Verifying Software Network Functions with No Verification Expertise

Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa
Katerina Argyraki, George Candea

School of Computer & Communication Sciences
EPFL, Switzerland

Abstract

We present the design and implementation of Vigor, a software stack and toolchain for building and running software network middleboxes that are guaranteed to be correct, while preserving competitive performance and developer productivity. Developers write the core of the middlebox—the network function—in C, on top of a standard packet-processing framework, putting persistent state in data structures from Vigor's library; the Vigor toolchain then automatically verifies that the resulting software stack correctly implements a specification, which is written in Python.

Vigor has three key features: network function (NF) developers need no verification expertise, and the verification process does not require their assistance (*push-button* verification); the entire software stack is verified, down to the hardware (*full-stack* verification); and verification can be done in a *pay-as-you-go* manner, i.e., instead of investing upfront a lot of time in writing and verifying a complete specification, one can specify one-off properties in a few lines of Python and verify them without concern for the rest.

We developed five representative NFs—a NAT, a Maglev load balancer, a MAC-learning bridge, a firewall, and a traffic policer—and verified with Vigor that they satisfy standards-derived specifications, are memory-safe, and do not crash or hang. We show that they provide competitive performance.

The Vigor framework is available at http://vigor.epfl.ch.

1 Introduction

Over the past decade, there has been a migration from ASIC-based to software-based implementations of middleboxes, and this brings both benefits and drawbacks. The main benefits are shorter development cycles and the flexibility to deploy network functionality on demand, which is an enabler of network function virtualization [20]. The main drawback is the lack of reliability that is characteristic of today's software: bugs, unpredictable behavior, and security vulnerabilities.

This work focuses on the data plane of software middleboxes, i.e., the code that implements the core packet-processing functionality of the middlebox and is performance-critical; we refer to such a piece of code as a "network function" (NF). A middlebox typically comprises other components too, such as a control plane or a web interface, which have a different reliability/performance trade-off from NF code.

The goal of this work is to enable the development of software network functions (NFs) that are guaranteed to be semantically correct while offering competitive performance and preserving developer productivity. This is different from what is typically called "network verification," which guarantees that a certain network behaves correctly given its topology, configuration, and NF *models* [17, 18, 23–25, 29, 30, 43, 47, 49]. Instead, this work is about "NF verification," which guarantees that a certain NF *implementation* behaves correctly [12]. Since we want to offer competitive performance and preserve developer productivity, we target NFs implemented in C on top of a standard I/O framework like the Data Plane Development Kit (DPDK) [14], because this is how high-performance NFs are developed today.

Recent work presented VigNAT, a Network Address Translator (NAT) that is guaranteed to be semantically correct and memory-safe [50]. VigNAT is split into a stateful and a stateless part. The former consists of carefully written data structures that hold all the NF's state that persists across packets; these data structures can be thought of as "components." The latter encodes the NAT logic itself, in essence "wiring" together the components to achieve the desired functionality: parsing the packets, looking up and manipulating NF state, modifying the packets, etc. The stateless and stateful parts pose different challenges to verification, so the two parts are verified using different verification techniques. The VigNAT "validator" tool then stitches the two proofs—after cross-checking their assumptions—into a final proof of the NAT's correctness with respect to the NAT RFC 3022 [45].

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359647

VigNAT left three questions open:

(1) How to generalize to other NFs beyond a NAT? Applying the original approach to other NFs still requires experience with verification techniques, in order to encode the semantics of the state into a validator specific to that NF that can stitch together the two sides of the proof. Our interactions with industry suggest that requiring NF developers to understand how the underlying verification works is a show-stopper for the adoption of NF verification.

(2) What to do about the software layers underneath the NF? All NF verification work we are aware of (see §8) assumes that the I/O framework, the operating system, and the network device drivers are correct. This means that the resulting proofs are just as correct as the thousands or millions of lines of code running underneath the NF. Furthermore, a single misunderstanding about how a particular system call works could render the entire NF proof false.

(3) How to lower the cost of verifying NF correctness in the real world? Practitioners at major Internet companies tell us that the upfront cost of verifying NFs is prohibitive. Learning a new language to write specifications, investing multiple days in writing a full specification before being able to verify even the first line of code, writing lemmas that a theorem prover can understand, figuring out whether a proof is failing because of an error in the spec or an error in the code, and so on are all reasons why verification is perceived by modern-day developers as an undue burden.

To address these three challenges, we developed Vigor, a framework for building and running NFs. Vigor takes as input an NF implementation and a specification that the implementation must satisfy, and it automatically produces either a proof that the implementation satisfies the spec, or a counter example. We do not reinvent the wheel: we start from the VigNAT approach and toolchain, and we change them in three ways: (1) We generalize to arbitrary NFs and remove the need for developers to "know verification." A key element that enables this is the level of abstraction of the NF spec and a technique to bridge the spec to the implementation. (2) We verify the entire software stack, reducing the trusted computing base (TCB) to the hardware, the clang and GCC compilers, a small piece of OS startup code, and our verification toolchain. A key element that enables this is a domain-specific operating system we built and a mechanism that verifies just those parts of the stack that are required by the NF in question. Since we expect NFs to run on dedicated machines, either physical or virtual, using a custom OS is reasonable. (3) We provide "pay-as-you-go" verification, a mode that reduces the cost of using verification in practice, both when developing and deploying NFs.

We present five NFs that we verified with Vigor—a NAT, a Maglev load balancer, a MAC-learning bridge, a traffic policer, and a stateful firewall. We show that each performs on par with standard, functionally equivalent, non-verified NFs that use the same high-performance packet I/O framework.

In the rest of the paper, we give an overview of Vigor (§2), then describe our three contributions: push-button NF verification (§3), full-stack NF verification (§4), and pay-as-you-go NF verification (§5). We evaluate our Vigor prototype (§6), discuss the limitations of our approach (§7), and close with related work (§8) and conclusions (§9).

2 Overview

We now provide an overview of Vigor: its intended audience and use, what kind of specifications it verifies, and a brief summary of how it works.

Audience and Use

We designed Vigor with four categories of audience in mind: (1) *NF operators* are network operators who deploy NFs in their networks. (2) *NF developers* write NF code in C on top of standard I/O frameworks like DPDK [14]. They are competent in popular programming languages, like C and Python, but are not (and do not wish to become) verification experts. For instance, they would not have the time or experience to productively use an interactive theorem prover like Coq or Isabelle, or to write specifications in first-order logic. (3) *Standardization bodies* are the entities that define what NFs ought to do. For example, they write the RFCs that define IP forwarding or Network Address Translation. They are not verification experts either, but they may be willing to write a specification for the NF they are standardizing, as long as it can be done in an accessible language. (4) *Vigor contributors* are the maintainers of the Vigor stack and toolchain. They have plenty of verification expertise and enjoy using it.

The Vigor contributors develop and maintain a library of formally verified data structures (libVig in Fig. 1), a small NF-specific operating system, the Vigor toolchain, and models of newly supported NICs (i.e., simplified specifications of NIC behavior to be used for verification). They also provide a standard packet I/O framework (in our prototype, DPDK [14]) and NIC drivers, all of which are off-the-shelf but slightly patched. These patches are minor (~100 LOC added/removed) and include fixes for bugs that prevent correctness proofs from succeeding (see §6.2) plus adjustments to make DPDK verifiable. They do not change the public API, and porting them to a new version of DPDK takes ~15 minutes.

Standardization bodies write full specifications of NF functionality and publish them together with the corresponding RFCs, IEEE standards, etc. Their goal is to reduce the confusion inherent to natural language and enable testing and mechanical verification of the standard's implementations.

NF developers write NF implementations in C, using libVig to store all state that persists across packets, and the I/O framework to send/receive packets. For each release of an NF, developers use Vigor in a push-button manner (§3) to verify the implementation against an RFC-derived specification. While developing the code, they use Vigor in a pay-as-you-go

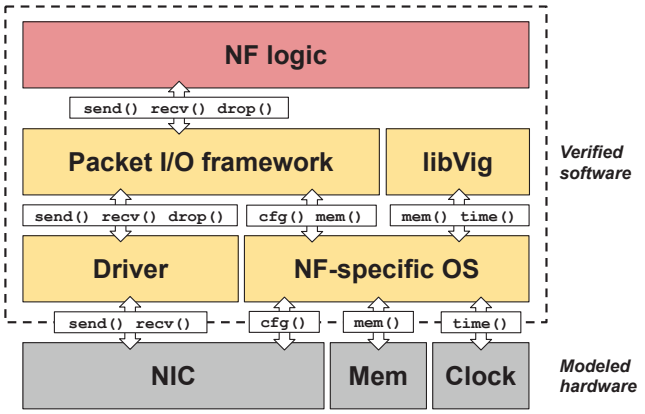

Figure 1. The Vigor stack for running NFs.

manner (§5) to check one-off properties (see "Specification vs. Implementation" below) and thereby better understand their code, debug it, and converge onto a correct implementation more quickly. Neither mode requires verification expertise, but it does require NF developers to keep Vigor's limitations (§7) in mind. Fig. 2 illustrates this workflow.

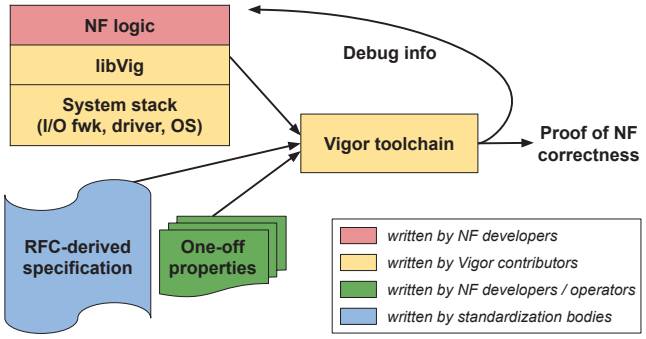

Figure 2. Vigor workflow: who writes what.

Similarly, NF operators can use Vigor to verify the NF implementations they deploy in their networks, either against full specs or in a pay-as-you-go manner (e.g., to check whether the NF fulfills custom, "in-house" expectations). They can also combine Vigor with network verification tools to reason about their network without relying on unverified NF models.

Specification vs. Implementation

Vigor verifies low-level properties (e.g., memory safety, crash freedom, hang freedom, absence of undefined behavior) and semantic properties, and it supports two types of semantic specifications: (1) *Full NF specs* define the entire semantics of an NF. They are written by standardization bodies. (2) *One-off properties* specify particular aspects of an NF implementation's behavior that perhaps are not even part of the RFC, such as "Could this load balancer ever send connections to those high-per-connection-cost nodes when any of the cheap ones over here are under-utilized?" NF developers and operators have little interest in formulating an entire spec just to verify that such a one-off property holds in all possible circumstances. This ad-hoc way of verifying properties is particularly suitable when one is trying to learn things about the NF implementation in the context of its deployment rather than prove that it is correct in an absolute sense.

Specs are written in a dialect of Python, specified in [48]. Fig. 3 shows an example. We believe that using Python makes writing specs easier and friendlier for all three categories of Vigor users. Our dialect uses a small subset of Python syntax.

When choosing a spec language, a key decision is the level of abstraction; in Vigor, specs are written in terms of the abstractions exposed by the libVig API. For example, libVig exposes the abstraction of a map with automatic entry expiration. The spec in Fig. 3 uses this abstraction (`macTable`) to define the functionality of a MAC-learning bridge, which uses the map to associate MAC addresses with NIC IDs. An NF developer implementing this spec must also use the map abstraction in their implementation.

```
1  from state import macTable
2
3  macTable.expireOlder(now - EXP_TIME)
4  macTable[pkt.src_mac] = (port, now)
5
6  if pkt.dst_mac in macTable:
7    out_port = macTable[pkt.dst_mac]
8    if out_port == port:
9      return DROP
10   else:
11     return out_port, pkt
12 else:
13   return BROADCAST, pkt
```

Figure 3. Part of a spec for a MAC-learning bridge.

Currently, libVig provides a small number of data structures that we consider sufficient for writing most NFs: a map, a consistent hashtable, a vector (all three having the ability to automatically expire entries), a longest-prefix-match table, a number range manager (e.g., used for port allocation), and a packet queue. libVig also provides primitives for reading system time, getting random numbers, and parsing packet headers. With some further engineering, verified crypto primitives [52] and verified regexp parsing primitives [2] can be added to complete the library. As shown in §6.4, it is reasonable to expect that such an expanded version of libVig would be sufficient for building most NFs. Compared to VigNAT's data-structure library, libVig required some reorganization, reimplementation, and the addition of new primitives.

Under the Covers

The Vigor verification process has two components: (1) *libVig verification* – For each libVig function, the Vigor contributors

write a contract (i.e., pre- and post-conditions) and proves using a theorem prover that the function implementation satisfies this contract (i.e., given the pre-conditions, an invocation of the function leads to the post-conditions). More details appear in §3. (2) *NF stack verification* – Vigor takes as input the code of the entire software stack, the contracts of the libVig functions called by the NF, and a spec to verify. It then uses a combination of exhaustive symbolic execution and theorem proving to prove that the NF, running on the given stack, does or does not satisfy the spec for all possible execution scenarios. If the proof fails, Vigor provides debug information to help pinpoint the cause.

The next three sections describe the contributions of this paper in more detail: how Vigor achieves push-button verification (§3), how it verifies the full NF stack (§4), and how it provides pay-as-you-go verification to developers and operators (§5). Our approach leverages aspects specific to the domain of NFs and thus does not extend to general-purpose software; however, we conjecture (§7) that it can generalize to other domains of systems software beyond NFs.

3 Push-button Verification

In push-button verification, the developers write their code (and potentially also a specification), then "push a button," and out comes the proof. NF developers neither write proofs nor guide the theorem prover with lemmas or code annotations written in a special proof language.

Vigor inherits from VigNAT the split NF architecture and builds on that approach to verification. There are four key steps, illustrated in Fig. 4: (1) Use exhaustive symbolic execution of the stateless NF code to obtain all live paths through the code. (2) Convert the resulting symbolic execution traces to C programs, each one representing one path through the stateless code. (3) Annotate each trace with lemmas corresponding to the NF specification. (4) Validate each annotated trace, using the pre- and post-conditions from the libVig API contracts, i.e., verify (using a theorem prover) that the stateless code uses the libVig API correctly and that the resulting post-conditions imply the desired specification. Once this process completes, the theorem prover has formally verified that the NF is a correct implementation of the specification. libVig is already verified and provides formal contracts for its API.

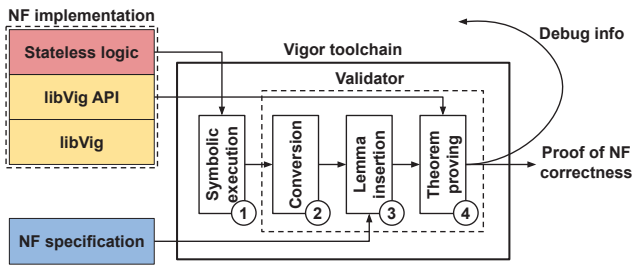

Figure 4. The Vigor verification process.

For step (1), we reuse the machinery developed for VigNAT. The main challenge is path explosion [6], in particular given that an NF is almost always structured as an event loop, which gives rise to an infinite number of paths. To tackle this and other loop-related challenges, VigNAT employs techniques like havoc-ing [50]. Another problem is that the data-structure implementations use features that make symbolic execution impractical (e.g., raw pointers), so VigNAT uses models for step (1). These models must then be validated a posteriori, using lazy proofs [50], essentially verifying their correctness for the specific context of the target NF.

It is the Validator—steps (2), (3), and (4)—that we modified to enable push-button verification. The Validator primarily stitches together proofs of data structure code with symbolic traces of stateless NF code into a single proof. The VigNAT Validator was only able to fill in the proof steps for the particular sequences of calls to the data structures that are present in VigNAT. Furthermore, it assumes that the NF's state is keyed on a 5-tuple network-flow structure. This makes the Validator in the VigNAT approach NF-specific: a developer writing, say, an L2 bridge would have to manually adapt the Validator, and that requires verification expertise.

The challenge in making the Validator generic across NFs, even when libVig is shared by all NFs, is that each NF uses the libVig data structures differently. Our first approach to generalizing the Validator was to design heuristics that infer how each NF uses libVig and then generate corresponding support lemmas automatically. But, for each attempt, we always found a scenario in which our heuristics didn't work. This pushed us to think differently about the problem, i.e., find a way to side-step automatic lemma generation.

The key is to identify a "language" that enables efficient communication between spec writers/operators, developers, and verification tools. On the one hand, standardization bodies and NF operators use a certain vocabulary when they discuss NF functionality, and we should expect them to use the same vocabulary to write NF specifications. On the other hand, NF developers use the primitives provided by libVig to write NF implementations. So, to verify that an NF implementation honors an NF specification, the Vigor toolchain must correctly translate between (a) libVig primitives and (b) specification vocabulary. This can be hard, and we found that current verification tools cannot close non-trivial conceptual gaps between (a) and (b) without assistance from a human verification expert. Hence, if we want to achieve push-button verification today, the primitives used by NF specs ought to be at the same level of abstraction as those provided by libVig.

After reading RFCs, IEEE standards, and papers that specify NF functionality, we found that such a common language does exist, and it consists of a handful of popular data structures. For example, RFC 3022 [45] expresses NAT behavior in terms of a "translation table," and describes "mapping of tuples of type (local IP address, local port number) to tuples of

type (registered IP address, assigned port number)." According to [15], the Maglev load-balancer's "connection tracking table uses a fixed-size hash table mapping 5-tuple hash values of packets to backends." NF specs in general talk about network addresses, ports, flows, etc. and data structures that map these to relevant state—essentially the same primitives that developers use to write NF implementations.

We therefore designed libVig to provide primitives at a level of abstraction that enables not only NF developers to write NF implementations but also spec writers to naturally express NF functionality. In theory, writing specs with such primitives might seem inappropriate, because they seemingly would end up dictating implementation details. In the specific case of NFs, however, we find that specs *anyway* imply such implementation details, e.g., RFC 3022 essentially defines the NAT translation table as a map that translates between local and external IP addresses/ports. In fact, trying to define a NAT without using a map might even be confusing.

Using the same primitives in both the spec and the implementation reduces the conceptual gap between abstract state and implementation state. This makes the proofs simpler, since the theorem prover needs fewer lemmas to match libVig call sequences in the symbolic traces with abstract-state manipulations in the specs. In other words, instead of automating the human task of proving refinement from high-level abstract state to low-level implementation state, in Vigor we remove the need for doing it in the first place.

Vigor requires NF developers to explicitly link the persistent state in the NF implementation (expressed in terms of libVig primitives) with the corresponding abstract state in the NF specification. Establishing this link is easy for the developers, since abstract state is expressed using semantically similar primitives. For example, the spec in Fig. 3 references on line 1 a MAC table `macTable` (abstract state), while the corresponding implementation code in Fig. 5 references map `mac_table` (implementation state). The developer makes the correspondence between abstract and implementation state explicit via an `NF_EXPORT_STATE` macro, as on line 3 in Fig. 5. This statement informs Vigor that the map `mac_table` implements `macTable`, the MAC table in the spec.

To use libVig data structures, developers instantiate generic type templates from libVig. As part of this instantiation, they select parameters like key and value types, and convey information on bounds (e.g., max number of elements in a vector) and on forbidden values (e.g., disallowed values for an external device id). For example, the first two lines in Fig. 5 declare `mac_table` as an `emap` of total capacity `CAPACITY` with keys of type `ether_addr` and values of type 32-bit integer in the range `0...DEV_COUNT`. This is akin to type instantiation in languages that are more strongly typed than C—this does not require verification expertise and poses little burden on modern developers. These declarations provide hints to Vigor that help it reason about the code. For example, it uses the bounds information to both verify that the bounds

```
1 NF_STATE(mac_table, emap, CAPACITY,
2    ether_addr, uint32_t, 0, DEV_COUNT);
3 NF_EXPORT_STATE(mac_table, macTable);
4
5 int main(int argc, char *argv[]) {
6   while (1) {
7     uint8_t in_port;
8     packet_t pkt;
9
10    if (receive_packet(&in_port, &pkt)) {
11      expire_entries(mac_table, now - EXP_T);
12
13      if (map_has_key(mac_table,
14                      pkt.src_mac))
15        map_refresh(mac_table, pkt.src_mac, now);
16
17      map_put(mac_table, pkt.src_mac, in_port);
18
19      if (map_has(mac_table, pkt.dst_mac)) {
20        uint8_t out_port = map_get(mac_table,
21                                   pkt.dst_mac);
22        if (out_port == in_port)
23          drop_packet(pkt);
24        else
25          forward_packet(out_port, pkt);
26      } else {
27        broadcast_packet(pkt);
28      }
29    }
30  }
31 }
```

Figure 5. Simple implementation of a MAC learning bridge.

are observed and that values can be safely used as an index into the array of NICs. Fig. 6 shows the API of the `emap` data structure in libVig. The full libVig API appears in [48].

```
1 void expire_entries(struct EMap* map, time_t t);
2 void map_refresh(struct EMap* map, void* key,
3                  time_t t);
4 void map_put(struct EMap* map, void* key,
5              value_t value);
6 bool map_has(struct EMap* map, void* key);
7 value_t map_get(struct EMap* map, void* key);
```

Figure 6. `emap`: libVig's map with automatic expiration.

The information in these template instantiations enables the Validator to automatically produce lemmas for annotating the symbolic traces, and to combine them into a proof—something that had to be done by hand in VigNAT. For example, the Validator uses the declarations in lines 1–3 of Fig. 5 to automatically compose a loop invariant that describes the state that persists across the event loop's iterations, and this invariant includes the user-provided constraints on the table

values. The invariant may or may not verify in the end, depending on the correctness of the implementation, but the hard part of coming up with the invariant and ultimately the proof can now be done automatically by Vigor.

In summary, the turning point for enabling push-button verification in Vigor was choosing a common level of abstraction for state used by NF specifications and NF implementations. This choice makes it easier for Vigor to reason about the correspondence between abstract state and implementation state, which enables it to automatically generate the supporting lemmas needed for verification. This automation, in turn, means that NF developers no longer need to modify the toolchain for each new NF, and so Vigor generalizes across NFs.

Our experience developing five NFs suggests that this level of abstraction works well for both specifying and implementing NFs. In §6.4, we attempt to partly quantify why.

4 Full-stack Verification

To make the vision of NF virtualization come true, we argue for formal guarantees of correctness that cover middlebox software in its entirety. This work focuses on the data plane software stack—the NF and its libraries, the underlying packet-processing framework, the operating system, and the device drivers—but our argument also applies to the control plane as well as any compiler used to generate middlebox code. When deploying hardware middleboxes, operators only have to trust hardware, not software. For them to make the switch to software NFs with peace of mind, they need strong guarantees that hardware (and humans) will be the only source of failures.

Merely verifying the NF code is not sufficient, especially when using a kernel-bypass framework like DPDK, because a bug or security vulnerability in the framework can compromise the entire machine running the middlebox. We have direct evidence that blindly trusting the layers underneath the NF is unsafe: during verification, we found bugs in both DPDK and an Intel NIC driver (see §6.2).

The following observation enables Vigor to verify the full NF stack: even when an NF stack is too big to be verified efficiently, the fraction of the stack used to accomplish a specific purpose is likely small. In the case of the NFs we wrote, only ~3.5% of DPDK's roughly 65 KLOC end up ever running, respectively ~18% of the Intel 82599ES NIC driver's 24 KLOC. The rest of the code is dead as far as the execution of that NF is concerned.

It therefore makes sense to use exhaustive symbolic execution of the NF and to not model *all layers* below the NF, as done in VigNAT and most NF verification work, but rather model only the *hardware layer*. The NF calls the NIC driver directly and uses only simple DPDK utility functions that can be symbolically executed. Most of DPDK's functionality is invoked only during initialization, which can be executed concretely (i.e., not with symbolic input) during symbolic execution. The NIC driver code is a sequence of reads and writes to hardware, with little branching.

Unfortunately, even this observation does not allow the straightforward exhaustive symbolic execution approach to scale when running on a commodity operating system. Even though packet processing frameworks like DPDK bypass the kernel, so the operating system gets out of the way shortly after setup, it is still the case that the OS could affect the NF any time after setup (e.g., by preempting the NF, unmapping its memory pages, reconfiguring the way the NF talks to hardware). This is because a large part of the commodity operating system remains live, even if it is not directly invoked by the NF or the packet processing framework. So we would need to prove isolation between the OS and the packet processing framework, and doing this is hard. Using symbolic execution would result in path explosion, and interactively verifying Linux is not practical. For reference, tools like VeriFast [22] require ~10 lines of annotations for each line of code, and the Linux kernel contains over 25 MLOC [19].

Our solution to this challenge is a small, custom operating system that can be automatically verified. We think of this as an NF-specific operating system, or NFOS. The NFOS performs the necessary setup for the packet I/O framework to take over: set up the CPU and memory, scan the PCI bus to find all available NICs, configure the NICs, and start the NF. Then it gets out of the way—and that we can prove formally. Beyond regular setup, the NFOS also provides timing information from the hardware to the NF (see Fig. 1). NFs use this information, for example, to expire TCP flows, MAC address entries, or to police IP traffic.

Being specifically meant to run NFs built with Vigor, there are many things the NFOS simply does not need to do. Given that Vigor formally verifies the NF, DPDK and the NIC driver to be crash-free and memory-safe, the NFOS need not provide any inter-process isolation. Kernel-bypass frameworks like DPDK often come with their own NIC drivers that use polling rather than interrupts. This means that the NFOS need not include device drivers or provide support for handling interrupts. Finally, given that NFs in Vigor are currently single-threaded (see §7 for a discussion), the NFOS does not need a scheduler. This lean design ensures that the NFOS can be symbolically executed in an exhaustive manner.

This makes it possible to achieve push-button verification of not only the NF but also DPDK, the NIC driver, and the operating system. In the case of the NFs we wrote, this results in symbolically executing on average ~6.5 KLOC of DPDK, OS, and driver code for each NF, with absolute certainty that the remainder of ~82.5 KLOC is dead and thus cannot influence correctness of this NF. Our use of exhaustive symbolic execution formally proves the complete separation between the part of the stack that is invoked by the specific NF and the part that is not. Verification then focuses automatically, for each NF, on the live code only. Of course, the part of the code that is dead may differ from NF to NF.

The only part of the NFOS that we do not symbolically execute is a piece of code that consists of 140 `x86` assembly instructions run during boot, 170 lines of C code to scan the PCI bus, and 40 lines of C code for a trivial memory allocator. Early startup code is typically excluded from OS verification due to a simple trade-off: one would need to formally define and/or model large amounts of hardware details in order to prove correct tiny, straightforward code fragments. In our case, we argue the code's correctness with detailed code comments that explain why each instruction and LOC has the intended effects [48]. Our correctness argument was vetted by several developers other than the NFOS author. However, to be conservative, we include this code in Vigor's TCB.

An alternative to our approach is to verify the packet processing framework, NIC drivers, and OS in isolation. For the OS and drivers, we could have used an approach similar to seL4 [26] or Hyperkernel [35]; this would require translating the OS interface contracts to stitch with our proofs, and would likely impose a performance penalty on the resulting NF. For DPDK, verifying the entire framework through exhaustive symbolic execution is impractical due to path explosion. Interactively verifying it with a theorem prover is not practical either, due to the human effort it entails, which would furthermore have to be repeated for every new release. Overall, we believe that allowing the NF code to implicitly indicate which parts of the stack it needs, and then let Vigor automatically verify just those, is a more practical approach.

In conclusion, Vigor's full-stack verification reduces the TCB to a level we expect NF operators to be comfortable with: the early startup code mentioned above, the hardware, the clang and GCC compilers, and the Vigor toolchain. This toolchain is composed of the KLEE symbolic executor [8], the VeriFast theorem prover [22], and the Vigor Validator. The Vigor stack currently uses the unverified GRUB boot loader, but it could be replaced with a formally verified one [11].

5 Pay-as-you-go Verification

Any verification task has a fixed cost and a variable cost, and pay-as-you-go verification is about lowering the fixed cost as much as possible. The fixed cost is dominated by writing a complete specification, which ends up including many things that are not needed for verifying the first line of code. In Vigor, developers need only write sufficient spec to cover the property they wish to verify. So it becomes possible to specify a single property, verify the NF against that property, fix any bugs, write another property, and so on. This feature of Vigor is inspired by [36], and our use of the term "pay as you go" was suggested to us by the author.

Vigor enables pay-as-you-go verification by changing the closed-world assumption present in VigNAT to an open-world assumption [36]. In logic, under a closed-world assumption, a true statement is always known to be true; as a result, a spec must include or imply all true properties of the NF code; any properties that are not part of, or implied by, the spec, are deemed to not hold. This imposes the need to write the entire spec before verifying any property of the system, and complete specs are often long and tedious, which discourages practitioners from writing them. Under an open-world assumption, however, what is not known to be true is simply unknown, not false. This removes the need for all knowledge about the NF's properties to be captured at once in a spec.

First, Vigor allows writing specs in a compositional way, by using references. The open-world assumption is amenable to an incremental build-up of knowledge about the NF's properties, and this matches well the modern software development process. For instance, a developer can write a spec saying "The bridge always learns the association between source MAC address and the port" (Fig. 7). Then, they can write another spec saying "The bridge forwards the frame according to the association in its MAC address table," and reference the previous specification (Fig. 8). Both Fig. 7 and Fig. 8 are sound specs, in the sense that they specify a bridge behavior that must hold for *all* possible executions. This compositional approach dovetails with the monotonicity of first-order separation logic used in our toolchain: adding new information cannot falsify a previous conclusion.

```
1 from state import macTable
2 macTable.expireOlder(now - EXP_TIME)
3 macTable[pkt.src_mac] = (port, now)
```

Figure 7. Example property spec: after expiring state entries, the bridge learns the source MAC address of a frame.

```
1 import bridge_learn
2 if pkt.dst_mac in macTable:
3   out_port = macTable[pkt.dst_mac]
4   if out_port == port:
5     return DROP
6   else:
7     return out_port, pkt
8 else:
9   return BROADCAST, pkt
```

Figure 8. Composed specification: the bridge forwards a frame according to the entry in the MAC address table.

Second, in Vigor, one-off properties can be described with small specs that exclude, via the Python `pass` keyword, knowledge about any number of NF properties that remain unspecified. For instance, if we wanted to focus only on the broadcast case shown in Fig. 8, we would use `pass` for other behaviors, as shown in Fig. 9. Here, Vigor only checks that, for unknown MAC addresses, the frame will indeed be broadcast, ignoring the I/O and state changes for known MAC addresses. This property allows the developer to focus on one aspect of NF behavior at a time, while still getting formal guarantees.

```
1 import bridge_learn
2 if pkt.dst_mac in macTable:
3   pass
4 else:
5   return BROADCAST, pkt
```

Figure 9. Broadcast case in the MAC-learning bridge spec.

One-off properties in Vigor are of three types: contained in the NF's official specification, implied by it, or independent of it. The properties in Fig. 7 and Fig. 8 are of the first type. The second type of properties can be fully derived from the spec, such as how an NF behaves in a certain configuration. For example, an operator might want to check properties that hold for a particular deployment scenario, such as "Does my load balancer ever send connections to the high-per-connection-cost nodes when any of the cheap nodes are under-utilized?"

Specification-independent properties relate to behaviors of the NF implementation that are outside the spec. For example, NAT middleboxes that correctly implement RFC 3022 are free to choose what to do with a non-SYN TCP packet that arrives from the internal network and does not correspond to an established flow—forwarding or dropping it are both correct behaviors. An operator may want to know whether the NF she is about to deploy would forward such spurious TCP packets to the external network. Perhaps she uses network verification tools, and none of the available NAT models capture the non-SYN behavior of this particular NAT. She would formulate this property as in Fig. 10, then ask Vigor to verify it.

```
1 from state import flowTable
2 if pkt.ip.next_proto_id == PROTOCOL_ID_TCP:
3   if SYN not in pkt.tcp.flags:
4     if pkt.flow not in flowTable:
5       return DROP
```

Figure 10. One-off property checking for non-SYN behavior.

In Vigor, it is possible not only to pick relevant parts of the specification but also to select which layers of the stack to verify. The NF developer can choose whether to verify only their NF code, or to include additional parts of the stack. Verifying more code provides more guarantees but takes more time. To allow the developer to pick which parts of the stack they want to verify, we wrote models for each layer in the stack. As we show in Table 3, depending on how much of the stack one wants to verify, the developer can speed up verification significantly. It makes sense for NF developers to use partial-stack verification during daily development, and full-stack verification before each release of the software.

Pay-as-you-go verification makes it less daunting to write standards as precise specifications. First, with pay-as-you-go, one can write specs incrementally, during development of a reference implementation for an NF by a standardization body. Second, even in the absence of an initial formalization of an RFC by a standardization body, development teams building NF implementations could collaboratively formalize the RFC as they write one-off properties to verify their code, and these properties can then be curated into a unified spec of the RFC. Third, since specifying properties requires only Python knowledge, network operators could write specifications that correspond to their own datacenter or network setup, which NF developers and vendors would likely find useful.

To complement pay-as-you-go verification, Vigor helps NF developers pinpoint whether a proof failure is the result of an erroneous spec or erroneous implementation ("Debug info" edge in Fig. 2). When verification fails, Vigor presents a counter-example for the proof, in the form of a piece of linear C code representing the execution that led to the property violation. For example, if Fig. 8 were missing the `DROP` case on lines 4–5, Vigor would produce the trace leading to line 23 in Fig. 5 and flag a mismatch. The developer can then use the counter-example to determine whether the spec is wrong or the code is buggy, and in the latter case fix the code.

The three aspects described in this section—one-off properties and composable specifications, verification by layers, and debugging by counter-example—enable an incremental form of push-button, full-stack verification. They enable developers and operators to get the benefits of formal verification quickly, with little upfront cost.

6 Evaluation

We now evaluate the main aspects of Vigor by answering four questions: (1) Does Vigor generalize? We show in §6.1 that the answer is yes: we developed five NFs covering a variety of representative NF functionality, and we push-button verified them with Vigor. (2) Does verification have tangible benefits? We discuss in §6.2 how Vigor helped prevent 5 bugs in NF code and discovered 9 bugs in DPDK and an Intel NIC driver. (3) Does verification come at the price of performance? We show in §6.3 that it does not: NFs verified with Vigor perform on-par with third-party alternatives. (4) Does verification come at the price of reduced productivity? We show in §6.4 that using Vigor is easy and can improve productivity by helping write correct code faster.

Our code, specifications, and proofs are available at [48]. This work received a "Results Replicated" badge from the Research Artifact Evaluation Committee based on tag `SOSP19AE`.

6.1 Does Vigor generalize?

We used Vigor to develop and verify the five NFs shown in Table 1. These implement several types of typical NF functionality (per-flow state, header rewriting, etc). Some notable NF types, such as intrusion detection systems, are missing,

because our current libVig prototype does not yet provide primitives for regular expression matching or cryptography (see §7 for a discussion). For each NF, we wrote a full specification of its behavior based on published standards and used Vigor to prove that the NF correctly implements that spec. §6.4 provides details on the effort involved and spec sizes.

Name	Description	Class of NFs
VigNAT	Network address translator	Per-flow state Header rewriting
VigBr	Eth bridge with MAC learning	Packet duplication
VigLB	Load balancer (implements Maglev[15] algo)	Per-flow state Consistent hashing
VigPol	Traffic policer (rate-limits traffic by source IP)	Per-flow state Fine-grained timing
VigFw	Firewall (blocks ext. connections)	Per-flow state

Table 1. The NFs we developed and verified with Vigor.

For every row in Table 1, all software is verified except for the Vigor toolchain, the GRUB boot loader and NFOS initialization code, and compilers. Table 2 shows the size in LOC of each layer of the Vigor stack that is verified. As explained in §4, the entire stack is mechanically verified, except for ~350 LOC of assembly and C, whose correctness is argued by hand [48]. We reiterate that we could replace GRUB with a formally verified boot loader [11] and thus eliminate it from the TCB.

Stack layer			Lines of code		
VigNAT	VigBr	VigLB	969	815	850
VigPol	VigFw		725	754	
libVig			1,674		
KLEE-uClibc (libc)			60,556		
DPDK			62,380		
Ixgbe Driver			24,211		
Operating system (NFOS)			1,958		

Table 2. Size of each layer in the Vigor stack.

Table 3 shows the time it takes to verify the NFs. We measured three scenarios: verifying just the NF code against the full spec, verifying the NF together with DPDK, driver, and libC, and finally adding NFOS to verify the entire software stack. The difference between verifying with or without NFOS is negligible ($\pm$ 20 sec), so we report the first and third scenarios only. Total verification time is the sum of the time for exhaustive symbolic execution to obtain the symbolic traces (columns 2 + 3) plus the time to validate all the traces (columns 4 + 5 multiplied by column 6). We report validation time as *# of traces* $\times$ *per-trace validation time* because validation is an "embarrassingly" parallel task, so total completion time depends linearly on the number of thread contexts available. The reported number of symbolic traces corresponds to code paths analyzed *after* the various optimizing analyses done by Vigor (see §3), such as loop havoc-ing; without these optimizations, the number of traces would be infinite.

Verification time is dominated by full-stack verification, in particular the trace validation step. VigLB has significantly higher per-trace validation time than the other NFs. This is partly because VigLB traces make more calls to libVig—unlike the other NFs, VigLB employs two different maps, one for flows and one for backends. VigLB traces also trigger several slow-path behaviors in VeriFast, as used by the Validator: it struggles to check that each call made by VigLB honors the pre-conditions in the corresponding contract. Still, trace validation completes in ~1.5 hours on our test machine.

NF	Symbolic execution time		# of traces			Per-trace validation time (avg)
	NF only	*with rest of stack*	*NF only*	*with rest of stack*		
VigNAT	7 sec	+8 min	54	+434	×	88 sec
VigBr	7 sec	+10 min	69	+542	×	80 sec
VigLB	23 sec	+26 min	146	+1,190	×	219 sec
VigPol	14 sec	+6 min	37	+272	×	82 sec
VigFw	6 sec	+7 min	43	+326	×	88 sec

Table 3. Verification statistics.

For the verification measurements, we used a setup consisting of Intel DPDK v.17.11 for the packet I/O, with the ixgbe driver for the Intel 82599ES NIC. We ran the verification on a dual-socket Intel Xeon Gold 6132 machine @ 2.6 GHz, providing a total of 28 cores (56 thread contexts). Full-stack verification consumed <700 GB of DRAM, and verifying just the NF took < 2 GB; the machine had 1.48 TB available. Each NF was configured with table sizes of 65,536 entries.

Verifying the NF code alone takes a few minutes on our machine, so it could be done regularly as part of continuous integration or in a post-commit hook. For most NFs, verifying the full software stack takes <1 hour on our machine, so doing it at least once per release cycle is reasonable. It is, however, possible to drastically speed up the validation phase through parallelization, since each trace can be validated independently from all others. Validating the traces on a cluster with hundreds or thousands of cores would lower the verification time of the full NF stack to minutes or seconds, making it practical to do after every commit.

In summary, we developed and verified five varied and representative NFs with Vigor, thus showing that the Vigor approach generalizes to multiple kinds of NFs. Verification time matches well the patterns of modern software development. We therefore conclude that Vigor can provide practical push-button, full-stack verification for NFs.

6.2 Does verification have tangible benefits?

One of formal verification's greatest promises is that it prevents *all* bugs from making it into released code. Since Vigor verifies both semantic properties and low-level properties like memory safety, it is able to identify both high-level bugs (e.g.,

Stack Layer	Bug Description	Vigor step
NF	Incorrect use of by-ref parameter	VA
	Missing checks for packet spoofing	VA
	Incorrect use of the IP header	VA
	Infinite loop in consistent hashing	LV
DPDK	Out-of-bounds array access	SE
	Incorrect use of *mmap()*	SE
	Incorrect use of *libnuma*	SE
ixgbe NIC driver	Out-of-order register write	SE
	Use of potentially invalid NIC register	SE
	Incorrect timing of register write	SE
	Incorrect writes to reserved bits	SE
	Write to unknown NIC registers	SE

Table 4. Bugs uncovered by Vigor, and the step in the Vigor workflow that discovered each bug: libVig verification (LV), symbolic execution (SE), or validation (VA).

packet-injection vulnerabilities) and low-level ones (e.g., incorrect writes to reserved bits). Table 4 shows some of the bugs Vigor found in our own as well as in third-party code.

We used Vigor during the development of our NFs, and it caught bugs in our NF code ranging from crashes to packet-injection vulnerabilities and packet corruption. The high-level bugs (i.e., not crashes, hangs, etc.) were caught by the validation step. These bugs are of two types: (1) violations of the contracts that govern the libVig API, and (2) violations of the NF specification. Type (1) bugs can only be caught by the validation step: libVig verification checks that libVig code behaves correctly when accessed correctly; symbolic execution determines how NF code accesses libVig; it is the validation step that puts the two together. Type (2) bugs can also only be caught by the validation step, because that is when the semantics implied by the NF spec get checked; the symbolic-execution step is unaware of these semantics.

Full-stack NF verification uncovered bugs in the public releases of DPDK and Intel ixgbe driver. For example, we found that DPDK can crash during initialization if the first 128 CPU cores are disabled—an unlikely scenario today that will be hard to debug if it arises in the future. Of the eight bugs we found in DPDK and the driver, six were confirmed by the DPDK maintainers and four have been fixed at the time of writing. We first reported these bugs in [39].

In summary, our experience demonstrates the benefit of using verification during development, as well as the added value of full-stack verification: By intercepting bugs early, Vigor saved us a substantial amount of debugging time. By flagging bugs in third-party code we depended on, it further saved us from debugging unknown code.

6.3 Does verification compromise performance?

Whenever verification is discussed for real-world use, one "elephant in the room" is performance: in many cases, verified code tends to perform more poorly than its unverified counterpart. In this section, we show that writing verified NFs with Vigor does not have to compromise performance.

We compare each Vigor NF to a baseline that provides similar functionality but was not developed with verification in mind. All baselines are popular third-party NFs that offer competitive performance. Ideally, each baseline would provide functionality identical to the corresponding Vigor NF, but we were unable to find popular third-party NFs that satisfied this; we could have implemented our own baselines, but then we could not have argued that they are representative unverified NFs. Instead, for each Vigor NF, we picked the baseline that, in the context of our experiments, provided as much as possible (but always a subset of) the functionality provided by the Vigor NF. Comparing to these baselines is a good indicator of whether verification hurts performance.

For all NFs except VigPol, the baseline is a chain of standard Click [27] elements. Click is arguably one of the most popular ways to implement high-performance NFs today. For VigPol, we could not find appropriate Click elements, so the baseline is the Moonpol open-source policer [33], which relies on a DPDK-based framework [16]. All the baselines run on a standard stack: DPDK v.17.11 and Ubuntu Linux with kernel 4.15.0-55-generic. The Click baselines run on the latest stable version of Click (v2.1).

We run the Click baselines with and without batching. Receiving and sending packets in batches can amortize the cost of certain bookkeeping operations (e.g., updating memory-mapped NIC registers), thus increasing throughput at the cost of higher latency. Vigor does not yet support batching (see §7), but the Click baselines do, so we run them both with batching (throughput-optimized) and without (latency-optimized). For batching, we use the default Click batching parameters.

To assess whether performance differences are due to the NF code or to the layers underneath the NF, we also measure a no-op NF, which receives packets at a fixed input port and forwards them to a fixed output port without any processing.

In summary, for each NF type, we compare the Vigor NF running on top of the Vigor stack to a baseline NF running on top of the standard Linux stack, with and without batching. We also ran Vigor NFs on the standard Linux stack, and their performance is the same as on top of the Vigor stack.

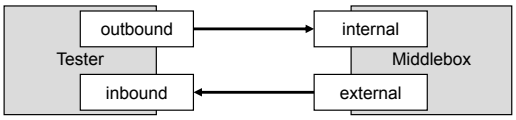

Figure 11. Testbed topology for performance evaluation.

We use the testbed shown in Fig. 11, as per RFC 2544 [7]. The Tester and Middlebox machines are identical, with an Intel Xeon E5-2667 v2 processor @ 3.30 GHz, 32 GB of DRAM, and 82599ES 10 Gbps DPDK-compatible NICs. The Tester machine runs MoonGen [16] to generate traffic and measure packet loss, throughput, and latency; for the latency

measurements, we use hardware timestamps for better accuracy [41]. In each experiment, the Middlebox machine runs either our NF or a baseline, always on a single core.

For each NF, we measure latency and throughput. In particular, for each NF, we generate: (a) "background traffic," which causes the NF's main data structure to be 90% occupied; (b) "probe packets," each of which causes every home bucket in the NF's main data structure to be searched. We make sure that the probe packets always exercise the longest path in the NF by generating a new flow-id for each packet. For example, in the case of a NAT, the background traffic consists of established, long-running TCP flows, while the probe packets come from new, short-lived TCP flows. Vigor NFs receive the same traffic as their baselines. We report the overall throughput achieved by each NF when it starts dropping 0.1% of the packets it receives. We also report the latency experienced *by the probe packets*, because these experience the highest latency. This methodology measures performance under stressful but still realistic conditions; under less stressful conditions, the performance differences are less evident.

Table 5 shows the results. Each reported latency number is the average latency experienced by all the 10^5 probe packets in a single experiment. For Vigor NFs and their no-batching baselines, the standard deviation of the latency across different packets in the same experiment is $100-200$ nanosec. Both average latency and standard deviation are stable across different runs of the same experiment.

NF type	Vigor		Baseline (no batch)		Baseline (batch)	
	Latency (μsec)	Thruput ($Mpps$)	Latency (μsec)	Thruput ($Mpps$)	Latency (μsec)	Thruput ($Mpps$)
NOP	3.90	8.27	4.62	4.07	15.51	14.7*
NAT	4.07	4.86	5.59	1.63	16.30	2.80
Bridge	4.07	4.94	4.76	2.88	15.84	11.2
Load Balancer	4.12	4.02	7.24	1.63	16.26	2.79
Policer	4.03	5.21	5.28†	2.91†	5.20†	11.5†
Firewall	4.02	5.36	5.59	1.63	16.19	2.79

Table 5. Throughput and latency of Vigor NFs and the corresponding baselines. *10 Gbps saturated † Moonpol

Vigor NFs have latency and throughput that is at least as good as the latency-optimized (no batching) baselines. Unexpectedly, the throughput of VigNAT, VigLB, and VigFw is better even than the throughput-optimized (batching) baselines, but this has little to do with the Vigor approach. It is mainly due to the fact that Click loads modules dynamically, which disables a number of the link-time optimizations (LTO) that do apply to the Vigor stack. We confirmed this explanation by disabling the optimizations for our stack and finding that the differences vanish. Not surprisingly, in the case of the no-op NF, batching has ample room to compensate for this, so the batching baseline saturates the network interface.

We conclude that developing and verifying NFs with Vigor does not come at the cost of NF performance.

6.4 Does verification compromise productivity?

Whenever verification is discussed in a practical context, the other elephant in the room (besides performance) is developer productivity. While it is generally true that introducing formal verification can increase the burden on developers, we designed Vigor specifically to lighten this burden: desired properties can be specified in Python, there is no need to write lemmas or understand formal methods, the high upfront cost of verification can be avoided with pay-as-you-go verification, and Vigor automatically provides debug information that helps resolve proof failures. In our subjective assessment, using verification actually improved our productivity when developing our NFs. In this section we try to more objectively evaluate the extent of the burden introduced by Vigor.

The first question is whether developers indeed can use Vigor in a fully push-button mode? The answer is yes, as long as the code they write manages all its persistent state using libVig primitives. Then, is libVig sufficient to write all (or at least most) NFs? The first NF we wrote using libVig was VigNAT, and we added to libVig all the data structures and primitives we thought would be useful. Writing the second one (VigLB) required adding a consistent hashtable to libVig. For the remaining three NFs, we did not need to add anything new to libVig.

Related to the sufficiency of the libVig API, we also asked ourselves whether the choice of abstraction level influences the rate at which libVig's API converges. In other words, had we chosen a different level of abstraction, would libVig have to be bigger or smaller to be sufficient, and how rapidly would the API stabilize? We performed the following thought experiment: we designed "on paper" the 18 NFs listed in [31] using libVig's API, and compared to building them in Click. Then we looked at how many new data structures we needed to add to libVig with each new NF vs. how many new elements had to be added to Click according to Martins et al. [31]. Fig. 12 shows the result. The NFs are listed along the x-axis, sorted in decreasing order of the number of Click elements they use. The continuous lines show how the number of data structures and elements evolve in libVig and Click, respectively, serving as a metric for convergence of the API. We assume we start with an empty libVig and Click.

We conclude (with no claim of statistical significance) that, had we chosen a higher level of abstraction for libVig, such as the Click API, we would have had to add many more primitives to libVig. This is because, the less "primitive" a primitive is, the more likely it is to require changes in order to be useful to a new NF. Once we add verified crypto primitives from HACL*[52] and regexp primitives from LMS [2], libVig will have all the data structures needed to write all the 18 NFs. At that point, NF developers should be able to write any NF they wish in Vigor, push a button, and verify it all the way down to the hardware. Note that this comparison has no bearing on Click itself, because the purpose of the Click

framework is different from that of libVig's.

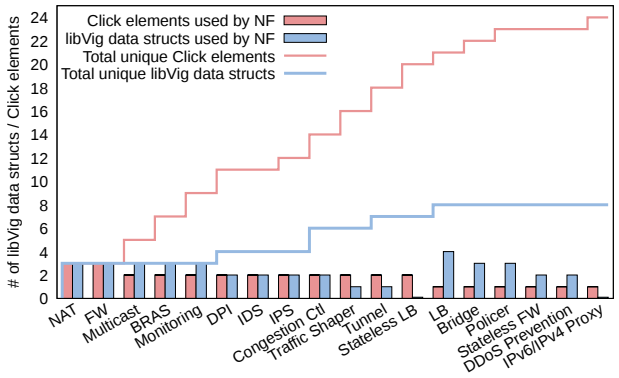

Figure 12. Hypothetical evolution of libVig and Click, as they get used to write new NFs. Click data based on [31].

The second question is how hard is it for developers and standardization bodies to specify the properties they're interested in? For each NF, Table 6 shows the lines of code in the corresponding spec, the amount of time it took to generate the spec from the corresponding RFC (or other standard), and the number of bounds supplied by the developer to instantiate the necessary libVig types (as described in §3). The full RFC-derived specs are relatively short, because Python is expressive. The number of user-supplied bounds is small. The average full spec size across our NFs is 40 LOC, which is $10-20\times$ fewer LOC than the implementation. Generating a spec from an RFC takes a non-trivial amount of time, but this is done once per RFC. Plus, we envision that generating a spec will become part of writing an RFC, which is already a scientifically and socially complex process that can take from weeks to years; we hope that generating a spec will not be the bottleneck, but may actually help the process converge faster.

NF	LOC in spec	Time to write spec	# of bounds
VigNAT	47	3 days	2
VigBr	29	2 days	2
VigLB	56	3 days	4
VigPol	41	3 hours	2
VigFw	32	1 hour	1

Table 6. Statistics on writing NF specifications in Vigor.

The third question is whether pay-as-you-go verification really helps? Table 7 shows the number of modular properties in each NF spec, and the fraction of the spec they account for. Each NF possesses a handful of meaningful, well modularized properties. To verify the one with the shortest spec in isolation requires writing from 1 to 13 LOC of spec, representing $3-40\%$ of the full spec. To verify subsequent properties takes, on average across each NF, $4.1-9.1$ LOC of spec. This suggests that a developer can express meaningful properties in a few minutes, as opposed to taking multiple days to formalize an RFC with all of its (possibly not of interest) corner cases.

NF	Modular properties	Cost of 1st prop. (LOC, % of spec)	Avg. cost of each further prop (LOC,% of spec)
VigNAT	9	13 (27%)	4.1 (9%)
VigBr	7	1 (3%)	4.5 (15%)
VigLB	7	3 (5%)	8.9 (15%)
VigPol	4	13 (31%)	9.1 (22%)
VigFw	5	13 (40%)	5.3 (16%)

Table 7. Cost of writing one-off semantic properties.

Combining this evidence with that on verification time (§6.1), we conclude that the impact on developer productivity would at worst be negligible. Based on our experience described in §6.2, we actually believe that productivity would be enhanced. This is especially true when factoring in the use of traces for diagnosing the root cause of a proof failure.

7 Discussion

We now discuss several aspects of the approach described in this paper, including limitations and lessons learned.

Restrictions on NF Code

Writing NFs with Vigor doesn't require verification expertise, but it does impose constraints:

Event loop: Vigor assumes that the NF's stateless code has one top-level infinite loop that receives/sends packets, and all its other loops are finite. More generally, any loop that can be statically fully unrolled is supported. Loops inside NFs are almost always bounded by maximum packet size, so this imposes negligible restrictions in practice.

Handling persistent state: Any state that persists across packet receives must reside in libVig data structures. This eliminates the need for complex pointer structures in the NF code that could render exhaustive symbolic execution impractical. However, this is a catch-all rule that is broader than necessary: not all persistent state must go into libVig, rather only the state that cannot be effectively havoc'd. If a developer accidentally keeps persistent state in the NF code itself, then verification time is likely to be long, due to path explosion. But it could also happen that Vigor succeeds in havoc-ing that state, and the developer never notices the "mistake." In fact, there does exist persistent state outside libVig, namely in DPDK, the NIC driver, and NFOS, which is successfully havoc'd. To ensure that NF developers are presented with a simple and verification-agnostic model of state handling, we decided to make this rule broad, even if it is over-constraining.

Memory ownership: NF developers must respect the memory ownership model of libVig data structures, as documented in the libVig interface. For instance, after inserting a value into a map, the caller is not allowed to change that value by

reference; instead it must remove the entry from the map, modify it, then re-insert it. Vigor does not currently enforce the memory ownership model at compile time, rather only at verification time, which could make debugging harder. We are exploring the possibility of either piggybacking on a language like Rust, which has compiler support for this ownership model, or performing our own compile-time analysis.

Inherited constraints: Vigor also inherits restrictions imposed by the underlying tools. First, KLEE does not handle symbolic-pointer arithmetic, even with tight constraints. Fortunately, pointer arithmetic is infrequent in NF code. Second, VeriFast does not support all of the C standard library. We have not found this to be a problem, since most of the unsupported functions would anyway impose undue performance overheads on the NF, so a developer is unlikely to use them.

Limitations

The Vigor prototype is now publicly available as Vigor 1.0 [48]. It has several known limitations:

Missing data structures: libVig 1.0 does not provide primitives for regular expression matching, cryptography, or variable-length headers. As a result, we cannot verify NFs that need such primitives, e.g., deep packet inspection or intrusion detection systems. We can fix this by incorporating into libVig verified open-source libraries like LMS [2] for regular expression matching and HACL*[52] for cryptographic primitives.

Parallelism: Vigor 1.0 can only verify single-threaded NFs.

Batching: Vigor 1.0 assumes no batching, i.e., that the NF processes one packet per iteration through the event loop. Batching enables trading off latency for throughput, so it is an important feature to add to a subsequent version of Vigor.

Lessons and Open Questions

Our work on Vigor taught us several lessons and also raised a few interesting questions:

Lower-level abstractions can reduce specification length. We already described in §3 how lowering the level of abstraction for writing specifications facilitates push-button verification. At the same time, this lower level of abstraction enables describing state in terms of basic, generic data structures (maps, vectors, queues, etc.) that have well understood semantics. As a result, a spec expressed in terms of such data structures need not define the data structures but merely reference them (via `import` in Vigor's case). This makes the spec shorter, as well as more comfortable to write. In contrast, the initial spec for VigNAT [50] consisted of ~150 lines of Verifast (a subset of C extended with inductive data types and some generics), and ~150 lines defining the abstract state independently of libVig definitions. Now, with Vigor 1.0, the VigNAT spec has 47 lines of Python, and 19 lines that define the abstract state in terms libVig data structures.

Impact of verification knowledge: We ran an experiment to check if a basic level of verification expertise on the part of NF developers could significantly impact verification time. We made two minor modifications to the NFs, aiming for a reduction in the number of code paths, while preserving the exact same semantics. First, we replaced short-circuiting Boolean operators `&&` and `||` (which produce one code path per term) with their non-short-circuiting counterparts `&` and `|` whenever there were no side effects. Second, we replaced `if` blocks that only assign Boolean values with equivalent Boolean expressions (e.g. `if(c) {b = true;}` turned into `b = b|c;`). As can be computed based on Table 8, these minor changes netted an average reduction of 51% in overall verification time on our 56-contexts machine: The number of paths, and thus symbolic execution time, is roughly halved, but per-trace validation time increases, due to the higher complexity of path constraints. This effect is well understood [28] by verification experts, but verification-agnostic developers can also try these heuristics out with potentially big rewards.

NF	Symbolic execution time		# of traces			Per-trace validation time (avg)
	NF only	*with rest of stack*	*NF only*	*with rest of stack*		
VigNAT	−29%	−38%	−52%	−58%	×	−16%
VigBr	−14%	−40%	−44%	−50%	×	0%
VigLB	−52%	−58%	−60%	−67%	×	+66%
VigPol	−43%	−17%	−32%	−40%	×	−4%
VigFw	−17%	−43%	−42%	−50%	×	−15%

Table 8. Verification statistics for NFs written with some verification expertise. We show decreases (−) and increases (+) as a percentage of the numbers in Table 3.

Going from NF specs to C code: In Vigor, a developer writes the NF implementation, which is then verified against the NF specification written in Python. An alternative approach is to automatically translate the spec into an implementation. This is non-trivial, and we question whether, given a spec, there exists a single efficient C implementation to translate to, or is the decoupling between spec and implementation indeed essential to practicality? Future work on this would need to find ways to translate one-off properties ("partial specs") to code, to translate negative properties (such as "packet is never sent back on its source port") to code, and finally to check that libVig contracts are obeyed by the spec.

Generalizing beyond NFs: We conjecture that the Vigor approach can be used for other types of systems code, such as embedded systems (e.g., alarm systems, industrial controllers). In essence, we expect Vigor-like approaches to work for any system that (a) consists of an event handling loop implementing function f, where $(Output, NewState) = f(Input, OldState)$; (b) can be cleanly separated into stateful code and stateless code implementing f, with the former using a handful of data structures common to systems in that domain; and (c) whose functional specification can be written naturally in terms of these commonly used data structures. We believe such generalization to be feasible, but have no proof.

8 Related Work

Gravel (developed in parallel with Vigor) also aims to verify software NFs, focusing on NFs written in Click. According to the latest tech report [51], it can automatically verify 47% of Click elements as they are, and an additional 21% after minor modifications. Both Gravel and Vigor use exhaustive symbolic execution to verify part of the code. The fundamental difference is what happens with code that cannot be symbolically executed ("non-SE code"). In Gravel, non-SE code needs to be modeled using SMT (satisfiability modulo theories) formulae. In Vigor, non-SE code resides in libVig, and the Vigor contributors verify it using VeriFast; the Validator then combines the outcome of that verification with that of symbolic execution to automatically produce a proof of the entire NF. Our approach is more complicated—it involves more steps and requires Vigor contributors to write proofs—but verifies entire NFs down to the hardware, including non-SE code. In fact, reasoning about non-SE code and its interaction with the rest of the NF is precisely where our contributions arise from. To use Gravel for verifying entire NFs, one needs to model each piece of non-SE code and ideally prove that the models are correct. Nevertheless, Gravel enables developers to use an existing framework (Click), and this may turn out to be a major advantage for real-world adoption.

Earlier work looked at providing low-level properties such as crash freedom and memory safety for NFs. Dobrescu et al. [13] uses exhaustive symbolic execution to prove these properties in simplified or stateless NFs, while Panda et al. [37] ensure these properties by using safe languages like Rust [42]. Vigor can be viewed as generalizing some of this prior work to make high-level semantic verification more accessible, in addition to verifying the entire NF stack.

Work on verification of systems software, including kernels [26, 35], drivers [3, 40], compilers [46], and filesystems [1, 10, 44], served as inspiration for this work. For example, Cogent separates out data structures for the sake of verification [1]. These approaches, however, require verification expertise and rely on languages (like Coq, Haskell, and Cogent) that we cannot expect typical NF developers to master. In contrast, we focused on how to verify NFs written in C without requiring verification expertise from NF developers.

Many specific networked systems have posed interesting challenges that drove the creation of specific tools. Musuvathi et al. [34] model-checked the Linux TCP implementation against a formal specification, Bishop et al. [5] rigorously specified the sockets API and the TCP/UDP protocol stacks for specification-driven testing, Canini et al. [9] combined symbolic execution with model checking to automate testing OpenFlow applications, Hawblitzel et al. [21] verified network applications written in Dafny, a high-level language with built-in verification support, and Beringer et al. [4] verified an OpenSSL implementation, proving functional correctness and cryptographic properties. Vigor doesn't focus on specific systems, but rather is to be used with a broad range of NFs.

Finally, as explained in §1, *network verification* is orthogonal to our work. Individual nodes are modeled and assumed to be correct, allowing the tools to reason about network-wide properties, like reachability, absence of loops, and black holes [18, 23–25, 29, 30, 38, 43, 49]. While most work simply models routers and switches, some tools [17, 47] do go a step further and test the models' compliance. Recent work [32] attempts to bridge the gap between network and NF verification by automatically building models for data plane NFs; these models are then used to prove network-wide properties. Such work is complementary to ours, and can be strengthened by Vigor-generated proofs of NF implementation correctness.

9 Conclusion

We presented Vigor, a software stack and toolchain for building and running formally verified software network functions (NFs). Our goal was to build a verification toolchain that can be used without any verification expertise. Vigor employs symbolic execution—a good candidate, because it requires little hand-holding—but designed our approach knowing that there will always exist NF code that cannot be symbolically executed without running into path explosion.

We based our work on two conjectures about the NF domain. First, the NF code that cannot be practically symbolically executed tends to be common across NFs, hence can be stored in a specialized library and verified by experts using theorem proving, without any input from NF developers. Second, the same primitives provided by this specialized library to NF developers can be productively used to write useful functional specifications. Our two conjectures make it possible to outsource the complex verification tasks to experts, and integrate the outcome of their work automatically into proofs of the entire NF. We tested Vigor on five representative NFs, and it was able to verify them within minutes or hours, depending on whether we verified only the NF or the entire software stack.

Our experimental evaluation shows that the confidence that formal verification offers need not come at the cost of performance. When performing non-batched processing, our formally verified NFs performed at least as well as their non-verified counterparts. Once we extend Vigor to support batching, we hope to show that formal verification does not come at any cost in performance.

10 Acknowledgments

We thank our shepherd, Leonid Ryzhyk, and the anonymous reviewers for their constructive help in improving our paper. We are grateful to the Artifact Evaluation reviewers for their feedback that improved Vigor 1.0. We thank Peter O'Hearn for discussions that put us on the path to pay-as-you-go verification. This work was partly supported by a Starting Grant from the Swiss National Science Foundation.

References

[1] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. 2016. Cogent: Verifying High-Assurance File System Implementations. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*.

[2] Nada Amin and Tiark Rompf. 2017. LMS-Verify: Abstraction Without Regret for Verified Systems Programming. In *Symp. on Principles of Programming Languages*.

[3] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. 2010. SLAM2: Static Driver Verification with Under 4% False Alarms. In *Intl. Conf. on Formal Methods in Computer-Aided Design*.

[4] Lennart Beringer, Adam Petcher, Q Ye Katherine, and Andrew W Appel. 2015. Verified Correctness and Security of OpenSSL HMAC. In *USENIX Security Symp.*

[5] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. 2005. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets. *SIGCOMM Computer Communication Review* 35, 4 (2005).

[6] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. 2008. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*.

[7] S. Bradner and J. McQuaid. 1999. *Benchmarking Methodology for Network Interconnect Devices*. RFC 2544. RFC Editor. http://www.rfc-editor.org/rfc/rfc2647.txt

[8] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symp. on Operating Sys. Design and Implem.*

[9] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Symp. on Networked Systems Design and Implem.*

[10] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Symp. on Operating Systems Principles*.

[11] Elaine Chou. 2016. *A Secure Bootloader for Demonstrating Formal Verification of Hardware-Firmware Interactions on SoCs*. Technical Report. Princeton University. Senior Thesis.

[12] Mihai Dobrescu and Katerina Argyraki. 2014. Software Dataplane Verification. In *Symp. on Networked Systems Design and Implem.*

[13] Mihai Dobrescu and Katerina Argyraki. 2014. Software Dataplane Verification. In *Symp. on Networked Systems Design and Implem.*

[14] DPDK 2019. Data Plane Development Kit. http://dpdk.org.

[15] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *Symp. on Networked Systems Design and Implem.*

[16] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conf.*

[17] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. 2016. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *Symp. on Networked Systems Design and Implem.*

[18] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *Symp. on Networked Systems Design and Implem.*

[19] GitStats 2018. GitStats - linux - Lines. https://phoronix.com/misc/linux-20180915/lines.html.

[20] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. 2015. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* (2015).

[21] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Symp. on Operating Systems Principles*.

[22] Bart Jacobs and Frank Piessens. 2008. The VeriFast program verifier. https://lirias.kuleuven.be/retrieve/30786

[23] Peyman Kazemian, Michael Chan, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real Time Network Policy Checking Using Header Space Analysis. In *Symp. on Networked Systems Design and Implem.*

[24] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking For Networks. In *Symp. on Networked Systems Design and Implem.*

[25] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Symp. on Networked Systems Design and Implem.*

[26] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Symp. on Operating Systems Principles*.

[27] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. on Computer Systems* 18, 3 (Aug. 2000).

[28] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. In *Intl. Conf. on Programming Language Design and Implem.*

[29] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking Beliefs in Dynamic Networks. In *Symp. on Networked Systems Design and Implem.*

[30] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the Data Plane with Anteater. In *ACM SIGCOMM Conf.*

[31] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Symp. on Networked Systems Design and Implem.*

[32] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. 2019. Alembic: Automated Model Inference for Stateful Network Functions. In *Symp. on Networked Systems Design and Implem.*

[33] Moonpol 2018. Moonpol. https://github.com/erkinkirdan/moonpol.

[34] Madanlal Musuvathi, Dawson R Engler, et al. 2004. Model Checking Large Network Protocol Implementations. In *Symp. on Networked Systems Design and Implem.*

[35] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Symp. on Operating Systems Principles*.

[36] Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. In *Symposium on Logic in Computer Science*.

[37] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *Symp. on Operating Sys. Design and Implem.*

[38] Aurojit Panda, Ori Lahav, Katerina Argyraki, Mooly Sagiv, and Scott Shenker. 2017. Verifying Reachability in Networks with Mutable Datapaths. In *Symp. on Networked Systems Design and Implem.*

[39] Solal Pirelli, Arseniy Zaostrovnykh, and George Candea. 2018. A Formally Verified NAT Stack. In *SIGCOMM Workshop on Kernel-Bypass Networks*.

[40] Hendrik Post and Wolfgang Küchlin. 2007. Integrated Static Analysis for Linux Device Driver Verification. In *Intl. Conf. on Integrated Formal Methods*.

[41] Mia Primorac, Katerina Argyraki, and Edouard Bugnion. 2017. How to Measure the Killer Microsecond. In *SIGCOMM Workshop on Kernel-Bypass Networks*.

[42] Rust 2019. Rust Language. https://www.rust-lang.org/.

[43] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B. Terry, and George Varghese. 2017. Correct by Construction Networks Using Stepwise Refinement. In *Symp. on Networked Systems Design and Implem.*

[44] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Symp. on Operating Sys. Design and Implem.*

[45] P. Srisuresh and K. Egevang. 2001. *Traditional IP Network Address Translator*. RFC 3022. Internet Engineering Task Force.

[46] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W Appel. 2015. Compositional CompCert. *SIGPLAN Notices* 50, 1 (2015).

[47] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM Conf.*

[48] Vigor 2019. Source code repository. https://github.com/vigor-nf/vigor.

[49] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. 2005. On Static Reachability Analysis of IP Networks. In *Intl. Conf. on Computer Communications*.

[50] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. In *ACM SIGCOMM Conf.*

[51] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Wang Xi. 2018. *Gravel: Automated Software Middlebox Verification*. Technical Report CSE-18-09-03. University of Washington.

[52] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Conf. on Computer and Communication Security*.