View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Infoscience - École polytechnique fédérale de Lausanne

A Formally Verified NAT

Arseniy Zaostrovnykh
EPFL, Switzerland
arseniy.zaostrovnykh@epfl.ch

Solal Pirelli
EPFL, Switzerland
solal.pirelli@epfl.ch

Luis Pedrosa
EPFL, Switzerland
luis.pedrosa@epfl.ch

Katerina Argyraki
EPFL, Switzerland
katerina.argyraki@epfl.ch

George Candea
EPFL, Switzerland
george.candea@epfl.ch

ABSTRACT

We present a Network Address Translator (NAT) written in C and proven to be semantically correct according to RFC 3022, as well as crash-free and memory-safe. There exists a lot of recent work on network verification, but it mostly assumes models of network functions and proves properties specific to network configuration, such as reachability and absence of loops. Our proof applies directly to the C code of a network function, and it demonstrates the absence of implementation bugs. Prior work argued that this is not feasible (i.e., that verifying a real, stateful network function written in C does not scale) but we demonstrate otherwise: NAT is one of the most popular network functions and maintains per-flow state that needs to be properly updated and expired, which is a typical source of verification challenges. We tackle the scalability challenge with a new combination of symbolic execution and proof checking using separation logic; this combination matches well the typical structure of a network function. We then demonstrate that formally proven correctness in this case does not come at the cost of performance. The NAT code, proof toolchain, and proofs are available at [58].

CCS CONCEPTS

• **Networks** → **Middle boxes / network appliances**; • **Software and its engineering** → *Formal software verification*;

KEYWORDS

Network-Function Verification; Lazy Proofs; Symbolic Execution

ACM Reference format:
Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017,* 14 pages.
DOI: https://doi.org/10.1145/3098822.3098833

1 INTRODUCTION

This work is about designing and implementing software network functions (NFs) that are proven to be secure and correct. Software NFs have always been popular in low-rate environments, such as home gateways or wireless access points. More recently, they have also appeared in experimental IP routers [20] and industrial middleboxes [8] that support multi-Gbps line rates. Moreover, we are witnessing a push for virtual network functions that can be deployed on general-purpose platforms on demand, much like virtual machines are being deployed in clouds.

There exists a lot of prior work on network verification, but, to the best of our knowledge, none that reasons about both the security and semantic correctness of NF implementations. Most of that work relies on models of NFs that are different from their implementations, hence it cannot reason about the latter (although we should note that NF models can be very effective in reasoning about network configuration [24, 25, 30–32, 38, 39, 46, 52, 55, 59]). One exception is Dobrescu et al. [19], which introduced the notion of software data-plane verification, and which proves low-level properties for NF implementations written in Click (i.e., C++) [35]. That work, however, cannot prove semantic correctness of stateful NFs, because it does not reason about state. For instance, even though Dobrescu et al. prove crash-freedom and bounded execution for a specific NAT implementation, they cannot prove that it is semantically correct, due to not having a way to reason about the content of the flow table (e.g., whether entries are added or expired correctly).

Our contribution is a NAT function, written in C and using the DPDK packet-processing library [21], which we prove to implement the semantics specified in RFC 3022 [53] and to be crash-free and memory-safe. We chose this particular NF because it is arguably one of the most popular ones, yet it has proven hard to get right over time: the NAT on various Cisco devices can be crashed [17] or hung [15] using carefully crafted inputs; similar problems exist in Juniper's NAT [16], the NAT in Windows Server [40], and NATs based on NetFilter [18]. Moreover, like many NFs, NATs maintain per-flow state that needs to be properly updated and expired, which is a typical source of verification challenges.

We implemented our NAT in C, because this is the language typically used for high-performance packet processing, and it benefits from a rich and stable ecosystem that includes DPDK. Given that we anyway wrote our NAT from scratch—and our approach, in general, requires refactoring—we did consider using a more verification-friendly language. In the end, however, we considered that NF developers are more likely to adopt our toolset if it allows them to code in a familiar language and leverage existing expertise and tools, even if they have to follow extra constraints (such as using a specific library of data structures) and annotate their code. Recent work argues that verifying the C implementation of a real, stateful NF is infeasible with symbolic execution [55], but we show

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCOMM '17, August 21-25, 2017, Los Angeles, CA, USA
© 2017 Copyright held by the Owner/Author. Publication rights licensed to ACM.
ISBN 978-1-4503-4653-5/17/08…$15.00
DOI: https://doi.org/10.1145/3098822.3098833

that it can be done if symbolic execution is combined with other verification techniques.

The rationale behind our approach is that different verification techniques are best suited for different types of code. The beauty of symbolic execution [9] lies in its ease of use: it enables automatic code analysis, hence can be used by developers without verification expertise. The challenge with symbolic execution is its notorious lack of scalability: applying it to real C code typically leads to path explosion [19, 55]. The part of real NF code that typically leads to unmanageable path explosion is the one that manipulates state.

Hence, we split NF code into two parts: (1) A library of data structures that keep all the "difficult" state, which we then formally prove to be correct—this takes time and formal methods expertise, but can be amortized if the library is re-used across multiple NFs; and (2) stateless code that uses the library, which we automatically and quickly verify using symbolic execution. The challenge lies in combining the results of these two verification techniques, and for that we developed a technique we call "lazy proofs". A lazy proof consists of sub-proofs structured in a way that top-level proofs proceed *assuming* lower level properties, and the latter are proven lazily a posteriori. For example, symbolic execution requires the use of models that must be correct; we first do the symbolic execution and only afterward validate automatically the correctness of the models. This approach enables us to avoid having to prove that our models are *universally* valid—which is hard—but instead only prove that they are valid for the specific NF and the specific properties we verified earlier with symbolic execution. This is much easier.

We show that formally verifying the correctness of our NAT does not come at the price of performance: compared to an unverified NAT written on top of DPDK, our verified NAT offers similar latency and less than 10% throughput penalty. Any DPDK-based NAT we experimented with, verified or not, significantly outperformed NetFilter, the popular Linux built-in NAT.

The rest of the paper is structured as follows: after providing background (§2), we illustrate our approach with a simple example (§3), state formally what we proved about our NAT (§4), describe our verification process (§5), and report on our experimental evaluation (§6). Then we discuss limitations and future work (§7), present related work (§8), and conclude (§9).

2 BACKGROUND

Our work falls in the general area of "data-plane verification." This term is typically used to denote two different types of approaches: One category of work treats as one big data plane the combination of the *configured* data planes of network devices in a network, and reasons about network properties (reachability, loops, etc.)—we refer to this as "network verification." An orthogonal category reasons about properties of the data-plane *software* running on individual devices, and reasons about software properties (crash freedom, bounded execution time, memory safety, etc.)—we refer to this as "NF verification." In network verification, the goal is to demonstrate that a particular property (e.g., that a packet with certain header features will always reach a given destination) holds in a specific network with particular NFs configured and connected in a particular way. In NF verification, the goal is to prove that a particular property (e.g., there exists no input packet that can trigger a buffer overflow in the NF) holds for all networks and workloads, i.e., regardless of how the NF is configured or connected. There is a rich body of work on network verification [24, 25, 30–32, 38, 39, 46, 52, 55, 59]. In contrast, there is much less work on NF verification [19].

The success of network verification depends on the success of NF verification: Network verification relies on models of the NFs that compose the network, whether these models are informally captured in an RFC or more formally in a SEFL model [55], NICE model [10], etc. However, a model-based proof that a packet will always reach a destination is trivially invalidated by an implementation bug in a middlebox that causes that packet to be dropped, in violation of the model. There are ways of testing whether such a model is faithful to a given implementation [55], but there is a big gap between testing and verification: a successfully tested model can still exhibit behaviors that do not occur in the implementation, and vice versa. NF verification can, however, ensure that an NF implementation deployed in the real network is indeed faithful to the model used for verifying the network.

Our work belongs to the category of NF verification and aims to improve the state of the art on two fronts: (1) verify high-level semantic properties, such as the correct implementation of an RFC, and (2) verify NFs that are stateful. Dobrescu et al. [19] did verify a stateful NAT, but proved only low-level properties (crash freedom and bounded execution), therefore not encountering some of the harder challenges of stateful NFs. We aim to resolve these challenges, while not placing on operators the burden of writing or adapting models, and at the same time keeping the NF implementations' performance in the same ballpark as that of non-verified NFs. In this paper we report on our first step in this effort: the development of a stateful, well-performing NF, which we prove to implement the NAT semantics as understood from RFC 3022, in addition to being free of crashes, memory bugs, leaks, and other low-level properties.

3 THE VIGOR APPROACH

To verify our NAT, we developed a verification toolchain that we call Vigor, which includes a library of verified data structures, called libVig. We envision the software development process with Vigor to revolve around three distinct developer roles with a clear separation of concerns: libVig developers, standards developers who write contracts in formal logic to specify public standards, and NF developers who implement these standards with verified NFs. The first two roles require expertise in software verification and formal methods, but their time and effort investment can be amortized across the many NFs that share common components and implement the same standards in different ways. Developers in the latter role, however, should need little to no expertise in verification. It is they who are the true beneficiaries of Vigor, as they can now write code that they prove correct with relative ease. In this paper, the authors took on all three roles, but we envision that eventually the roles could be taken on by different specialized teams or even different organizations.

We illustrate the use and functioning of Vigor with a trivially simple NF that implements the discard protocol [48]: an infinite loop receives packets from one interface, discards the ones sent to port 9, and forwards the rest through another interface.

Code. The NF developer does two extra things relative to writing standard code: she annotates loops and encapsulates state in libVig data structures that Vigor can reason about. Fig. 1 shows our verified implementation. It includes an annotated event loop (`VIGOR_LOOP` on l.6) and a ring buffer (`r` on l.4) for absorbing bursts, which is accessed through four calls (ll.9, 11, 12, 13). Network interaction happens via three functions: `receive` (l.10) non-blockingly reads an inbound packet and stores it in the output argument, returning success or failure; `can_send` (l.12) checks if a new packet can be sent; and `send` (l.14) sends the packet pointed to by its argument.

```
 1 #define CAP 512
 2 int main() {
 3   struct packet p;
 4   struct ring *r = ring_create(CAP);
 5   if (!r) return 1;
 6   while(VIGOR_LOOP(1))
 7   {
 8     loop_iteration_begin(&r);
 9     if (!ring_full(r))
10       if (receive(&p) && p.port != 9)
11         ring_push_back(r, &p);
12     if (!ring_empty(r) && can_send()) {
13       ring_pop_front(r, &p);
14       send(&p);
15     }
16     loop_iteration_end(&r);
17   }
18   return 0;
19 }
```

Figure 1: Verified implementation of the discard protocol.

Loop invariants. Our verification process requires loop invariants to reason about the effect of loops. Currently, the NF developer writes these invariants manually, in formal logic (Fig. 2, ll.1-5) and in C (ll.7-9). In future work, we hope to be able to extract them automatically from the code using existing techniques [23, 47], or at least to automatically help the NF developer formulate them.

```
1 /*@
2   fixpoint bool packet_constraints_fp(packet p) {
3     switch(p) { case packet(port): return port != 9; }
4   }
5   @*/
6
7 static bool packet_constraints(struct packet* p) {
8   return p->port != 9;
9 }
```

Figure 2: An invariant preserved by the loop in Fig. 1 is that $\forall packet\ p \in ring\ r, packet\_constraints(p) == true$.

Target properties. Vigor proves that this NF never crashes (an example of a low-level property) and that it never yields a packet with target port 9 (an example of a semantic property). For the latter, the gist of the proof is to show that the code never pushes onto the ring packets with target port 9, and that the ring never alters the stored packets; these two properties imply that a popped packet can never have target port 9. There are three steps:

Step 1: Function contracts & proofs. For each method of a libVig data type[1], the libVig developer writes a contract, i.e., a formal specification of what the method guarantees; she also writes a formal proof that the implementation of that method satisfies the contract. This is a significant undertaking, but can be amortized across the potentially many NFs that will use the same data type. Fig. 3 shows the contract (ll.2-6) and the implementation of the `ring_pop_front` function, which removes the packet at the front of the ring[2]. The contract says that this function will not damage the ring, will remove the packet at the front of the ring, and will honor certain constraints that hold for all packets in the ring (l.6), as long as the ring was in a good state and honored these constraints before the function was called (l.2). In the contract, `packet_constraints_fp` is an abstract function, i.e., the contract says that `ring_pop_front` will honor *any* packet constraints as long as these hold before it is called. The NF developer can provide desired constraints when using libVig; in this example, the provided constraint (Fig. 2) conveniently serves as a loop invariant too. §5.1.2 has the details on libVig contracts.

```
 1 void ring_pop_front(struct ring* r, struct packet* p)
 2 /*@ requires ringp(r, ?packet_constraints_fp, ?lst, ?cap) &*&
 3               lst != nil &*& packetp(p, _); @*/
 4 /*@ ensures ringp(r, packet_constraints_fp, tail(lst), cap) &*&
 5              packetp(p, head(lst)) &*&
 6              true == packet_constraints_fp(head(lst)); @*/
 7 {
 8   //@ extract_first(r);
 9   struct packet* src_pkt = r->array + r->begin;
10   p->port = src_pkt->port;
11   r->len = r->len - 1;
12   r->begin = r->begin + 1;
13   if (r->cap <= r->begin) {
14     r->begin = 0;
15     //@ stitch_with_empty_overflow(r);
16   } else {
17     //@ stitch_with_empty(r);
18   }
19 }
```

Figure 3: Excerpt from the implementation of `ring_pop_front()` and its formal contract.

Step 2: Exhaustive symbolic execution. (a) Vigor replaces all function calls that access state or interact with the network with calls to a symbolic model. For example, the symbolic model for `ring_pop_front` (model (a) in Fig. 4) returns a packet with fully symbolic content (i.e., a packet whose content could be anything whatsoever) constrained via `packet_constraints` to have its target

[1] We use the terms "data structure" and "data type" interchangeably, with the understanding that the data structure state is encapsulated behind a well-defined interface.

[2] This implementation is only an illustrative example. In a our verified NAT, and in most real implementations, we would not copy packets field by field (Fig. 3, l.10) but rather return a pointer to the packet.

port different from 9. Despite its simplicity, this model captures all the behavior of `ring_pop_front` that matters in our context, namely that it never yields a packet with target port 9. (b) Once all function calls have been replaced with calls to symbolic models, Vigor symbolically executes the resulting code. Even though Vigor is symbolically executing real C code, this step terminates quite quickly, because the models are stateless and with few branching points (like the one in Fig. 4), and the loop annotations help prevent unnecessary unrolling. This exhaustive symbolic execution has two outcomes, both assuming the symbolic model is valid: First, it proves that the target low-level property (i.e., that the NF cannot crash for any input) holds. Second, it produces all the feasible function-call sequences that could result from running the code, along with the constraints on program state that hold after each call. Fig. 5 shows one such call sequence that results when the ring is full. For details on exhaustive symbolic execution, see §5.2.1.

Model (a)

```
void ring_pop_front(struct ring* r, struct packet* p) {
  FILL_SYMBOLIC(p, sizeof(struct packet), "popped_packet");
  ASSUME(packet_constraints(p));
}
```

Model (b)

```
void ring_pop_front(struct ring* r, struct packet* p) {
  FILL_SYMBOLIC(p, sizeof(struct packet), "popped_packet");
  // No constraint on the packet's target port.
}
```

Model (c)

```
void ring_pop_front(struct ring* r, struct packet* p) {
  p->port = 0;
}
```

Figure 4: Symbolic models of `ring_pop_front`.

Step 3: Lazy model validation. (a) For each function call that accesses state, in each feasible call sequence, Vigor verifies that the symbolic model used to produce the speculative verification via symbolic execution in Step 2 was, in retrospect, valid *for that call*. This validity means that the output of the model is a superset (in the sense of constrained symbolic state) of the output that the actual implementation could produce. For example, consider the call to `ring_pop_front` (Fig. 5, l.13): Vigor extracts the constraints on symbolic program state that held right after the model was symbolically executed in Step 2; inserts, right after the call, an assertion for these so-called path constraints (l.16); and asks a proof checker to verify that this assertion is compatible with `ring_pop_front`'s contract (details in §5.2.3). The proof checker concludes that it is, and in particular that the output of the model (a packet whose target port can be anything but 9) is a superset of the output specified by the function's contract, hence also of the function's implementation (since Step 1 proved that the implementation satisfies the contract). (b) Vigor verifies that, after every packet `send()`, not shown in Fig. 5, the target semantic property holds, i.e., the output packet does not have target port 9. More details appear in §5.2.2.

```
1  struct ring* arg1;
2  struct packet arg2;
3
4  loop_invariant_produce(&(arg1));
5  //@ open loop_invariant(_);
6  bool ret1 = ring_full(arg1);
7  //@ assume(ret1 == true);
8  bool ret2 = ring_empty(arg1);
9  //@ assume(ret2 == false);
10 bool ret3 = can_send();
11 //@ assume(ret3 == true);
12 //@ close packetp(&(arg2), packet((&(arg2))->port));
13 ring_pop_front(arg1, &(arg2));
14 //@ open packetp(&(arg2), _);
15
16 //@ assert(arg2.port != 9);
```

Figure 5: Example function-call sequence that results from Step 2, annotated by Vigor with an assertion of a path constraint (l.16).

Invalid models. An invalid model will cause either Step 2 or Step 3 to fail, but it will never lead to an incorrect proof. For example, model (b) in Fig. 4 is too abstract for our purpose: it returns a packet whose content could be anything at all, including having a target port 9. This is an "over-approximate" model in verification speak. If Vigor uses this model in Step 2, then Step 3b fails: since the model can return packets with target port 9, Vigor cannot verify for all call sequences that the output packet does not have target port 9. Conversely, model (c) in Fig. 4 is too specific for our purpose: it always returns a packet with target port 0, i.e., it is an "under-approximate" model. If Vigor uses this model in Step 2, then Step 3a fails: Recall that, for each call, Vigor obtains the path constraints that held right after the model was symbolically executed in Step 2, and inserts, right after the call, an assertion for these path constraints. With this model, the assertion would be `//@ assert(arg2.port == 0)`. The proof checker cannot confirm that this assertion is always true, because `ring_pop_front`'s contract (Fig. 3, ll.4–6) specifies a wider range for `arg2.port` than `0`.

This section illustrated how Vigor stitches symbolic execution with proof verification via the function-call sequences. There are a couple other steps involved in the proof (omitted here for clarity) that we describe fully in §5.

4 PROVEN PROPERTIES

We verified our NAT (which we call VigNAT) using the approach outlined in §3. We now describe the specific properties we verify.

4.1 Semantic Properties

We proved that VigNAT correctly implements the semantics specified in the Traditional NAT RFC [53]. For this, we wrote a NAT specification that formalizes our interpretation of the RFC, and which we believe to be consistent with typical NAT implementations. The specification [58] has 300 lines of separation logic [51] and took 3 person-days to complete.

We started from formally describing NAT behavior as shown in Fig. 6, in terms of the effect that a packet arrival has on abstract state (*flow_table*). There are three static configuration parameters: the capacity of the flow table (*CAP*), the flow timeout (T_{exp}), and the

```
1  Packet $P$ arrives at time $t$ $\rightarrow$ $P$ is accepted
2                                 $\rightarrow$ expire_flows($t$)
3                                 $\rightarrow$ update_flow($P$, $t$)
4                                 $\rightarrow$ forward($P$)
5
6  expire_flows($t$) :=  $\forall G \in flow\_table$
7                        s.t. $G.timestamp + T_{exp} <= t$ :
8                            remove $G$ from $flow\_table$
9
10 update_flow($P$, $t$) := if ($F(P) \in flow\_table$) {
11                           $\forall G \in flow\_table$ s.t. $F(P) = G$ :
12                               set $G.timestamp = t$
13                         } else {
14                           if ($P.iface = internal$) {
15                               if ($size(flow\_table) < CAP$) {
16                                  insert $F(P)$ in $flow\_table$
17                               }
18                           }
19                         }
20 forward($P$) :=   if ($F(P) \in flow\_table$) {
21                       if ($P.iface = internal$) {
22                           $\rightarrow S.data = P.data$
23                           $\rightarrow S.iface = external$
24                           $\rightarrow S.dst\_ip = P.dst\_ip$
25                           $\rightarrow S.dst\_port = P.dst\_port$
26                           $\rightarrow S.src\_ip = EXT\_IP$
27                           $\rightarrow S.src\_port = F(P).ext\_port$
28                           $\rightarrow$ send packet $S$
29                     } else {
30                           $\rightarrow S.data = P.data$
31                           $\rightarrow S.iface = internal$
32                           $\rightarrow S.dst\_ip = F(P).int\_ip$
33                           $\rightarrow S.dst\_port = F(P).int\_port$
34                           $\rightarrow S.src\_ip = P.src\_ip$
35                           $\rightarrow S.src\_port = P.src\_port$
36                           $\rightarrow$ send packet $S$
37                       }
38                   } else {
39                       drop packet $P$
40                   }
```

Figure 6: A conceptual summary of the formal specification of the NAT semantic properties (based on RFC 3022).

IP address of the external interface (*EXT_IP*). The $F(P)$ function extracts from the flow table the packet flow ID, based on its source and destination IP addresses and ports. With every packet arrival (l.1), the NAT finds and removes expired flows (l.2), updates the flow table according to the received packet (l.3), then potentially rewrites the packet and forwards it (l.4). To update the flow table, the NAT finds all entries with the same flow ID as the received packet and updates their timestamps (ll.10–12); if there are no matching entries (l.13), and if the packet arrived at the internal interface (l.14), and if the flow table is not full (l.15), then the NAT adds a new entry in the flow table (l.16). If, at this point, there exists an entry in the flow table with the packet's flow ID (l.20), then the NAT forwards the packet, modifying its headers depending on whether the packet arrived at the internal or external interface; otherwise, it drops the packet (l.39).

We wrote a formal machine-readable specification of NAT semantics, organized along similar lines. It merges `expire_flows`, `update_flow`, and `forward` into a single decision tree. The tree consists of branches over the conditions shown in Fig. 6 (pre-conditions), such as $P.iface = internal$, and assertions that check the corresponding output (post-conditions), such as "packet P is dropped" or "$S.dst\_ip$ equals $P.dst\_ip$." Both pre- and post-conditions are written in separation logic, formulated as predicates on abstract NAT state and/or incoming/outgoing packets. The decision tree covers both branches of each pre-condition, so it provides a complete specification of how the NAT behaves under every circumstance. We formally verify that the C implementation of VigNAT implements the behavior in this formal specification derived from Fig. 6.

Finally, since VigNAT maintains its state in libVig data structures, we also prove that it uses these data structures correctly (§5.2.4), i.e., that the data structures' pre-conditions are satisfied.

4.2 Low-Level Properties

Besides NAT semantics, we also prove that VigNAT is free of the following undesired behaviors: buffer over/underflow, invalid pointer dereferences, misaligned pointers, out-of-bounds array indexing, accessing memory that is not owned by the accessor, use after free, double free, type conversions that would overflow the destination, division by zero, problematic bit shifts, and integer over/underflow. Proving these properties boils down to proving that a set of assertions introduced in the VigNAT code—either by default in the KLEE symbolic execution engine [9] or using the LLVM undefined behavior sanitizers [42, 43, 57]—always hold.

5 DESIGN AND IMPLEMENTATION

In this section, we describe the design of VigNAT and how we verify its properties, as well as a few implementation-related highlights. Full details and source code are available at [58].

While the goal of the work presented in this paper is specifically to build a formally verified NAT, our broader goal is to find a practical way to verify *any* stateful NF, so we took a more principled approach than strictly necessary. In our view, practicality consists of the simultaneous achievement of two design goals: competitive performance and low verification effort. The latter has three components: writing the code in a way that can be verified, writing the proof, and verifying the proof. Vigor supports C, and thus does not impose an undue burden on writing the NF code, so we focus on devising a technique for productively writing and verifying a realistic NF.

Well-known verification approaches that are relevant to this task include whole-program theorem proving (e.g., seL4 [34]) or per-path/per-state techniques like symbolic execution [19, 55]. With the former, verifying a property proof is relatively fast, but writing the proof is a slow, often manual job. With the latter, verifying a property in a real NF can take long, even forever, due to path explosion [19, 55], but is easy to automate. To verify a stateful NF, neither approach seems practical on its own.

In our approach, we decompose the proof into parts, and prove each part with whichever technique is suited for that part; after that, we stitch the proofs together. We posit that most NFs consist of one part that is common across many NFs (thus making it worthwhile

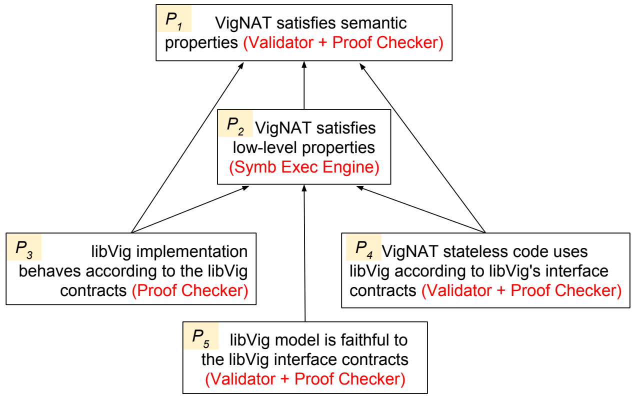
P_1 VigNAT satisfies semantic properties (Validator + Proof Checker)
P_2 VigNAT satisfies low-level properties (Symb Exec Engine)
P_3 libVig implementation behaves according to the libVig contracts (Proof Checker)
P_4 VigNAT stateless code uses libVig according to libVig's interface contracts (Validator + Proof Checker)
P_5 libVig model is faithful to the libVig interface contracts (Validator + Proof Checker)

Figure 7: Structure of the VigNAT correctness proof. $P_i(X) \leftarrow P_j$ symbolizes that the proof of property P_i is done by X under the assumption that P_j holds.

to invest manual effort in proving its correctness) and another part that is different in each NF (and thus its verification should be automatic). We also posit that, over time, NFs will converge to using a stable, common set of data types to encapsulate NF state, and the difference between NFs will result primarily from how their stateless code employs these data types. For VigNAT, we put all NF state in data structures that reside in a library, and use human-assisted theorem proving to verify the correctness of this library. We then use symbolic execution to prove the correctness of the stateless code.

The challenge is how to stitch together the results of the two verification techniques. We developed *lazy proofs*, a way to automatically interface symbolic execution to a proof checker based on separation logic. We built a *Validator* that implements this technique and glues sub-proofs together into the final proof that VigNAT implements the NAT RFC [53].

The VigNAT proof consists of five sub-proofs, shown in Fig. 7. The top-level proof objective P_1 is to show that VigNAT exhibits correct NAT semantics. The proof of P_1 assumes three things: First, the code must work properly in a basic sense, such as not crashing and having no overflows (P_2). Second, the implementations of the library data structures must work as specified in their interface contracts (P_3)—e.g., looking up a just-added flow should return that flow. Third, the stateless part of the NF must use the data structures in a way that is consistent with their interfaces (P_4)—e.g., a pointer to the flow table is never mistakenly passed in as a pointer to a flow entry. Assuming $P_2 \wedge P_3 \wedge P_4$, the Validator produces a proof of P_1 that is mechanically verified by the proof checker.

These three assumptions must of course be proven. To prove P_2—that VigNAT satisfies low-level properties—Vigor symbolically executes the stateless code and checks that the properties hold along each execution path. For this to scale, we employ abstract symbolic models of the library data structures. Therefore, the proof of P_2 must assume that these models are correct (P_5), that the data structure implementations satisfy their interfaces (P_3), and that the stateless code correctly uses the stateful data structures (P_4). If any of these three assumptions were missing, the proof would not work[3]. To prove P_3, we employ relatively straightforward (but still tedious) theorem proving to show that the library implementation satisfies the contracts that define its interface.

It is in the proof of P_4 and P_5 that we find a second scalability benefit of the lazy proofs technique: Not only does it allow us to stitch together proofs done with different tools (thereby allowing us to employ for each sub-proof whichever tool offers the optimal benefit-to-effort ratio) but also makes it possible to get away with proving weaker properties. For example, instead of proving that P_5 is universally true and then using this proof to further prove P_2, we first prove P_2 *assuming* P_5, and afterward only prove that P_5 holds for the *specific way* in which the proof of P_2 relies on P_5. This use-case-specific proof of P_5 is easier than proving P_5 for all possible use cases.

We now describe Vigor's data structure library and its correctness proof (§5.1), our "lazy proofs" technique and its use for proving the correctness of VigNAT's semantics (§5.2), and conclude with a summary of the Vigor workflow (§5.3) and of the assumptions underlying our approach (§5.4).

5.1 A Library of Verified NF Data Types

VigNAT consists of stateless application logic that manipulates state stored in data structures, like hash tables and arrays, provided by the Vigor library (libVig). For example, in §3, we placed incoming packets into a `ring` data structure from libVig. Generally speaking, stateless NF code should be free of any dynamically allocated state and complex data structures. It can retain basic program state, such as statically allocated scalar variables as well as `struct`s of scalars.

Dealing with explicit state in the verification of imperative, non-typesafe programs is hard, mainly due to the difficulty of tracking memory ownership and type information, as well as disentangling pointer aliases. For example, the question of which memory a `void*` pointer could ever reference is often undecidable. Functional, typesafe languages (Haskell, ML, etc.) are appealing for verification, but to us it was paramount to both support C (preliminary evidence [60] suggests C to be widely popular among NF developers) and enable verification of a fully stateful NF. We accomplish both by encapsulating NF state behind libVig's interface and adopting a disciplined use of pointers. While this approach is not compatible with all software, we believe it is a good match for NFs.

Besides data types for NFs, libVig also provides a formal interface specification that defines the behavior of these data types, along with a proof that the libVig implementation obeys the specification. To enable symbolic execution of stateless NF code, libVig also provides symbolic models of its data types. We verify libVig once, and the proof carries over to any NF that uses the library.

This subsection details the libVig implementation (§5.1.1), describes our use of abstraction and contracts to formally specify libVig's semantics (§5.1.2), shows how we prove that the implementation satisfies its interface specification (§5.1.3), and presents the symbolic models of libVig's data structures (§5.1.4) to be used by a symbolic execution engine.

[3] It may seem strange that assumptions P_3 and P_4 are needed both for the proof of low-level properties and that of semantic properties, but this is because "satisfying interface contracts" relates both to high-level interface semantics and to basics like proper data encapsulation.

5.1.1 libVig implementation. Competitive performance is an important design goal, and libVig is a good place to optimize for performance. A key design decision we made is to *pre*allocate all of libVig's memory. While this lacks the flexibility of dynamically allocating at runtime, it offers control over memory layout, making it possible to control cache placement and save the run-time memory management overhead. The cost of preallocation is negligible (e.g., VigNAT's peak resident set size is 27 MB during our experiments), and we believe preallocation is fully compatible with how real NFs use state. In terms of verification with Vigor's proof checker, static allocation does not offer noticeable benefits over dynamic allocation, but may do so for other checkers.

As of this writing, libVig provides several basic data structures that we needed to develop VigNAT: a *flow table*, implemented as a double-keyed hash map, a *network flow* abstraction, a *ring buffer*, an *expirator* abstraction for tracking and expiring flows, a *batcher* for grouping homogeneous items, a *port allocator* to keep track of allocated ports, and a classic hash table, array, and vector. libVig also provides an *nf_time* abstraction for accessing system time and a *dpdk* layer on top of the DPDK framework.

5.1.2 Using abstraction and contracts to formally specify libVig semantics. We specify the semantics of libVig data types in terms of abstract state that the data types' methods operate on. This is the same approach we took in formalizing the NAT RFC (§4.1). The pre-conditions and post-conditions for each method form the *contracts* that define what each data type is supposed to do.

Fig. 8 shows a snippet of a `get` method for the libVig flow table. The pre-condition is on lines 3-6 and the post-condition on lines 7-14. The contracts are meant for the Validator's and proof checker's consumption, but they can also serve as documentation, when ambiguous natural language or reading source code fall short.

Each `requires` pre-condition states the requirements for the function to run: a relationship between its arguments and the abstract state, or a memory ownership token for a pointer. Each `ensures` post-condition specifies what holds after the completion of the method: a relationship between the arguments and the return value, the updated value at a certain memory location, or a memory ownership token.

We adopt a "sanitary" policy on the use of pointers: stateless code can pass/receive pointers across the libVig interface, but the libVig data structure remains opaque to the caller. Stateless code can copy pointers, assign them, compare them for equality, but not dereference them. Vigor automatically checks that stateless code obeys this discipline (§5.2.4).

5.1.3 Verifying libVig correctness (P_3). Once the formalization of the interface is complete, we write the proof, i.e., we annotate the code with assertions, loop invariants, etc. and define lemmas for the intermediate steps of the proof.

The proof checker starts by assuming the pre-condition (ll.3-6) and steps through every code statement while developing its set of assumptions. When it encounters a branch condition, it explores both branches. Inlined annotations (lines 16-18, 20-22, 24, 28-30) help the checker along the way to understand the transformations of abstract state, and it verifies that they indeed correspond to the transformations of concrete machine state. On memory accesses (l.19), the proof checker checks the validity of the address and the memory ownership token. On method calls (l.23), it checks the pre-condition of the called method and then assumes its post-condition, in essence replacing the call with an assumption of the callee's post-condition (it verifies separately that the post-condition indeed holds whenever the callee returns). When reaching a return point (l.31), the proof checker checks the post-condition.

```
 1 int dmap_get_by_first_key /*@ <K1,K2,V> @*/
 2        (struct DoubleMap* map, void* key, int* index)
 3 /*@ requires dmappingp<K1,K2,V>(map, ?kp1, ?kp2, ?hsh1, ?hsh2,
 4                                  ?fvp, ?bvp, ?rof, ?vsz,
 5                                  ?vk1, ?vk2, ?rp1, ?rp2, ?m)
 6                &*& kp1(key, ?k1) &*& *index |-> ?i; @*/
 7 /*@ ensures dmappingp<K1,K2,V>(map, kp1, kp2, hsh1, hsh2,
 8                                 fvp, bvp, rof, vsz,
 9                                 vk1, vk2, rp1, rp2, m) &*&
10                kp1(key, k1) &*& (dmap_has_k1_fp(m, k1) ?
11                    (result == 1 &*& *index |-> ?ind  &*&
12                        ind == dmap_get_k1_fp(m, k1) &*&
13                        true == rp1(k1, ind)) :
14                    (result == 0 &*& *index |-> i)); @*/
15 {
16   /*@ open dmappingp(map, kp1, kp2, hsh1, hsh2,
17                      fvp, bvp, rof, vsz,
18                      vk1, vk2, rp1, rp2, m); @*/
19   map_key_hash *hsh_a = map->hsh_a;
20   //@ map_key_hash *hsh_b = map->hsh_b;
21   //@ assert [?x]is_map_key_hash(hsh_b, kp2, hsh2);
22   //@ close [x]hide_map_key_hash(map->hsh_b, kp2, hsh2);
23   int hash = hsh_a(key);
24   //@ open [x]hide_map_key_hash(map->hsh_b, kp2, hsh2);
25   int res = map_get(map->bbs_a, map->kps_a, map->khs_a,
26                     map->inds_a, key, map->eq_a,
27                     hash, index, map->keys_capacity);
28   /*@ close dmappingp(map, kp1, kp2, hsh1, hsh2,
29                       fvp, bvp, rof, vsz,
30                       vk1, vk2, rp1, rp2, m); @*/
31   return res;
32 }
```

Figure 8: Top-level `get` method in the libVig flow table data type. `index` is an output parameter for the index of the entry whose first key matches `key`. The method returns 1 if the entry is found, 0 otherwise.

Implementation: In Vigor, we use the VeriFast proof checker [28], which works for C programs annotated with pre-conditions and post-conditions written in separation logic [51]. Annotating code is not an easy task, especially for non-experts. However, separation logic is relatively friendly: It is an extension of classic Hoare logic designed for low-level imperative programs that use shared mutable data structures. It has a good notion of memory ownership, which makes it easy to express transfer of ownership through pointers. Separation logic supports local reasoning [44], in that specifications and proofs of a method refer only to the memory used by that method, not the entire global state.

libVig contains 2.2 KLOC of C, 4K lines of pre- and post-conditions and accompanying definitions, and 21.8K lines of proof code (inlined annotations). The human effort of writing the proof is about 2 person-months. VeriFast verifies the proof in less than 1 minute.

5.1.4 Symbolic models for libVig data types. Symbolic execution [6, 9, 14, 27, 33, 50] is the method we use for verifying VigNAT's low-level properties (P_2) in §5.2.1. This approach entails having a symbolic execution engine execute the NF with symbolic rather than concrete values. A symbolic value represents simultaneously multiple possible values (e.g., an unconstrained symbolic packet header represents all possible packet headers). Assignment statements are functions of their symbolic arguments, while conditional statements split execution into two paths, each with symbolic state correspondingly constrained by the branch condition.

When symbolically executing the VigNAT stateless code, which calls into libVig, we do not wish to also symbolically execute the libVig implementation, because that would lead to path explosion. Therefore, we abstract libVig with a *symbolic model* that simulates the effect of calling into libVig and keeps track of the side effects in a per-execution-path manner. The symbolic model differs from the formal contracts in two ways: it is executable code, and it may be imperfect—it might miss some possible behaviors of the libVig implementation, or exhibit behaviors that could never occur.

As we discuss in the next section, writing a good symbolic model is hard, and our lazy proof technique helps deal with this challenge: it tolerates imperfections in a symbolic model while at the same time formally guaranteeing that accepted imperfections do not affect the overall proof of the NF.

5.2 Lazy Proofs

Our proposed *lazy proofs* technique glues together a symbolic execution engine (SEE) with a proof checker to produce proofs of NF properties that were previously out of reach. Together with the stateful/stateless separation described earlier, this constitutes the cornerstone of how we verify VigNAT.

The main idea is to use an SEE to enumerate all execution paths through the stateless NF code, and (a) record for each path a symbolic trace of how the stateless code interacted with the outside world and with libVig; and (b) verify that P_2 (low-level properties) holds on each path. The Validator then transforms the symbolic traces (i.e., a representation of all possible observable behaviors of VigNAT's stateless code) into mechanically checkable proofs that P_4, P_5, and ultimately P_1 (NAT semantics) hold.

Not only do lazy proofs allow us to use the right tool for each desired property but they also resolve the modeling challenge: writing a symbolic model of libVig requires reconciling two conflicting objectives. On the one hand, the model must remove enough details, i.e., be abstract enough to make symbolic execution terminate in useful time—after all, it is abstraction that reduces the number of paths to explore symbolically. On the other hand, the model must be detailed enough to capture enough libVig behaviors to be faithful to the libVig implementation in the context of the properties being verified. How good the model is depends directly on which details are relevant to the proof vs. not, which in turn depends both on the properties to be proven and on the code that uses the model. Thus, devising a good model is often an iterative process that converges after multiple attempts onto a good model customized to the code and the property to be proven. Spending time proving the faithfulness (P_5) of draft models before actually knowing that they are fit for proving P_2 would be wasteful. With lazy proofs, we can now *first* attempt the proof of P_2 assuming the model is OK and, if the model indeed helps prove the desired property, only then invest in validating the model (P_5). From a practical standpoint, this approach makes it cheap to write models, because we don't need to spend time ironing out the very last bugs; instead we rely on Vigor to surface these bugs over time.

Said differently, lazy proofs exploit the fact that an application typically uses only a subset of the semantics offered by its libraries. So, instead of proving that the libVig model accurately captures all of libVig's semantics, we only prove that it does so for the semantics used by VigNAT.

We now describe the proof of P_2—low-level properties (§5.2.1), the Vigor Validator and how it uses symbolic traces to prove P_1—correctness of VigNAT's semantics (§5.2.2), and finally how the Validator proves correctness of the libVig symbolic model (§5.2.3) and correctness of how libVig is used by the stateless code (§5.2.4).

5.2.1 Proving that VigNAT satisfies low-level properties. Low-level coding mistakes, like the misuse of memory, can cause a program to crash or behave erratically, so proving the absence of such mistakes is essential to proving higher level semantic properties. As described in §4.2, the desired low-level properties refer to the absence of bugs such as buffer over/underflow, out-of-bounds memory accesses, double free, arithmetic over/underflow, and others.

To prove the absence of such bugs, Vigor performs *exhaustive* symbolic execution (ESE), using an SEE to enumerate all execution paths through the stateless part of the NF. The SEE explores all feasible branches at conditional statements, therefore ESE is fully precise: it enumerates only feasible paths, i.e., paths for which there exists a set of inputs that takes the program down that path, and does not miss any feasible paths. Low-level properties are stated as `assert`s, and for each feasible execution path the SEE reasons symbolically about whether there exists an input that could violate the `assert`. In order to make this approach feasible, Vigor first replaces all calls to libVig with calls to the libVig model; this abstracts away all state handling code, thereby removing almost all constructs that lead to path explosion, such as loosely constrained symbolic pointers. Next, we make the SEE aware of loop bounds by marking the loop guard of an infinite loop with `VIGOR_LOOP` and providing loop invariants, so that the SEE can transform the loops to avoid unnecessary loop unrolling (e.g., by havocing [1]). This eliminates the last source of path explosion in VigNAT.

If the `assert` for each low-level property holds on *every* feasible path during ESE, then we have a proof that that stateless code is free of low-level bugs, since ESE reasons about all possible inputs without enumerating those inputs. The formal proof that libVig behaves according to its interface contracts (§5.1.3) guarantees that libVig too is free of low-level bugs, otherwise its proof would not verify. This means that all VigNAT code satisfies the low-level properties. If this was not stateless code but a stateful program, ESE would likely not complete. Yet, in our case, the SEE checks all 108 paths through VigNAT's stateless code in less than 1 minute.

Of course, the proof makes certain basic assumptions, like compiler correctness (more in §5.4), and most importantly it assumes that the libVig model is correct (P_5) and the stateless code uses VigNAT data structures correctly (P_4). Verifying these assumptions a posteriori requires a Validator and symbolic traces, which we describe in the next subsection.

Implementation. In Vigor we use the KLEE SEE [9]. It checks out-of-the-box several low-level properties, and we add the checks from LLVM's undefined behavior sanitizers [42, 43, 57]. We modified KLEE in several ways: First, we added loop invariant support, and we enabled KLEE to automatically find the variables that may change inside a loop and havoc [1] them. The NF developer still has to manually insert the assertions and assumptions for the loop invariants, but KLEE can now use them to avoid enumerating unnecessary paths. Second, we added dynamic pointer access control by providing primitives that allow libVig developers to enable/disable dereferenceability of a pointer between libVig calls. Third, we added the ability to record symbolic traces, described next.

5.2.2 *Proving that VigNAT satisfies NAT RFC semantics.*

We think of the formalized NAT semantics as "trace properties": given a trace of the interaction between an NF and the outside world, what must hold true of the trace for it to have been generated by a correct NAT NF? More specifically, the NAT properties (shown in §4.1 and Fig. 6) are in the form of pre- and post-conditions for actions triggered by the arrival of a packet. The pre-conditions, expressed on the abstract NAT state plus the incoming packet, select which action applies. The corresponding post-condition states what must hold of the abstract state and the potential outgoing packet after the action is completed.

In order to verify that the desired properties hold, Vigor collects from the SEE a trace of each explored execution path. This trace summarizes how the VigNAT code interacted, during symbolic execution, with (a model of) the outside world, be it the libVig library or the DPDK framework. Since the traces have common prefixes, they form a tree—the NF's execution tree. In the context of this section, a *symbolic trace* is a path from the root of the execution tree to a node in the tree, be it an internal or a leaf node. In other words, the set of symbolic traces considered by Vigor consists of all execution path traces and all their prefixes.

Each trace has two parts: a sequence of calls that were made across the traced interface, and a set of constraints on symbolic program state. Fig. 9 shows a simple example of a trace for the code in Fig. 1 using the `ring` data structure. The seven calls in this trace result from the execution of lines 8→9→12→13→14→16 in Fig. 1. `loop_invariant_produce` and `loop_invariant_consume` are markers indicating the beginning and the end of a loop iteration. In the `ring_pop_front` call, `packet` is an output parameter pointing to the popped packet; the trace records its initial and final value.

The *constraints* section shows the relationship between the different symbols. In this simple example there is only one constraint: $y \neq 9$ is the result of the application of `packet_constraints` (Fig. 2) in the `ring` model. The initial value x is unconstrained.

The Validator now takes each trace, weaves into it the properties to be proven, and turns it into a verification task. This is a C program that contains the sequence of calls from the trace, enriched with metadata on the symbolic variables used as arguments and return values, as well as the constraints that describe the relationships between these symbolic variables at each point in the trace. The Validator also inserts lemmas into the trace, to help the proof checker. In essence, the Validator translates each symbolic trace into a proof that the trace satisfies the desired properties. It then passes the proof to the proof checker to verify it.

```
1 loop_invariant_produce(ring=[..]) ==> []
2 ring_full(ring=[..]) ==> true
3 ring_empty(ring=[..]) ==> false
4 can_send() ==> true
5 ring_pop_front(ring=[..],
6                packet={.port=:x:} --> {.port=:y:}) ==> []
7 send(packet={.port=:y:}) ==> []
8 loop_invariant_consume(ring=[..]) ==> []
9 --- constraints ---
10 :y: != 9
```

Figure 9: Symbolic trace for a path through the code in Fig. 1. Colons (`:val:`) designate a symbol, `-->` separates the input and output value of a pointer argument, `==>` marks the return value of a function call, `[..]` indicates omitted details.

Fig. 10 shows the Validator-transformed version of Fig. 9. The seven calls are now on lines 6, 8, 10, 12, 15, 18, and 22. The uninitialized arg_1 and arg_2 variables are unconstrained symbols initially. The constraints on return values recorded in Fig. 9 turn into the `@assume` statements on lines 9,11, and 13. The symbolic constraint from l.10 in Fig. 9 turns into the `@assume` on l.17 of the proof. These four `@assume` statements constitute the pre-condition of this symbolic trace. In order to set up the post-condition that needs to be verified, the Validator initializes special handle variables *packet_is_sent* and *sent_packet* on lines 19 and 20 to capture externally visible effects immediately after the `send()` call. Then it inserts the NF specification (semantic property) into the trace on ll. 24-26:

```
if (packet_is_sent) {
  assert(sent_packet->port != 9);
}
```

Vigor verifies that the NF spec holds after every loop iteration.

Once the proof checker completes all verification tasks received from the Validator, we have a proof that the trace properties weaved in by the Validator hold for all possible executions of the stateless code. Vigor proves that VigNAT satisfies the NAT specification by weaving the properties of §4.1 into the symbolic traces, similarly to ll. 24-26 in Fig. 10. Trace verification is highly parallelizable: to verify all 431 traces resulting from the 108 execution paths of stateless VigNAT takes 38 minutes on a single core and 11 minutes on a 4-core machine. As will be noted later, this verification time includes not only proving P_1 but also P_4 and P_5.

5.2.3 *Validating the libVig symbolic model.*

We say a symbolic model is *valid* if the behavior it exhibits is indistinguishable from the behavior of the libVig implementation captured by the formal interface contracts[4]. Any behaviors of the model that are not observed during ESE are irrelevant to its validity *for this particular*

[4] It is sufficient to show that the model over-approximates the contracts, because any proof that holds for the over-approximation holds for an exact model as well.

```
struct ring* arg1;
struct packet arg2;
bool packet_is_sent = false;
struct packet* sent_packet = NULL;

loop_invariant_produce(&arg1);
//@ open loop_invariant(&arg1);
bool ret1 = ring_full(arg1);
//@ assume(ret1 == true);
bool ret2 = ring_empty(arg1);
//@ assume(ret2 == false);
bool ret3 = can_send();
//@ assume(ret3 == true);
//@ close packetp(&arg2, packet(arg2.port));
ring_pop_front(arg1, &arg2);
//@ openpacketp(&arg2, _);
//@ assume(arg2.port != 9);
send(&arg2);
packet_is_sent = true;
sent_packet = &arg2;
//@ close loop_invariant(&arg1);
loop_invariant_consume(&arg1);

/*@ if (packet_is_sent) {
  assert(sent_packet->port != 9);
} @*/
```

Figure 10: The trace of Fig. 9, translated into a proof.

proof. This is the insight behind our lazy proof technique: it doesn’t matter whether a model is universally valid, but rather what matters is whether *the parts of the model used during symbolic execution* are valid—this is a much weaker property than universal validity, but is sufficient for our purposes. The symbolic traces capture all necessary information on how the model is used.

The technique for proving that the libVig model is consistent with the libVig interface contracts is similar to the one we use for proving NAT semantics. Only this time, instead of weaving the NAT pre- and post-conditions into the traces, the Validator weaves in the assertions for the given trace’s path constraint. It then asks the proof checker to verify whether the assertions hold based solely on the post-conditions of the libVig functions. If verification succeeds, then it means that, after each invocation of the libVig model, the outcome covers all possible outcomes prescribed by the libVig interface contracts.

A libVig model can be over-approximate, under-approximate, or both. The question that the Validator aims to answer is whether, for the particular NF and properties, the model is sufficiently accurate. If a model is “too under-approximate” for the desired proof, it will cause the validation phase to fail, because its narrow behavior does not cover the spectrum of behaviors allowed by the contracts. If it is “too over-approximate” for the target proof, it will either cause exhaustive symbolic execution (ESE) or validation to fail—the former if the model exhibits behavior that is too general and makes it impossible to verify the low-level properties, the latter if low-level properties verify but a loop invariant or a high level semantic property is violated. When ESE completes, we have proof that the low-level properties hold, as long as the model is valid. ESE failure means that either there is a violation of a low-level property or the model offered by libVig is not suitable—Vigor does its best to help the developer distinguish between the two, but there is still room for improvement. In either case, it’s back to the drawing board: either the NF developer needs to fix her bug, or the Vigor developer needs to alter the model.

Vigor also uses a model we wrote of the DPDK packet processing framework’s `send`, `receive`, and `free` calls. We do not formally validate this model, though there is no fundamental reason it cannot be done. We make it part of Vigor’s trusted computing base (§5.4).

5.2.4 Proving that VigNAT correctly uses libVig. There is one caveat to the proof in the previous section: if a libVig method implementation is invoked and the corresponding pre-condition does not hold, then the behavior of that method is undefined. For example, passing a null argument when the contract says it must be non-null could cause the implementation to crash, behave incorrectly, or behave correctly. It is therefore imperative that, in conjunction with validating the model’s behavior, Vigor also validate the caller’s behavior with respect to the interface contracts.

The method for proving that the VigNAT stateless code uses the libVig data structures consistently with the libVig interface contracts is the same as above, except that the Validator weaves in the pre-conditions contained in the libVig interface contracts. In fact, the Validator weaves in the NAT pre- and post-conditions and the libVig pre- and post-conditions in one go, and generates a single verification task per trace that simultaneously verifies properties P_1, P_4, and P_5.

The trickiest part in verifying the libVig pre-conditions is tracking memory ownership across the interface. A pointer returned by a libVig method (either as a return value or via an output parameter) references memory owned at first by libVig; upon return, ownership transfers to the caller. After using/modifying the pointer, the NF code calls another function to return ownership to libVig.

A pointer passed as an argument to a libVig method may be the address of a libVig data structure (equivalent to the `this` pointer in C++/Java or the `self` reference in Python). This type of pointer remains opaque to the stateless code: it can be copied, assigned, and compared for equality, but cannot be dereferenced. The Validator and proof checker need not look at the memory pointed to by such pointers but only keep track of aliasing information. The pointed-to memory is owned by libVig at all time. Vigor verifies that the stateless code obeys this pointer discipline during symbolic execution, using our addition to the SEE for enabling/disabling dereferenceability of a pointer between libVig calls.

A pointer used as an output parameter points to where the caller expects libVig’s return result to be written. In this case, the pointed-to memory is owned by the calling code. The Validator and proof checker trace the evolution of the pointed-to memory by including it in the function’s input set before the call and in the function’s output set after the call. A special case of output pointer is a double pointer to a library data structure (e.g., `X** p`) as appears in the data structure allocation functions. The VigNAT code owns the pointee `*p`, so the Validator tracks it, but the pointee of the pointee `**p` is a library data structure, thus the memory is owned by the library, so there is no need to track it. Vigor currently does not support other cases of double or deeper pointers.

Vigor also checks for memory leaks. Even though stateless code cannot dynamically allocate memory, leaks are possible if it uses libVig incorrectly, such as forgetting to call a release method. Unlike simple low-level properties (e.g., integer overflow) that can be stated as a simple `assert`, absence of memory leaks is a global property. Vigor therefore must keep track of memory ownership and validate that ownership is properly returned to libVig before the end of the execution. This facility, for example, caught an accidental memory leak in VigNAT where we failed to release DPDK memory corresponding to a packet returned by DPDK, thus violating the DPDK interface contracts.

5.3 The Vigor Workflow

The Vigor workflow described above can be summarized as follows: We split the NAT NF into a stateless and a stateful part, the latter contained in the libVig library. Then we use formal theorem proving to verify P_3—correctness of the data structures implemented in libVig. We use exhaustive symbolic execution (ESE) with a modified version of KLEE [9] to explore all paths in the stateless part (using symbolic models of the data structures) and verify P_2—low-level properties, like crash freedom, memory safety, and no overflows—as well as VigNAT's disciplined use of pointers. This step proceeds under the assumptions that the stateless code uses the libVig data types according to their interface contracts (P_4) and the libVig model is faithful to the libVig interface contracts (P_5) for the particular execution paths explored during ESE. Both of these assumptions we prove a posteriori using a combination of our Vigor Validator and the VeriFast proof checker [28]. Finally, we use this same combination of tools to prove VigNAT's semantic properties *(P_1)*, i.e., that it conforms to our formalization of RFC 3022 [53]. $P_1 \wedge P_2 \wedge P_3 \wedge P_4 \wedge P_5$ together formally prove VigNAT's correctness, under the assumptions described in the next section.

5.4 Assumptions

The trusted computing base for a Vigor-verified NF consists of the Vigor toolchain (the Clang LLVM compiler, VeriFast, KLEE, and our own Validator) and the environment in which the NF runs (DPDK, device drivers, OS kernel, BIOS, and hardware). We assume that the compiler implements the same language semantics employed by Vigor (e.g., same byte length for C primitive types). We wrote symbolic models for three DPDK functions and for system time, which as of this writing we have not verified. They are small (about 400 LOC), so we convinced ourselves manually that they are correct over-approximations. One could envision adopting an environment that has a formal specification, like seL4 [34], in which it becomes possible to prove the validity of these models.

6 PERFORMANCE EVALUATION

Having shown in previous sections how we verify VigNAT's correctness, we now demonstrate that this formal verification does not come at the cost of performance: compared to an unverified NAT written on top of DPDK, our verified NAT offers similar latency and less than 10% throughput penalty. We focus our evaluation on comparing VigNAT (labeled **Verified NAT** in the graphs) to three other NFs:

(a) **No-op forwarding** is implemented on top of DPDK; it receives traffic on one port and forwards it out another port without any other processing. It serves as a baseline that shows the best throughput and latency that a DPDK NF can achieve in our experimental environment.

(b) **Unverified NAT** is also implemented on top of DPDK; it implements the same RFC as VigNAT and supports the same number of flows (65,535), but uses the hash table that comes with the DPDK distribution. It was written by an experienced software developer with little verification expertise, different from the one who wrote and verified VigNAT. It serves to compare VigNAT to a NAT that was not written with verification in mind.

(c) **Linux NAT** is NetFilter [5], set up with straightforward masquerade rules and tuned for performance [29]. We expect it to be significantly slower than the other two, because it does not benefit from DPDK's optimized packet reception and transmission. We use it to make the point that VigNAT performs significantly better than the typical NAT currently used in Linux-based home and small-enterprise routers, as one would expect from a DPDK NAT.

We use the testbed shown in Fig. 11 as suggested by RFC 2544 [7]. The Tester and the Middlebox machines are identical, with an Intel Xeon E5-2667 v2 processor at 3.30 GHz, 32 GB of DRAM, and 82599ES 10 Gbps DPDK-compatible NICs. The Middlebox machine runs one of the four NFs mentioned above (we use one core). The Tester machine runs MoonGen [22] to generate traffic and measure packet loss, throughput, and latency; for the latency measurements, we rely on hardware timestamps for better accuracy [49]. We use DPDK v.16.07 on Ubuntu Linux with kernel 3.13.0-119-generic.

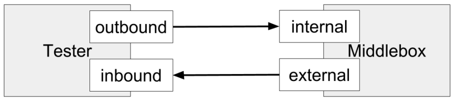

Figure 11: Testbed topology for performance evaluation.

First, we measure the latency experienced by packets between the Tester's outbound and inbound interfaces. We first run a set of experiments in which all the NATs are configured to expire flows after 2 seconds of inactivity. In each experiment, the Tester generates 10–64,000 "background flows," which produce in total 100,000 pps and never expire throughout the experiment, and 1,000 "probe flows," which produce 0.47 pps and expire after every packet. We use the background flows to control the occupancy of the flow table, while we measure the latency of the packets that belong to probe flows. We focus on the probe flows because, from a performance point of view, they are the worst-case scenario for a NAT NF: each of their packets causes the NAT to search its flow table for a matching flow ID, not find any match, and create a new entry.

Fig. 12 shows the average latency experienced by the probe flows as a function of the number of background flows, for the three DPDK NFs: the Verified NAT (5.13μsec) has 2% higher latency than the Unverified NAT (5.03μsec), and 8% higher than No-op forwarding (4.75μsec). So, on top of the latency due to packet reception and transmission, the Unverified and Verified NAT add, respectively, 0.28μsec and 0.38μsec of NAT-specific packet processing. For all three NFs, latency remains stable as flow-table occupancy grows, which shows that the two NATs use good hash functions to spread the load uniformly across their tables. The only case where latency increases (to 5.3μsec) is for the Verified NAT, when the flow table becomes almost completely full (the green line curves upward at the last data point). The Linux NAT has significantly higher latency (20μsec).

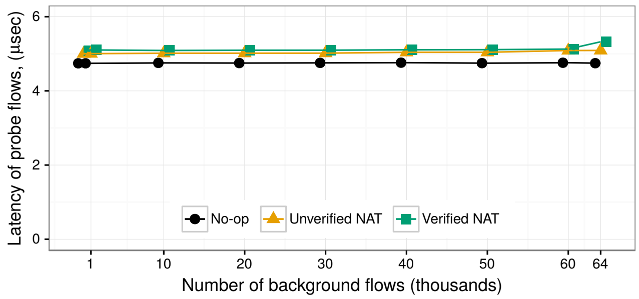

Figure 12: Average latency for probe flows. Confidence intervals are approximately 20 nanosec, not visible at this scale.

To give a sense of latency variability, Fig. 13 shows the complementary cumulative distribution function (CCDF) of the latency experienced by the probe flows, when there are 60,000 background flows (i.e., 92% occupancy): the Verified NAT has a slightly heavier tail than the Unverified NAT; all three NFs have outliers that are two orders of magnitude above the average, but these are due to DPDK packet processing, not NAT-specific processing (the three curves coincide for latency exceeding 6.5μsec). The CCDFs computed for different numbers of background flows look similar.

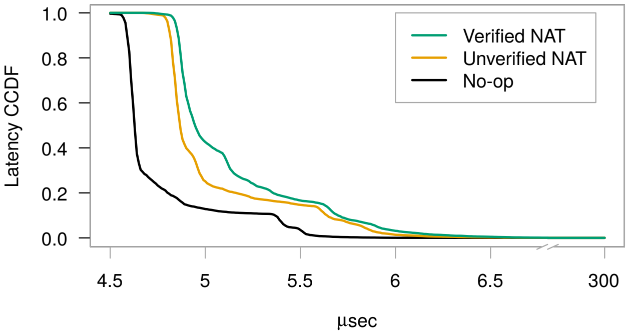

Figure 13: Latency CCDF for probe flows.

We get similar results in a second set of experiments, where the Tester produces the same flow mix as before, but the NATs are configured to expire flows after 60 seconds of inactivity (hence neither the probe flows nor the background flows ever expire). In this case, the average latency of the Verified NAT is slightly lower (5.07μsec), while that of the Unverified NAT the same as before (5.03μsec).

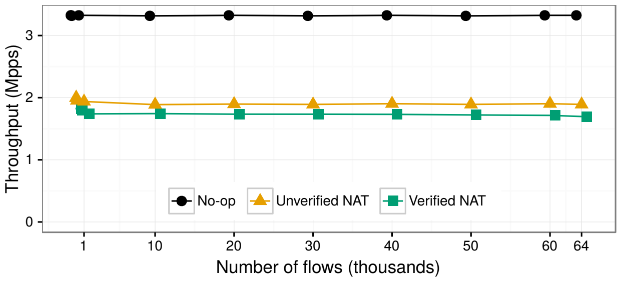

Figure 14: Maximum throughput with a maximum loss rate of 0.1%.

Finally, we measure the highest throughput achieved by each NF. In each experiment, the Tester generates a fixed number of flows that never expire, each producing 64-byte packets at a fixed rate, and we measure throughput and packet loss. During all experiments, the Middlebox is CPU bound. Fig. 14 shows the maximum throughput achieved by each NF with less than 0.1% packet loss, as a function of the number of generated flows. The Verified NAT (1.8 Mpps) has 10% lower throughput than the Unverified NAT (2 Mpps). This difference in throughput comes from the difference in NAT-specific processing latency (0.38μsec vs. 0.28μsec) imposed by the two NATs: in our experimental setup, this latency difference cannot be masked, as each NF runs on a single core and processes one packet at a time. The Linux NAT achieves significantly lower throughput (0.6 Mpps).

In essence, these results indicate that the performance of the libVig flow table (which has a formal specification and proof) is close to that of the DPDK hash table (which has neither), though not the same. The implementations of the two data structures are quite different. We did not try to reuse/adapt the implementation of the DPDK hash table, because it resolves hash conflicts through separate chaining—items that hash to the same array position are added to the same linked list—a behavior that is hard to specify in a formal contract. Instead, the libVig flow table resolves conflicts through open addressing: if an item hashes to an occupied array position, it is stored in the next array position that is free, together with auxiliary metadata that speeds up lookup. We have not yet optimized our implementation at the instruction level, so it has an overall slower access time because there are, on average, more candidate memory locations for each item[5]; the difference is greatest for lookups that find no match, because these result in searching all candidate memory locations.

In summary, our experimental evaluation shows that it is possible to have a NAT network function that both offers competitive performance and is formally verified.

7 DISCUSSION

Developing and verifying VigNAT is a first step toward the broader goal of verified, high-performance NFs. The Vigor approach and prototype have several limitations, and also offer opportunities for future research. In this section, we describe some of these.

[5] This is the case for the particular implementations used, respectively, by the Unverified and Verified NAT, not for separate chaining and open addressing in general.

Vigor will not produce an incorrect proof, but it may fail to prove a property that actually holds for a given NF, because of an invalid model. For example, in §3, if Vigor uses model (b) from Fig. 4, it cannot prove that the given NF correctly implements the discard protocol—even though that is the case—because the model is too abstract. So, if a proof fails, and the reported reason for the failure does not lead the NF developer to a bug in their code, it may be that the given NF exercises libVig functionality that is not properly captured by a symbolic model; in this case, the NF developer can request from the libVig developers a more detailed model.

The current version of Vigor cannot verify concurrent code. We expect that the biggest challenge will be the development and verification of useful concurrent data structures.

We do not have yet experience with applying Vigor to mature, legacy code. Such software often has state handling code sprinkled throughout, so refactoring it to put all state in libVig data structures could be challenging. It would also require annotating loops and extracting loop invariants, but we expect to be able to automate these tasks using known techniques [23, 47].

More generally, we hope to reduce the human effort needed to expand libVig and use Vigor by automating most tasks: (a) Many lemmas needed for Step 3 are boiler-plate, and we should be able to generate them automatically. (b) For proving low-level properties, it may be enough to use simple over-approximate models that leave outputs unconstrained, and such models can be generated automatically. We can also leverage techniques that learn invariants from traces [13, 45] to refine symbolic models or produce initial drafts for libVig contracts. (c) Much of the effort in verifying libVig goes into writing intermediate lemmas in order to bridge logical leaps that VeriFast cannot make on its own. Using a proof assistant or a more powerful theorem prover would reduce this effort [4, 12]. (d) Generating formal contracts that specify standards (e.g., as described in an RFC) will always require manual effort; however, standards often have structure that is amenable to natural language processing, so we could, perhaps, employ automated techniques [37] to generate first drafts, which can then be refined by humans.

8 RELATED WORK

As described in §2, our work on VigNAT falls in the area of *NF verification*, under the broader umbrella of "data-plane verification."

The closest work to ours is that of Dobrescu et al. [19], which verified NFs written in Click [35], including a NAT. They proved the low-level properties of crash freedom and bounded execution for these NFs. Their approach relies on exhaustive symbolic execution of individual Click elements and on-demand composition of the resulting analyses to reason about Click pipelines. Like Vigor, their approach puts all state in special data structures, however, it does not verify the data structures themselves nor that the NF uses them correctly. It is for this reason that Dobrescu et al.'s work cannot prove semantic properties—the step forward that enables such proofs is the "lazy proofs" technique we described here.

Orthogonally to NF verification is what we refer to as *network verification*: verify network properties (reachability, loops, etc.) of a combination of *modeled* network devices. There is a lot of prior work in this area [24, 25, 30–32, 38, 39, 46, 52, 55, 59].

Stoenescu et al. [55] focus on network verification, but nevertheless rely on more detailed NF models than other work in this area, and they test their models for faithfulness to the corresponding NF implementations. Vigor also relies on models (not of entire NFs but of state-accessing functions), but it does not rely on testing to gain trust in the models' faithfulness. Instead, Vigor formally verifies that the models are guaranteed to be valid for the proof, which means that replacing a model with the corresponding implementation preserves the proof, modulo the assumptions in §5.4.

Many before us have applied verification techniques to networked and distributed systems. We share tools and techniques with this work—symbolic execution, formal contracts, proof checkers—but the approach of using different verification techniques for different parts of the code, and combining the results through lazy proofs, is novel. Musuvathi et al. [41] tested the Linux TCP implementation for conformance to a formal specification. Bishop et al. [3] tested several implementations of TCP/IP and the sockets API for conformance to a formal specification. Kuzniar et al. [36] tested OpenFlow switches for interoperability with reference implementations. Hawblitzel et al. [26] verified network applications written in Dafny—a high-level language with built-in verification support. Beringer et al. [2] verified an OpenSSL implementation, proving functional correctness and cryptographic properties.

There exists a body of prior work that has made significant progress in verifying properties of systems software. Much of this work can be applied to NF verification as well. For example, seL4 [34], CompCert [54], and FSCQ [11] show how to prove semantic properties of systems. Unfortunately, they all require the use of high-level (sometimes esoteric) programming languages and deep expertise in verification, which we consider a high barrier to adoption. The motivation behind Vigor is to make the verification of network functions accessible to most (ideally all) NF developers.

9 CONCLUSION

We presented a NAT box along with a technique and toolchain for proving that it is semantically correct according to a formal interpretation of RFC 3022. Our main contribution is exploiting the specifics of NF structure to propose a new verification technique that stitches together exhaustive symbolic execution with formal proof checking based on separation logic. This technique, called "lazy proofs," can scalably prove both low-level and semantic properties of our NF. Experimental results demonstrate the practicality of our approach: the verified NAT box performs as well as an unverified DPDK NAT and outperforms the standard Linux NAT. We hope our technique will eventually generalize to proving properties of many other software NFs, thereby amortizing the tedious work that has gone into building a library of verified NF data structures.

ACKNOWLEDGMENTS

We thank Jonas Fietz, Vova Kuznetsov, Christian Maciocco, Mia Primorac, Martin Vassor, and Jonas Wagner for insightful technical discussions, as well as the members of the Network Architecture Lab and Dependable Systems Lab at EPFL for valuable feedback. We thank our shepherd, Arjun Guha, and the anonymous reviewers for their comments and guidance. This work is supported by a Swiss National Science Foundation Starting Grant and an Intel grant.

REFERENCES

[1] Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects* (2005).

[2] Beringer, L., Petcher, A., Katherine, Q. Y., and Appel, A. W. Verified Correctness and Security of OpenSSL HMAC. In *USENIX Security Symp.* (2015).

[3] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., and Wansbrough, K. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as applied to TCP, UDP, and Sockets. *SIGCOMM Computer Communication Review 35*, 4 (2005).

[4] Blanchette, J., Bulwahn, L., and Nipkow, T. Automatic Proof and Disproof in Isabelle/HOL. *Frontiers of Combining Systems* (2011).

[5] Boye, M. Netfilter Connection Tracking and NAT Implementation. In *Seminar on Network Protocols in Operating Systems* (2013).

[6] Boyer, R. S., Elspas, B., and Levitt, K. N. SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution. *SIGPLAN Notices 10* (1975).

[7] Bradner, S., and McQuaid, J. Benchmarking Methodology for Network Interconnect Devices. RFC 2544, RFC Editor, 1999.

[8] Brocade Vyatta Network OS. http://www.brocade.com/en/products-services/software-networking/network-functions-virtualization/vyatta-network-os.html. Accessed: 2017-01-24.

[9] Cadar, C., Dunbar, D., Engler, D. R., et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symp. on Operating Sys. Design and Implem.* (2008).

[10] Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. A NICE Way to Test OpenFlow Applications. In *Symp. on Networked Systems Design and Implem.* (2012).

[11] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Symp. on Operating Systems Principles* (2015).

[12] Chlipala, A. Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic. *SIGPLAN Notices 46*, 6 (2011).

[13] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-Guided Abstraction Refinement. In *Intl. Conf. on Computer Aided Verification* (2000).

[14] Clarke, L. A. A Program Testing System. In *Annual ACM Conf.* (1976).

[15] CVE-2013-1138. Available from CVE Details, CVE-ID CVE-2013-1138., 2013.

[16] CVE-2014-3817. Available from CVE Details, CVE-ID CVE-2014-3817., 2014.

[17] CVE-2015-6271. Available from CVE Details, CVE-ID CVE-2015-6271., 2015.

[18] CVE-2014-9715. Available from CVE Details, CVE-ID CVE-2014-9715., 2014.

[19] Dobrescu, M., and Argyraki, K. Software Dataplane Verification. In *Symp. on Networked Systems Design and Implem.* (2014).

[20] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *Symp. on Operating Systems Principles* (2009).

[21] Data Plane Development Kit. http://dpdk.org. Accessed: 2017-06-16.

[22] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., and Carle, G. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference* (2015).

[23] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering 27*, 2 (2001).

[24] Fayaz, S. K., Yu, T., Tobioka, Y., Chaki, S., and Sekar, V. BUZZ: Testing Context-Dependent Policies in Stateful Networks. In *Symp. on Networked Systems Design and Implem.* (2016).

[25] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. A General Approach to Network Configuration Analysis. In *Symp. on Networked Systems Design and Implem.* (2015).

[26] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving Practical Distributed Systems Correct. In *Symp. on Operating Systems Principles* (2015).

[27] Howden, W. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering 3*, 4 (1977).

[28] Jacobs, B., and Piessens, F. The VeriFast Program Verifier. Tech. Rep. CW-520, Department of Computer Science, KU Leuven, 2008.

[29] Kadlecsik, J., and Pásztor, G. Netfilter Performance Testing. Tech. rep., Netfilter Project, Berlin, Germany, 2004.

[30] Kazemian, P., Chan, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S. Real Time Network Policy Checking Using Header Space Analysis. In *Symp. on Networked Systems Design and Implem.* (2013).

[31] Kazemian, P., Varghese, G., and McKeown, N. Header Space Analysis: Static Checking For Networks. In *Symp. on Networked Systems Design and Implem.* (2012).

[32] Khurshid, A., Zou, X., Zhou, W., Caesar, M., and Godfrey, P. B. VeriFlow: Verifying Network-Wide Invariants in Real Time. In *Symp. on Networked Systems Design and Implem.* (2013).

[33] King, J. C. Symbolic Execution and Program Testing. *J. ACM 19*, 7 (1976).

[34] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., et al. seL4: Formal Verification of an OS Kernel. In *Symp. on Operating Systems Principles* (2009).

[35] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click Modular Router. *ACM Transactions on Computer Systems 18*, 3 (2000).

[36] Kuzniar, M., Peresini, P., Canini, M., Venzano, D., and Kostic, D. A SOFT Way for OpenFlow Switch Interoperability Testing. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012).

[37] Lee, B.-S., and Bryant, B. R. Automated Conversion from Requirements Documentation to an Object-oriented Formal Specification Language. In *Symposium on Applied Computing* (2002).

[38] Lopes, N. P., Bjørner, N., Godefroid, P., Jayaraman, K., and Varghese, G. Checking Beliefs in Dynamic Networks. In *Symp. on Networked Systems Design and Implem.* (2015).

[39] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T. Debugging the Data Plane with Anteater. In *ACM SIGCOMM Conf.* (2011).

[40] MS13-064. Available from CVE Details, CVE-ID MS13-064., 2013.

[41] Musuvathi, M., Engler, D. R., et al. Model Checking Large Network Protocol Implementations. In *Symp. on Networked Systems Design and Implem.* (2004), vol. 4.

[42] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. *SIGPLAN Notices 44*, 6 (2009).

[43] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. CETS: Compiler Enforced Temporal Safety for C. *SIGPLAN Notices 45*, 8 (2010).

[44] O'Hearn, P. W., Reynolds, J. C., and Yang, H. Local Reasoning about Programs that Alter Data Structures. In *CSL* (2001).

[45] Padhi, S., Sharma, R., and Millstein, T. Data-Driven Precondition Inference with Learned Features. In *Intl. Conf. on Programming Language Design and Implem.* (2016).

[46] Panda, A., Lahav, O., Argyraki, K., Sagiv, M., and Shenker, S. Verifying Reachability in Networks with Mutable Datapaths. In *Symp. on Networked Systems Design and Implem.* (2017).

[47] Perkins, J. H., and Ernst, M. D. Efficient Incremental Algorithms for Dynamic Detection of Likely Invariants. *ACM SIGSOFT Software Engineering Notes 29*, 6 (2004).

[48] Postel, J. Discard Protocol. RFC 863, RFC Editor, 1983.

[49] Primorac, M., Argyraki, K., and Bugnion, E. How to Measure the Killer Microsecond. In *ACM SIGCOMM Workshop on Kernel-Bypass Networks* (2017).

[50] Ramamoorthy, C., Ho, S.-B., and Chen, W. On the Automated Generation of Program Test Data. *IEEE Transactions on Software Engineering 2*, 4 (1976).

[51] Reynolds, J. C. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE-LCS* (2002).

[52] Ryzhyk, L., Bjørner, N., Canini, M., Jeannin, J.-B., Schlesinger, C., Terry, D. B., and Varghese, G. Correct by Construction Networks Using Stepwise Refinement. In *Symp. on Networked Systems Design and Implem.* (2017).

[53] Srisuresh, P., and Egevang, K. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, RFC Editor, 2001.

[54] Stewart, G., Beringer, L., Cuellar, S., and Appel, A. W. Compositional CompCert. *SIGPLAN Notices 50*, 1 (2015).

[55] Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. SymNet: scalable symbolic execution for modern networks. In *ACM SIGCOMM Conf.* (2016).

[56] Tange, O. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine 36*, 1 (2011).

[57] Clang 5 documentation - UndefinedBehaviorSanitizer - Available checks. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#available-checks. Accessed: 2017-01-24.

[58] VigNAT Project Repository. https://vignat.github.io, 2017.

[59] Xie, G. G., Zhan, J., Maltz, D. A., Zhang, H., Greenberg, A., Hjalmtysson, G., and Rexford, J. On Static Reachability Analysis of IP Networks. In *Intl. Conf. on Computer Communications (INFOCOM)* (2005).

[60] Zaostrovnykh, A., Argyraki, K., and Candea, G. Nework software verification survey. https://vignat.github.io/survey, Feb. 2016.