
LEMON (M)
1. Initialization
2. While True:
3.     Call rx_burst()
4.     Process the packet
5.     Call tx_burst()
6.         Fill the packets
7.         Switch stack to T
8.         Return
9. EndWhile
```

(a) Scheduling between the trampolines and the LEMON.

```
Hypervisor (H)
1. Load LEMON
2. Save the stack
3. If not init:
4.     Execute LEMON
5.     init← true
6. EndIf
7. Notify trampolines
```

```
LEMON (M)
1. Initialization
2. While True:
3.     Call rx_burst()
4.         If not init:
5.             Switch stack to H
6.         Fill the packets
7.         Return
8.     Process the packet
9.     Call tx_burst()
10. EndWhile
```

(b) Initialization with the hypervisor.

Figure 4: Scheduling LEMONs with the schedulable I/O and private stacks. Solid arrows indicate the executing paths, and dashed lines are the function/stack transitions. The blue paths are executed by the working thread, and the red ones are from the hypervisor thread. Shadowed texts are the pseudocode of `tx_burst` and `rx_burst`.

4.2 Scheduling the LEMONs

A typical NF implementation consists of four stages: (1) the NF initializes its own data structures and hardware; then it starts an infinite loop which (2) receives packets using a function like `rx_burst(pkts)`; (3) processes the packets; and (4) sends the packets out using a function like `tx_burst(pkts)`. The LEMON I/O interfaces unify the way for LEMONs to fetch and send packets, *i.e.*, stage 2 and stage 4. Specifically, the LemonNFV trampolines are responsible to communicate with the hardware NIC. And the “NIC” in the LEMON is actually a memory region that stores the packets. As a result, `rx_burst` should fill `pkts` (the pointers of packets) with the packets from the trampolines; and `tx_burst` should fill `pkts` back to trampolines for the downstream LEMONs.

Having those basic I/O operations, how to schedule the LEMONs such that the packets can flow through them as if they were chained together will be our focus here.

Scheduling LEMONs with schedulable I/O. Each NF has its own control flow, from `main` function to the infinite loop of packet processing. The LEMON scheduler needs to cooperate with those control flows to properly jump into and out from the LEMON execution.

To this end, LemonNFV creates private stack for each LEMON, making its control flow separated from the trampolines and one another. Specifically, LemonNFV allocates a dedicated memory region for each LEMON as its stack, and the trampolines maintain the corresponding stack pointers (*i.e.*, `SP` and `BP` registers). In this way, the LEMON scheduling can be simply implemented as saving the current states of registers (stack pointers and other callee-saved registers) and restoring the previously saved ones of the target LEMON. This process operates purely in the user space, thus incurs much less overhead than the context switch between processes or threads, which would trap into kernel and flush TLBs. We implement the above stack switch logic in the packet sending functions, because each time the NF is sending the current batch of packets out of the NIC, the control flow should move to the next NF in the SFC.

We use Figure 4a to illustrate how the scheduling works. Assume the SFC has only one LEMON, which has processed a batch of packets and is sending them out, *i.e.*, calling (the schedulable) `tx_burst` in Line 5 of M. After filling the packets, the execution stack is saved and switched to the trampolines (from Line 7 of M to Line 3 of T). The trampolines then send the packets to the NIC (Line 4 of T). In the next loop it receives a new batch of packets (Line 2 of T) and schedules the LEMON again (Line 3 of T), which will jump back to Line 7 of M. The LEMON will continue to receive and process the packets from the trampolines, *i.e.*, Line 8–9, 3–4 of M, until it is scheduled out, *i.e.*, `tx_burst` is called again. Any additional logic besides Line 4 (*e.g.*, profiling after sending packets) will also be executed at this moment.

Handling the logic other than packet processing. The hypervisor needs to take care of the LEMON initialization, which, like the packet processing logic, has no explicit boundary in the NF's code. To this end, we view the initialization as the logics from the first line of `main` function to the *very first time* the `rx_burst` is called, which means all preparations for packet processing have been done. We use Figure 4b to depict such process. The hypervisor thread (red path) loads the LEMON, saves its stack and executes the LEMON (Line 1–4 of H). After LEMON is initialized (Line 1 of M), it will eventually call (the virtualized) `rx_burst` (Line 3 of M). For the very first time it is called (*i.e.*, `init==0`), the LEMON should switch back to hypervisor's logic, and the hypervisor will notify the trampolines that the initialization is done. The trampolines will execute the LEMON from the saved stack (Line 6 of M) next time it is scheduled.

Except for the initialization, NFs may also include logic for event logging and runtime configuration. These routines are usually conducted in individual threads. See Appendix A for scheduling a LEMON with such threads.

Complex I/O behaviors. The above scheduling assumes the NFs call the packet I/O in a synchronized way, *i.e.*, receiving (Rx) and sending (Tx) once per batch, while NFs can also invoke less Rx and more Tx (*e.g.*, multicast), or more Rx and less Tx (*e.g.*, packet buffering). The current scheduling is based on Tx, thus can still handle the former case, and we extend the virtualized Rx I/O to deal with the latter. To be specific, we add a flag in virtualized Rx function to check whether it is called *for the first time in this batch*. If not, the Tx function is not called, which means this LEMON buffers or drops all packets, and blocks the downstream LEMONs. In such case, the trampolines should continue to receive the next batch instead of scheduling the next LEMON, ensuring correctness of NFs that buffer packets (*e.g.*, Reframer [28]).

4.3 Isolating the LEMONs

LemonNFV provides each LEMON with a separate memory domain via the custom allocators, so that the legal region a LEMON can access is bounded. However, achieving this is far from sufficient to isolate each LEMON from each other, because the illegal accesses are only *defined* but not *prevented*. We now discuss how LemonNFV checks illegal memory accesses efficiently. In the following, we first present the threat model of LEMON isolation, then introduce two *software fault isolation (SFI)* techniques to sandbox the memory accesses. After making our design choice, we present how LemonNFV realizes the LEMON isolation in runtime.

Threat model. LemonNFV isolates the SFCs from different tenants with processes. For each process, LemonNFV allocates two virtual functions (VFs), *i.e.*, NICs virtualized by SR-IOV [15], which provide almost the same performance compared to the physical NIC. Given such strict isolation between SFCs, we only need to take care the isolation between LEMONs along the SFC, *i.e.*, the intra-process isolation.

We assume each NF has its own packet processing logic, which are expected to be independent of the others. That is, even in the same process/thread, the data and states of NFs are strictly independent. This assumption disables most communications between NFs, and thus is weaker than NetBricks's threat model that supports state sharing [52]. However, we argue that this assumption is actually aligned with the case when chaining physical middleboxes, where the internal states are unavailable to external NFs, and packets are the only information that can be exchanged between NFs.

We assume the trampolines and hypervisor are written with care, while NFs are written in a negligent way that they might illegally modify the data, *e.g.*, overwrite the state in other NFs or hypervisor. Such bad operations can happen in either NF itself, *e.g.*, `*(bad_addr)=1`, or the libraries it depends, *e.g.*, `memset(good_addr,0,bad_size)`.

Two design options. The above threat model falls into the SFI area, where the most classical implementation is to check the bound of each memory write [38,67,71], *e.g.*, `*p=1` would be modified into `if(p>L&&p<H) *p=1` to ensure `p` is always within its own memory region `(L,H)`. Its performance is determined by (1) the number of the legal regions, since each instrumentation must check all these regions, and (2) the number of memory access statements, which equals to the number of instrumentation.

Recent studies advocate to leverage hardware-aided techniques to realize SFI [31,45,66]. To be specific, they partition the memory into domains, and a certain software module can only access its legal domains. When switching to another software module, this method has to change the legality of the domains. As such, its runtime overhead is mainly determined by the number of domains. Another defect of domain switching is that it restricts the number of domains due to the limitation of hardware.

Our design decision. Bounds checking would incur unacceptable runtime overhead for realizing LEMON isolation for the following reasons. First, the legal regions of a LEMON, *e.g.*, code, stack and heap, are separate, making each instrumentation quite costly. Second, the number of memory access statements could be large for NFs like encryption, which further lowers the performance. To make the performance penalty clear, we implement an extra compiling pass in LLVM that injects bound checking before every memory write at the IR level. We then prototype three typical NFs including NAT, firewall and IDS, and by chaining them into an SFC, we find that the isolated SFC is in average 24% slower than the original one. Such penalty is usually unacceptable for realizing the wire-speed processing.

On the contrary, we find domain switching is quite suitable for LEMON isolation. Recall the threat model that LEMONs do not share states between each other. That is, each LEMON can be packaged into a disjoint domain, such that the number of domain switching is actually the length of SFC, which is fixed and stable no matter how complex a LEMON is. For the limitation on number of domains, it should still be sufficient for LEMON isolation, since the length of SFC is usually small, *i.e.*, less than 10 for most real-world cases [4]. As a result, the domain switching approach is preferred to realize SFI between LEMONs.

Among the others like VMFUNC that require code modification [38,45], PKU has been viewed as the fastest domain switching approach and requires little modification on existing code [31,66]. PKU uses the spare four bits in each page table entry to partition memory into 16 domains, and specifies the access restrictions for each domain by a *pkey*. In doing so, the protection can be naturally guaranteed when accessing the page table, and thus incurs *zero extra cost* in runtime. More importantly, the operation of domain-switching, *i.e.*, writing the permissions to pkeys, purely works in userspace and only incurs negligible overhead, *i.e.*, less than 100 cycles.

Isolating LEMONs with PKU. Inside the process, LemonNFV specifies one pkey for each LEMON. In runtime, when a LEMON is scheduled by the trampolines, LemonNFV enables its pkey to grant the access rights to its own domain, *i.e.*, its own stack, heap and data segments, and disables the access to any other domain. Since the switching happens when scheduling LEMONs, the switch logic is embedded into the schedulable I/O for each LEMON.

Consider a simple example with two LEMONs, M1 and M2. The trampolines' domain occupies pkey0, and each LEMON (as well as its stack, heap and dependencies), is allocated with one pkey, *i.e.*, pkey1 and pkey2, respectively. The packet domain is always readable/writable to trampolines and LEMONs, and thus does not need a specific pkey to protect. At runtime, when trampolines are receiving or sending packets, all three pkeys are enabled, because trampolines have the full visibility to all domains. Before going into M1, trampolines would disable pkey0 and pkey2. After M1's processing, the schedulable I/O in M1 would enable pkey2 for M2's processing. After the processing of the last LEMON, *i.e.*, M2, its schedulable I/O will disable pkey2 and enable pkey0 to jump back to trampolines. Note that to reduce the domain switching, M1 would directly jump to M2 instead of jumping back to the trampolines. In general, LemonNFV would switch $N+1$ times for an SFC with N NFs.

4.4 Migrating the LEMONs

NF migration is essential when operators seek a balanced load and/or efficient resource utilization. Existing works realize this feature on top of a fully supervised infrastructure, *e.g.*, OS-level virtualization [12, 44], or through specific migration interfaces, *e.g.*, OpenNF [27]. Without above prerequisites, LemonNFV leverages the standard LEMON binary to empower users to migrate the LEMONs to other cores and machines in an efficient way [2].

Intra-process LEMON migration. In consolidation approach, it is important that packets should be evenly dispatched to the workers, otherwise the CPU resources are wasted [35]. Due to the dynamics in network (*e.g.*, traffic burst) and resources (*e.g.*, adding a core), such balance is hard to achieve if statically binding packet classes (*i.e.*, a set of flows) to cores. Instead, the NFV framework needs to migrate the packet classes *and* the corresponding NF states to a new core, to balance the working load.

LemonNFV addresses this challenge by relieving the close binding between cores and LEMONs, making the consistent migration possible for any state structures used in LEMONs. To be specific, LemonNFV creates a pool of LEMONs, each of which is dedicated for a packet class. Given the simple fact that both executable code and states are within the LEMON, "migrating the state of packet class P to core C" becomes "letting core C execute the LEMON corresponding to P".

Figure 5 shows a simple migration scenario, where the operator allocates two cores (C_1 and C_2) to handle three packet classes (P_1–P_3). In this case, LemonNFV will create three LEMONs (M_1–M_3) dedicated for P_1–P_3, and ensure that cores will always handle the packet classes with their corresponding LEMONs. Assume we want to migrate P_2 to C_2. The

[2] The proposed migration schemes are for common load balancing scenarios, while specific migration cases, *e.g.*, migrating an arbitrary flow, balancing a non-splittable flow, are not supported. See §7 for a discussion.

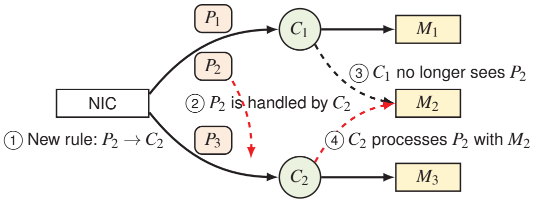

P_1
C_1
M_1
P_2
③ C_1 no longer sees P_2
NIC
② P_2 is handled by C_2
M_2
① New rule: $P_2 \rightarrow C_2$
P_3
④ C_2 processes P_2 with M_2
C_2
M_3

Figure 5: Intra-server LEMON migration. C_1 and C_2 are cores, P_1–P_3 are three packet classes, and M_1–M_3 are three identical LEMONs dedicated for P_1–P_3.

hypervisor first issues a new rule to the NIC (or modifies the global RSS table [16]), which directs the NIC to tag and send P_2 to the queue binding to C_2 (①). At this point, all packets belonging to P_2 will be handled by C_2 (②), and C_1 cannot see P_2's packets immediately (③). When C_2 receives the packets of P_2, it can safely process them with M_2 (④). Since the states of P_2 are always within M_2, there is no need to synchronize or copy states between cores. However, since each core has a receive queue, there is a minor risk that C_1 still holds a few packets of P_2 after the migration. For this case, we could create a receive queue for each packet class, and let different cores to handle the queue while migration, which would cost little performance [16].

②–④ are natural consequences of ①, meaning that the overhead of migration is essentially the time of rule installation, which only amounts to sub-milliseconds.

Inter-process LEMON migration. Besides the intra-server load balancing, the network-wide load balancing calls for migrating LEMONs across different LemonNFV servers (processes) in runtime. This is done by (1) allocating identical addresses for LEMONs across processes and (2) migrating the snapshots (memory dump) of LEMONs iteratively.

For the first, the LEMON loader has ensured the address consistency across different processes. One concern here is that can a single process provide enough address space, if considering different types of NFs and per-packet-class pool of LEMONs. In fact, modern 64-bit system can easily reserve 96TB of virtual memory (*e.g.*, 0x100000000000 - 0x700000000000) to support 32768 LEMON instances (6GB memory each), *i.e.*, 128 types of NFs with 256 packet classes, which are sufficient given limited number of popular NFs. Note that this restriction is for the unique LEMONs in *network-wide* servers, and the length of SFC in a single server is still bounded by the physical memory and PKU limitation.

Secondly, LemonNFV needs to realize a packet-lossless migration. In runtime migration, the snapshot is taken after the LEMON loosing references over a batch of packets, and is transmitted to its destination. The LEMON loader then loads it into the corresponding slot and modifies the SFC. Since all states are within the LEMON, the migration will not lose any state. However, the cross-machine transmission might be time-consuming due to the large snapshot, resulting in packet losses to the LEMON.

To this end, LemonNFV iteratively copies the snapshot [12]. Specifically, in phase -1, LemonNFV pre-copies the whole snapshot to the target, and uses dirty page tracking [8] to locate that a portion of memory d becomes dirty through this phase. Then in phase 0, LemonNFV only transmits the dirty bytes, and locates the new dirty memory d. When d is getting smaller, the transmission time also decreases, which further reduces d in the next phase. Iteratively, the dirty memory transmission can finish within a negligible time, which is the right timing to route the packets to the new machine. See Appendix B for a detailed implementation.

5 Implementation

We implement a prototype of LemonNFV with 5K lines of C code, including the unified abstractions, *i.e.*, the schedulable I/O and the restricted allocator, and the system components, *i.e.*, the LEMON loader, the LEMON scheduler and the LEMON isolator with PKU. We highlight several key implementations and enhancements in our prototype.

Schedulable I/O. The prototype implements a schedulable version for all I/O functions used by NFs in §6, which includes 28 libpcap [7] and 41 DPDK [5] functions. The user can easily intercept I/O interfaces not included in the prototype by adding functions to the preload libraries when involving new NFs.

Restricted allocator. As mentioned in §3.2, each LEMON should notify its memory budget to LemonNFV. This is to avoid the runtime fault like memory leak: if the customized memory allocator detects that a LEMON exceeds its memory limitation, LemonNFV can unload it before it drains all host memory. Besides, this also enables a simple but effective optimization, *i.e.*, memory pre-allocation, which can largely improve the memory performance in runtime for memory-intensive NFs. To be specific, when a LEMON requires a certain portion of memory, the customized memory allocator pre-allocates all memory in the heap, so that it does not need to make expensive syscalls like `mmap` in runtime.

Fault isolation. §4.3 mainly considers the memory isolation, while the *fault isolation* is also critical for consolidation approaches. Since all NFs are in the same process, a runtime fault, *e.g.*, divided by zero, of a single NF would fail the whole chain. This task is addressable with the help of the trampolines in LemonNFV. First, LemonNFV registers a set of signal handlers for dealing with the runtime fault like `SIGABRT`, `SIGSEGV` and `SIGILL`. Then, once those signals are captured, the trampoline can decide how to prevent the faulty LEMONs from impacting others. In the prototype we simply remove the faulty one and pass the packets to downstream LEMONs. Other policies like restoring a LEMON to its nearest checkpoint can also be implemented based on the above fault capture scheme.

6 Evaluation

In this section, we evaluate the practicality, performance and overhead of LemonNFV with real and synthesized NFs. We are particularly interested in the following questions.

1) Can LemonNFV consolidate heterogeneous NFs to obtain high performance without much coding effort? Experiments show that LemonNFV can consolidate real NFs from different frameworks as the virtualization approach, *i.e.*, *without modifying native code*, and achieve a high-performance SFC as the consolidation approach, *i.e.*, *incurring minor overhead between NFs* (§6.2).
2) Can LemonNFV outperform state-of-the-art NFV systems? Experiments show that LemonNFV is *1.9–2.4*× faster than a state-of-the-art virtualization approach, and incurs only *0.7–4.3%* overhead compared to a state-of-the-art consolidation approach (§6.3).

6.1 Experimental Setup

Testbed. Our testbed is an x86 machine (24×Intel Xeon 3Ghz, 256GB memory) equipped with a Mellanox CX-6 DX NIC (2-port 100Gbps). Hyperthreading and frequency boosting are disabled in all CPUs, and the host OS is Ubuntu 20.04 with Linux kernel 5.11. We use DPDK 21.05 as the packet I/O for LemonNFV with 32 as the batch size, and enable S-RSS in multi-core experiments.

We prepare two traces for testing: ISP trace from a large service provider that contains majorly TCP flows (8.6M packets, 3.9M active flows, 652 bytes in average) and ENT trace captured from an enterprise network which mostly consists of GTP (UDP) packets (11.2M packets, 462 bytes in average).

Another server with the same NIC replays the traces to the testbed at line rate, serving as the packet generator. The testbed is configured to forward all traffic back to the generator, no matter whether NFs drop them or not. We run each experiment under 100Gbps traffic and record the average value of 20 seconds. Each experiment is then repeated 10 times and the average result is reported.

Real NFs. We consider NFs built upon the fast userspace I/O (DPDK) and kernel I/O (`libpcap`). We involve three DPDK NFs: an IDS based on Rubik [41] that matches the reassembled payload with snort-like rules; a NAT based on FastClick [17] that is composed of many inherent elements like traffic classifier and ARP querier; and an ACL based on NetBricks [52]. We further involve two `libpcap` NFs: a connection tracker (CT) based on mOS [34] that tracks the status of TCP connections; and a DPI tool based on nDPI [9] that can identify 170+ L7 applications.

Synthesized NFs. We further choose several synthesized NFs which are feasible to be ported, such that we can fairly compare them under different frameworks. We use NFD [32] to produce four stateful NFs: network address port translator, heavy hitter detector, super spreader detector, and UDP flood mitigation. We further extract four NFs from OpenNetVM's repository [76], including payload scanning, stateless firewall, AES encryption and decryption, which are stateless but computing-intensive.

Table 1: The real NFs and the efforts for interoperation.

NF	Framework	Lang.	I/O	CN	CF	CH
IDS	Rubik [41]	C	DPDK	337	31K	2
NAT	FastClick [17]	C++	DPDK	94	331K	2
ACL	NetBricks [52]	Rust	DPDK	401	58K	8
CT	mOS [34]	C	`libpcap`	325	139K	4
DPI	nDPI [9]	C	`libpcap`	4498	121K	2

CN: *the LOC of the NF logic*, ***CF:*** *the LOC of the framework*
CH: *the LOC modified by LemonNFV for interoperation*

6.2 Comparing with the Ideals

When interoperating, *i.e.*, chaining, isolating and managing, heterogeneous NFs, we have two ultimate goals: without code modification *and* performance penalty. To this end, we use the real NFs to compare LemonNFV with two ideals, *i.e.*, a virtualization approach that does not modify a single line of code, and a pure consolidation approach that does not incur any performance penalty.

Efforts of interoperating NFs. As shown in Table 1, the real NFs are heterogeneous in many ways including the language, I/O, *etc*. We assume the virtualization approach emulates a full running environment, *e.g.*, with VM or Docker, and thus does not need to modify the real NFs for chaining them. On the other hand, the consolidation approach needs to extract or wrap the NF logic for interoperation.

Intuitively, it would only cost limited coding efforts, since the high-level NF logic is relatively simple and neat, as shown in CN in the table. However, the factual task would cost much more than that number. For example, to implement an NAT in FastClick only needs to write 94 LOC for a Click script, while to interoperate this NF with CT in mOS, one has to dig into the detailed implementation in FastClick *and* mOS, to ensure they have the same packet abstractions, are not conflicting in variable/function names, and do not incur dirty writes on global data structures, *etc*. Things get more complex when porting NFs with different languages, *e.g.*, rewriting a Rust NF into a C/C++ one. Generally, the effort of code reading and writing for consolidating heterogeneous NFs would approach to the numbers shown in CF.

LemonNFV does not modify a single line of native code (*e.g.*, C/C++, Rust) for consolidating these NFs. The only effort we made is to modify the compilation configurations of the NFs, *e.g.*, adding `-fPIC` and `-shared` in `Makefile`, to compile the NF as a shared library instead of an executable, which amounts to a handful of LOC. We also feed the command line arguments, *i.e.*, `argv`, to each LEMON with a configuration file, which also contains minor LOC. CH in Table 1 shows the total number of modified LOC.

Table 2: Performance comparison over real NFs.

	Per-Packet Latency (us)			Throughput (Gbps)		
	Ideal	Docker	LemonNFV	Ideal	Docker	LemonNFV
ISP	0.66	240 (+362×)	0.68 (+3.8%)	22.9	11.6 (-49%)	22.0 (-3.8%)
ENT	0.59	224 (+223×)	0.60 (+2.8%)	18.3	12.7 (-31%)	17.7 (-3.2%)

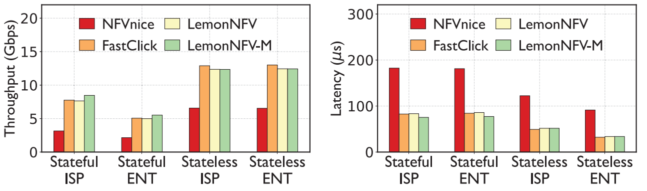
Performance comparison. We chain the real NFs to conduct a sequential SFC: IDS→NAT→ACL. We do not involve the two `libpcap` NFs here because the slow packet I/O would significantly enlarge the performance gap between virtualization and other approaches. As a reference, CT and DPI with LemonNFV are 5.6× and 3× faster than their `libpcap` versions respectively. We consider two performance metrics: the processing time of each packet, *i.e.*, the elapsed time after it entering the first NF and before it leaving the last NF, and the end-to-end throughput.

For virtualization approach, we use Docker containers to carry the NFs. Each container is equipped with one CPU core and two virtual NICs, and the SR-IOV [15] technique ensures the line-rate packet forwarding between Dockers without CPU interventions. Due to the large codebase of real NFs, we do not implement a pure consolidation approach. Instead, we estimate its performance as follows: for the per-packet latency, we simply accumulate the processing time of each single NF; for the throughput, we set up a basic DPDK I/O, and delay each packet for the accumulated processing latency. This should present the performance upper bound of consolidation since interference between NFs (*e.g.*, cache and memory contention) is removed.

Table 2 shows the performance comparison. It can be seen that the Docker approach adds more than 200× latency compared to the Ideal approach. Since those NFs are with high-speed I/O, the major overhead comes from the packet pool in the virtual NIC and DPDK, *i.e.*, *V2* mentioned in §2.1. On the other hand, LemonNFV eliminates such overhead and only incurs 2.8%–3.8% overhead, largely resulting from the PKU switch between NFs.

Since Docker uses three separate cores, we also scale Ideal and LemonNFV with three cores for throughput comparison. The results show that Docker is in average 40% slower than Ideal, while the overhead of LemonNFV is just 3.5%. The performance gap between Docker and LemonNFV is narrowed compared to the per-packet latency because the end-to-end measurement includes the I/O latency between the testbed and the packet generator (∼100us). Such gap would again enlarge with longer SFC.

Loading and migration. Under LemonNFV, loading a LEMON is essentially loading a shared library, which is fast and predictable. We measure the loading time of all 5 real NFs, and report that the max/average loading time is 28ms/7ms. Note that LEMONs are loaded by the hypervisor in an asynchronous way, so the factual overhead for the SFC can be neglected. As a comparison, booting a container usually takes hundreds of milliseconds [47].



(a) Throughput

(b) Latency

Figure 6: Performance comparison with different SFCs and traces using 4 cores. LemonNFV-M indicates LemonNFV with memory pre-allocation enabled.

We also verify that our inter-process migration can be finished rapidly. We setup two processes on a single server, each of which carries a naive packet forwarding LEMON. The source process then loads IDS in Table 1, runs it for some time, and migrates it to the destination process. Results show that iterative migration copies over 400 dirty pages within 1.2s, and efficiently reduces the total downtime to only 6.9ms.

6.3 Comparing with Existing NFV Systems

We compare LemonNFV with state-of-the-art NFV frameworks. For obtaining higher performance, these frameworks often require developing NFs using specific interfaces. To this end, we port the synthensized NFs to these frameworks and compose two SFCs: Stateful that chains the four stateful NFs from NFD, and Stateless that chains the four OpenNetVM NFs. Performance is the focus of this comparison.

Comparing with NFVnice. We port the synthesized NFs into NFVnice [40], a state-of-the-art virtualization-based approach that deploys NFs inside processes and enables back-pressure between them to minimize wasted work. NFVnice sets a large packet queue (2^{15}) between each NF for higher throughput and less packet loss, which would in turn increase the latency. To this end, we measure the throughput with the default setting, and measure the latency by reducing the queue size to 32 (default batch size). This could represent the ideal performance a virtualization approach can achieve.

Figure 6 shows the comparison of NFVnice and LemonNFV. LemonNFV is at least 88% faster than NFVnice with 55% less latency. Note that besides the four worker cores, NFVnice employs two extra cores dedicated for packet I/O, which, however, has been heavily hindered by the context switches and cache misses when traversing SFCs.

Comparing with FastClick. FastClick [17] is an enhanced version of Click [37] with high-speed I/O, optimized compilation stages, and many useful elements. FastClick is the basis of many state-of-the-art consolidation approaches like Metron [35], RSS++ [16], PacketMill [25] and Reframer [28]. In FastClick, each NF is a C++ object, and chaining NFs is just calling a function of the object. This implementation eliminates all extra overhead between NFs, indicating the ideal performance of a consolidation approach.

Figure 6 shows that LemonNFV is only 1.5% slower than

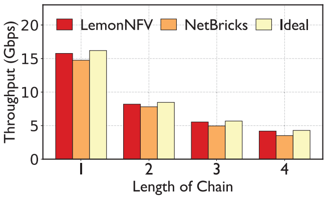


Figure 7: Throughput comparison with NetBricks.

FastClick for Stateful. This demonstrates the end-to-end overhead employed by LemonNFV, including the stack switch, the pkey set, *etc*. The overhead increases to 4.2% for Stateless, because this SFC is more lightweight, making the cost from LemonNFV more significant.

After enabling the pre-allocation, LemonNFV-M even outperforms FastClick by 9.2% in Stateful. This is because NFs in Stateful are memory-access-intensive, such that the pre-allocation would improve the performance significantly. For the similar reason, Stateless performs almost the same with and without this feature.

Comparing with NetBricks. The closest work to ours is NetBricks [52], which also runs SFC in a single thread and provides isolation among NFs. To compare the runtime overhead when hosting NFs, we implement a simple ACL NF using C in LemonNFV and Rust in NetBricks. This NF does not use any complex algorithms and data structures, *e.g.*, hash table, from the standard library, to minimize the performance difference from the language. We chain this NF for several times to measure the scalability, and further involve the pure consolidation, *i.e.*, chaining the C NF by function call, as the ideal approach.

Figure 7 shows that LemonNFV is on average 2.6% slower than Ideal, while NetBricks incurs 11.9% overhead. With a longer chain, the overhead of LemonNFV is stable (always ∼2.5%), but NetBricks incurs larger overhead (from 8.9% to 18.1%). This is because the overhead of NetBricks mainly comes from the NF logic itself, *i.e.*, checking array bounds with Rust (reportedly a 14% overhead for an LPM [52]), while the overhead of LemonNFV is irrelevant to the NF logic. As a result, though the increasing chain length also increases the number of domain switching, such overhead is still negligible compared to the workload of NF itself. Considering that the example NF is largely simplified, the factual gap between LemonNFV and NetBricks would be much significant in real cases.

Overhead analysis. We quantify the overhead employed by LemonNFV to better understand the performance illustrated above. For each LEMON switching, LemonNFV incurs the following overhead: (O1) switching the memory domain by writing the pkey registers, (O2) switching the private heap, and (O3) switching the private stack to the target domain.

For O1, each domain switching invokes one write to the pkey register, which averages to 82 cycles in both SFCs. O2 is stable and minor (*i.e.*, 9 cycles), because heap switching only changes the base pointer of the heap. For O3, stack switching needs to save the current context and restore the target. We measure such overhead for each NF switching in Stateful and Stateless and the result averages to only 31 cycles. Note that above overhead is for a whole batch, meaning that the per-packet overhead with default batch size (32) is only ∼4 cycles for each switching.

As a comparison, virtualization frameworks would be impeded by context switching, cache misses and TLB flushes. To the best of our knowledge, Quadrant [68] has the least overhead by pinning the NF instances on the chain to a single core, which results in a per-packet overhead of ∼110 cycles. In Stateless with 4 NFs, it will cost 530 more cycles than LemonNFV $((110-4)\times 5$ switching). As the per-packet latency of Stateless is ∼4000 cycles in LemonNFV, Quadrant is thus ∼13% slower in terms of the isolation overhead.

7 Related Work and Limitations

We discuss efforts that relate to or inspire LemonNFV.

NF development. FastClick [17] and BESS [30] can flexibly wire their elements/modules up to the SFC (in an off-line manner). However, those modules must be programmed with specified interfaces, *e.g.*, being inherited from certain base classes. To reduce the overhead between NFs, NetBricks [52] proposes to use a new language, Rust, with a set of domain-specific abstractions for building NFs. However, it is costly to re-implement all existing NFs using such new language. NFD [32] can generate NF code from a high-level behavior model. While this facilitates the porting task, NFD does not make heterogeneous NFs interoperable, and operators are still bound to a specific interface. Nethuns [18] advocates a socket-independent programming abstraction for NFs, trying to unifiying the packet I/O, but in turn proposing a new I/O.

Moreover, with LemonNFV, one can directly launch a NF with out-dated I/O, *e.g.*, official NetBricks with DPDK 17.08 [11], or develop a new NF with the simple packet I/O, *e.g.*, `libpcap`, and get the newest and fast I/O for free.

NF optimizations. OpenBox [19] and SNF [36] optimize the SFC by eliminating the redundancy logics in NFs. Metron [35] goes a step forward by offloading part of the merged logic to programmable switches. PacketMill [25] and Morpheus [50] compiles optimal data structures and control flows for NFs. These optimizations require inner logics of NFs, and thus cannot be borrowed by LemonNFV that views each NF as an opaque box. We see it as an inherent trade-off between the reusability and deep optimizations.

NF execution models. Virtualization approaches are eager to improve the performance [22, 33, 40, 51, 76]. Exploiting the parallelism of NFs is viewed to be a promising direction [64, 77]. However, according to its own results, the parallel approach (NFP [64]) is 16.5% faster than the non-parallel one (OpenNetVM [76]), but 26% slower than the RTC approach (BESS [30]). The factual gap should be larger because BESS has reached the wire-speed in that results. Quadrant [68] packages NFs into containers and schedules them in a single core to avoid the cache misses, which, however, would cause the thread context switch.

FastClick [17], BESS [30], OpenBox [19] and Metron [35] are pure consolidation approaches, which achieve high performance without isolation. NetBricks [52] is the first consolidation approach that considers the isolation. By reimplementing NFs with a safe language (Rust), NetBricks ensures each instruction cannot access the data beyond its legal scope. However, such operation in Rust is too strict, lowering ∼20% throughput compared to native consolidation approaches [52].

Intra-process isolation. Hodor [31], ERIM [66], EPK [29] and CubicleOS [59] leverage Intel PKU to isolate software modules inside the process. LemonNFV is inspired by them and is the first for using PKU to isolate NFs in SFC. However, NFs can change pkey privileges themselves and thus break the control flow integrity or subvert the hypervisor. We see this as the nature of PKU and can leverage methods discussed in existing works to harden PKU-based isolation [20].

Besides memory and fault isolation, previous work has also proposed and enforced packet isolation, which prevents NFs from accessing packets that don't belong to them. Specifically, an NF may tamper packets outside its batch since the packet pool is shared by all NFs, contradicting the isolation among them. To solve this, NetBricks [52] leverages safe language to disable pointer arithmetic, and Quadrant [68] copies packets to cast packets on memory private to each NF. Similarly, LemonNFV can either limit permission on the exact batch using pkeys, or simply copy packets to address this problem.

NF migration. NFV frameworks need to migrate NFs and their states to balance the load across the cores or machines. Previous literatures achieve this feature by adding migration APIs [27, 55], copying states with identical state structures [35], or using centralized state tables [16], all of which require significant code modification to NFs. LemonNFV addresses this need by migrating the LEMON that packages the NF logic and code together. However, each LEMON corresponds to a certain packet class, which means LemonNFV cannot migrate a specific flow [27], or balance the load for a single large (*i.e.*, non-splittable) flow [16].

Limitations. The design of LEMON and LemonNFV does not meet every possible situation in deployment. (1) NFs that do not run as a standalone process (*e.g.*, NFs based on eBPF [3] or partially offloaded to hardware [35]) are not supported due to their fundamental deviation from a LEMON's execution model. (2) LemonNFV enables fast inter-server migration by disabling ASLR, which might be exploited by buffer overflow. Nevertheless, the operator can still migrate the whole LemonNFV process with ASLR enabled by leveraging checkpoint/restore methods (*e.g.*, `CRIU` [2]), when consolidating untrusted NFs.

LemonNFV requires recompiling NFs from the source code, which is not always available for off-the-shelf NFs. However, we believe it does not compromise much practicality of LemonNFV because the recompilation (1) still uses the standard compiler (*e.g.*, `gcc`), not raising concerns of security and inconsistency, and (2) only recompiles the main program, meaning that the original dependencies can be reused. These facts help the NF vendors to re-publish their NFs as LEMONs with a simple recompilation, and the users can directly plug them into LemonNFV.

We emphasize that LemonNFV does not aim to address *all* challenges in NFV. Instead, it tries to shed the light for the NFV world by enabling the ability of heterogeneous NF interoperation. On such basis, approaches like centralized state management [70], NF fault recovery [62], load balancing [16, 58], VNF placement and orchestration [24, 26], *etc*, could be complementary to LemonNFV.

8 Conclusion

We presented LemonNFV, a novel NFV framework that consolidates the heterogeneous NFs without code modification. We demonstrated the practicality of LemonNFV with 5 real NFs, and evaluated LemonNFV by comparing with state-of-the-arts. The results showed that LemonNFV outperforms the state-of-the-art virtualization approach by 1.9–2.4×, while only sacrifices 0.7–4.3% performance for isolation compared to the ideal consolidation framework.

Acknowledgements. We would like to thank the anonymous NSDI reviewers and our shepherd Dejan Kostic for their valuable feedback. Peng Zhang is the corresponding author. This work is partially supported by the National Key Research and Development Program of China (No. 2022YFB2901403) and the National Natural Science Foundation of China (No. 62172323 and No. 62272382).

References

[1] Address space layout randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[2] Checkpoint/restore in userspace. https://criu.org/.

[3] eBPF - Introduction, Tutorials & Community Resources. https://ebpf.io/.

[4] Service Function Chaining Use Cases In Data Centers. https://datatracker.ietf.org/doc/html/draft-ietf-sfc-dc-use-cases-06, 2017.

[5] DPDK. http://www.dpdk.org/, 2018.

[6] Libnids. http://libnids.sourceforge.net/, 2018.

[7] libpcap. http://www.tcpdump.org/, 2018.

[8] mm: Ability to monitor task memory changes (v3). https://lwn.net/Articles/546966/, 2018.

[9] nDPI. https://bit.ly/3ITdis5, 2018.

[10] Snort. https://www.snort.org/, 2018.

[11] NetSys/NetBricks. https://github.com/NetSys/NetBricks, 2019.

[12] The vMotion Process Under the Hood. https://blogs.vmware.com/vsphere/2019/07/the-vmotion-process-under-the-hood.html, 2019.

[13] Passive Real-time Asset Detection System. https://github.com/gamelinux/prads, 2020.

[14] Memory Protection Keys - The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/core-api/protection-keys.html, 2021.

[15] Single Root IO Virtualization (SR-IOV) - Mellanox. https://docs.mellanox.com/pages/viewpage.action?pageId=47033949, 2021.

[16] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. RSS++: Load and State-Aware Receive Side Scaling. In *ACM CoNEXT*, 2019.

[17] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast Userspace Packet Processing. In *ACM/IEEE ANCS*, 2015.

[18] Nicola Bonelli, Fabio Del Vigna, Alessandra Fais, Giuseppe Lettieri, and Gregorio Procissi. Programming socket-independent network functions with nethuns. *SIGCOMM Comput. Commun. Rev.*, 52(2):35–48, 2022.

[19] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *ACM SIGCOMM*, 2016.

[20] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. Pku pitfalls: Attacks on pku-based memory isolation systems. In *USENIX Security*, 2020.

[21] R. Cziva and D. P. Pezaros. Container Network Functions: Bringing NFV to the Network Edge. *IEEE Communications Magazine*, 55(6):24–31, 2017.

[22] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM SOSP*, 2009.

[23] Mohamed Esam Elsaid, Hazem M Abbas, and Christoph Meinel. Virtual machines pre-copy live migration cost modeling and prediction: a survey. *Distributed and Parallel Databases*, pages 1–34, 2021.

[24] Mehmet Ersue. Etsi nfv management and orchestration - an overview. https://www.ietf.org/proceedings/88/slides/slides-88-opsawg-6.pdf, 2013.

[25] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward per-Core 100-Gbps Networking. In *ACM ASPLOS*, 2021.

[26] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. *Technical Report arXiv:1305.0209, 2013*, 2013.

[27] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In *ACM SIGCOMM*, 2014.

[28] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. Packet order matters! improving application performance by deliberately delaying packets. In *USENIX NSDI*, 2022.

[29] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. EPK: Scalable and efficient memory protection keys. In *USENIX ATC*, 2022.

[30] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, 2015.

[31] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX ATC*, 2019.

[32] Hongyi Huang, Wenfei Wu, Yongchao He, Bangwen Deng, Ying Zhang, Yongqiang Xiong, Guo Chen, Yong Cui, and Peng Cheng. NFD: Using Behavior Models to Develop Cross-Platform Network Functions. In *IEEE INFOCOM*, 2021.

[33] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *USENIX NSDI*, 2014.

[34] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *USENIX NSDI*, 2017.

[35] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *USENIX NSDI*, 2018.

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

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

[38] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *ACM EuroSys*, 2017.

[39] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *ACM EuroSys*, 2021.

[40] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *ACM SIGCOMM*, 2017.

[41] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. Programming Network Stack for Middleboxes with Rubik. In *USENIX NSDI*, 2021.

[42] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K.K Ramakrishnan, and Timothy Wood. Microboxes: High Performance NFV with Customizable, Asynchronous TCP Stacks and Dynamic Subscriptions. In *ACM SIGCOMM*, 2018.

[43] Guyue Liu, Hugo Sadok, Anne Kohlbrenner, Bryan Parno, Vyas Sekar, and Justine Sherry. Don't yank my chain: Auditable NF service chaining. In *USENIX NSDI*, 2021.

[44] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. Performance and energy modeling for live migration of virtual machines. In *ACM HPDC*, 2011.

[45] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *ACM CCS*, 2015.

[46] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-In-Time summoning of unikernels. In *USENIX NSDI*, 2015.

[47] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than Your Container. In *ACM SOSP*, 2017.

[48] Antonis Manousis, Rahul Anand Sharma, Vyas Sekar, and Justine Sherry. Contention-aware performance prediction for virtualized network functions. In *ACM SIGCOMM*, 2020.

[49] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the art of network function virtualization. In *USENIX NSDI*, 2014.

[50] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. Domain Specific Run Time Optimization for Software Data Planes. In *ACM ASPLOS*, 2022.

[51] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A Framework for NFV Applications. In *ACM SOSP*, 2015.

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

[53] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX ATC*, 2019.

[54] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding network functions in the cloud. In *USNIEX NSDI*, 2018.

[55] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *USENIX NSDI*, 2013.

[56] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. Fine-Grained isolation for scalable, dynamic, multi-tenant edge clouds. In *USENIX ATC*, 2020.

[57] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*, 2012.

[58] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *ACM APNet*, 2019.

[59] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *ACM ASPLOS*, 2021.

[60] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *USENIX NSDI*, 2012.

[61] Junxian Shen, Heng Yu, Zhilong Zheng, Chen Sun, Mingwei Xu, and Jilong Wang. Serpens: A high-performance serverless platform for nfv. In *IEEE/ACM IWQoS*, 2020.

[62] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-recovery for middleboxes. In *ACM SIGCOMM*, 2015.

[63] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. Snf: Serverless network functions. In *ACM SoCC*, 2020.

[64] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. NFP: Enabling Network Function Parallelism in NFV. In *ACM SIGCOMM*, 2017.

[65] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *USENIX NSDI*, 2018.

[66] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient in-Process Isolation with Protection Keys (MPK). In *USENIX Security*, 2019.

[67] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *ACM SOSP*, 1993.

[68] Jianfeng Wang, Tamás Lévai, Zhuojin Li, Marcos A. M. Vieira, Ramesh Govindan, and Barath Raghavan. Quadrant: A cloud-deployable nf virtualization platform. In *ACM SoCC*, 2022.

[69] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as Processes. In *ACM SoCC*, 2018.

[70] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *USENIX NSDI*, 2018.

[71] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

[72] Farnaz Yousefi, Anubhavnidhi Abhashkumar, Kausik Subramanian, Kartik Hans, Soudeh Ghorbani, and Aditya Akella. Liveness verification of stateful network functions. In *USENIX NSDI*, 2020.

[73] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. Verifying software network functions with no verification expertise. In *ACM SOSP*, 2019.

[74] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. A formally verified nat. In *ACM SIGCOMM*, 2017.

[75] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *USENIX NSDI*, 2020.

[76] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. OpenNetVM: A Platform for High Performance Network Service Chains. In *ACM HotMiddlebox*, 2016.

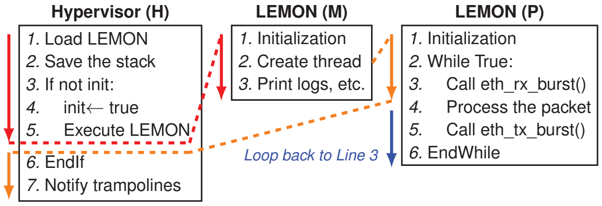
[77] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. ParaBox: Exploiting Parallelism for Virtual Network Functions in Service Chaining. In *ACM SOSR*, 2017.

[78] Peng Zheng, Arvind Narayanan, and Zhi-Li Zhang. A closer look at NFV execution models. In *ACM APNet*, 2019.

Appendix A Multi-Threaded LEMON Scheduling

For a single-threaded LEMON, the hypervisor would initialize it, and the working thread will continue executing it, as shown in Figure 4b. However, a LEMON could create its own threads, which, except for the packet processing threads, could include threads for event logging or user input communication. In the following, we present how LemonNFV cooperates with the multi-threaded LEMONs.

Taking Figure 8 as an example, a LemonNFV process has a hypervisor thread (T_h, red path), and an RTC working thread



```
Hypervisor (H)
1. Load LEMON
2. Save the stack
3. If not init:
4.     init← true
5.     Execute LEMON
6. EndIf
7. Notify trampolines
```

```
LEMON (M)
1. Initialization
2. Create thread
3. Print logs, etc.
```

```
LEMON (P)
1. Initialization
2. While True:
3.     Call eth_rx_burst()
4.     Process the packet
5.     Call eth_tx_burst()
6. EndWhile
```

Figure 8: Cooperating with multi-threaded LEMONs. The red path is the (original) hypervisor thread, and the orange path is the newly created packet processing thread. After the LEMON is initialized, the red thread becomes its main thread, and the orange thread becomes the hypervisor thread.

(T_r, blue path). Assume a new LEMON M1 is being loaded into the SFC, which creates a new thread for packet processing (T_p, orange path), and manages that thread in its main thread (T_m). The hypervisor would first execute the `main` function in T_h, that is, $T_m = T_h$ (Line 5 in H). After creating T_p in T_h (Line 2 in M), the execution in T_p would eventually call the virtualized I/O for the very first time, *i.e.*, the initialization part (Line 3 in P). Then the execution would be switched into the hypervisor stack (Line 6 in H). Note that since T_p is executing the hypervisor logic, this thread now actually plays the role of hypervisor thread. Next, T_p (the new hypervisor thread) would notify T_r that a LEMON is ready to be attached into the SFC (Line 7 in H), and T_r will switch the stack into the packet processing logics of this LEMON next time received the packets (Line 4 in P).

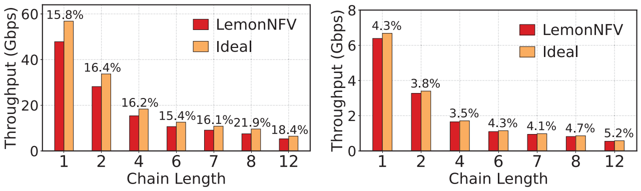
In sum, after loading a LEMON, T_h becomes the management thread of the new LEMON, T_p becomes the new hypervisor thread, and T_r is still the RTC working thread. Note that the LEMON can also use T_m as the packet processing thread, and create a new thread for management. In this case, T_m (T_h) will still be the hypervisor thread. The key here is that there is only one packet processing thread for each LEMON, so the virtualized I/O calls in that thread would eventually guide the hypervisor and trampolines to properly handle it.

Appendix B Dirty Pages Tracking in Inter-Process Migration

In the case of VM, the hypervisor can migrate the running VM to another host without halting it, namely live migration. The key is to capture the dirty pages by changing the accessing rights of the page table entries. For example, Xen implements a shadow page table to the original VM page table [44]. When the hypervisor decides to track memory modifications, the internal page table of the VM is transparently set to read only. That is, memory writes will not trigger a fault, but propagate to the shadow page table, recording the dirty pages during pre-copy stage in the hypervisor. vMotion in VMWare takes a similar approach [12].

Although each LEMON does not have a shadow or an isolated page table, its memory region has explicit boundary. As a result, it is possible to adopt the above method by directly changing the accessing rights of the page table inside a LEMON. To be specific, when migrating a LEMON, LemonNFV first blocks the LEMON and changes the permission of the LEMON's memory to read only. Then, all the memory writes to the LEMON will trigger a `SIGSEGV` signal. A pre-registered signal handler would capture this signal, record the address to be written (which pollutes the page), and grant write permission to the corresponding page. Upon completion of the pre-copy stage (*i.e.*, phase -1 in §4.4), the LEMON is set to read only again, waiting for the next iteration.

Experiments on `libnids` [6], prads [13] and nDPI [9] with real traffic show that after a few iterations, the number of the modified pages eventually converge to 13, 7 and 4, respectively (4KB for each page), which are far lower than the stopping condition of iteration in Xen (<50 pages) and vMotion (<16MB) [23], meaning that the factual migration would only halt the LEMON for a neglected moment.



(a) Lightweight (b) Heavy

Figure 9: Performance comparison of differently-loaded NFs between their LemonNFV and vanilla DPDK version.

Appendix C Microbenchmark

In this section we present some additional microbenchmarks of LemonNFV.

As in §6.3, the overhead of LemonNFV is irrelevant to the NF logic, contrast to array bound checking. It's then worth discussing how this overhead would impact the performance, under various workload and SFC length.

We prepare a light NF (∼200 cycles per packet) and a heavy NF (∼2300 cycles per packet), packing them as LEMON as well as porting them to DPDK. Under LemonNFV, variable number of identical LEMONs are chained together, while the DPDK version repetitively invokes the packet processing function until chain length is met. Since the vanilla DPDK version does not introduce additional overhead, it is referred to as 'Ideal' later as a baseline.

Figure 9 presents the end-to-end throughput under variable chain length and NF workload. We have the following observations and analysis for the results. First, the performance gap between LemonNFV and Ideal is large when the NF is lightweight (17.1% on average across all chain length), but becomes much less significant when the NF is heavy (4.3% on average). This corresponds to our analysis in §6.3 that the overhead of LEMON switching is irrelevant from NF logic. Since real-world NFs are generally more complex and time-consuming, LemonNFV would be more preferable than isolation methods based on array bound checking.

Second, given the fixed switching overhead, the performance gap should be stable when the chain length increases. This is also verified in Figure 9 when the chain length is under 7. However, the slowdown of LemonNFV grows when the chain length continues to increase. For example, the average slowdown of lightweight NF rises from 16.0% to 20.2% when the SFC is longer than 8. We believe that this is mainly due to cache contention of NFs co-locating on the same core [48,65]. Compared with virtualization based systems that often run NFs on separated cores, RTC scheduling will be more likely to drain cache and cause performance degradation [78]. We consider it as a feature of RTC and leave better profiling and optimizing cache performance as our future work.