VigNAT: A Formally Verified NAT

Arseniya Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, George Candea

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Formally verify a stateful NF
Formally verify a stateful NF with competitive performance
Formally verify a stateful NF with competitive performance and reasonable human effort.
Software Network Functions: Pros and Cons

● Everywhere
  ○ OpenWRT/NetFilter, Click, RouteBricks
  ○ Vyatta, OpenVswitch, DPDK

● Flexibility, short time to market, but ...
Software Network Functions: Pros and Cons

- Everywhere
  - OpenWRT/NetFilter, Click, RouteBricks
  - Vyatta, OpenVswitch, DPDK

- Flexibility, short time to market, but ...

- Bugs
  - Packets of death, table exhaustion, denial of service
  - Cisco NAT, Juniper NAT, NetFilter, Windows ICS
  - Network outages already cost up to $700B/year
Testing: Easy but Incomplete
Testing: Easy but Incomplete
Formal Verification: Complete but Expensive
Formal Verification: Complete but Expensive
Network Verification
Network Verification
Network Verification ≠ NF Code Verification

```c
int ret = rte_eal_init(argc, argv);
if (ret < 0) {
    rte_exit(EXIT_FAILURE, "Error with EAL initialization, ret=%d\n", ret);
}
argc -= ret;
argv += ret;
nf_config_init(argc, argv);
nf_prns_config();

// Create a memory pool
unsigned nb_devices = rte_eth_dev_count();
struct rte_mempool* mbuf_pool = rte_mempool_create("MEMPOOL",
    "MEMPOOL_BUFSIZE", nb_devices, // #elements
    "MEMPOOL_CACHE", // cache size
    0, // #pool for private array size
    "RTE_MBUF_DEFAULT_BUFFER_SIZE", // data buffer size
    rte_socket_id());
if (mbuf_pool == NULL) {
    rte_exit(EXIT_FAILURE, "Cannot create mbuf pool: %s\n",
        rte_strerror(rte_errno));
}
clone_pool = rte_mempool_clone("clone_pool", MEMPOOL_CLONE_COUNT,
    32,
    0, 0, rte_socket_id());
if (clone_pool == NULL) {
    rte_exit(EXIT_FAILURE, "Cannot create mbuf clone pool: %s\n",
        rte_strerror(rte_errno));
}

// Initialize all devices
for (uint8_t device = 0; device < nb_devices; device++) {
    if (nf_init_device(device, mbuf_pool) == 0) {
        NF_INFO("Initialized device %" PRIu8 ", device");
    } else {
        rte_exit(EXIT_FAILURE, "Cannot init device %" PRIu8 ", device");
    }
}
```
How to Verify an NF (Before Vigor)?
How to Verify an NF (Before Vigor)?
How to Verify an NF (Before Vigor)?
How to Verify an NF (Before Vigor)?

<table>
<thead>
<tr>
<th>Bits</th>
<th>Algebras</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human effort</td>
<td>Machine effort</td>
</tr>
</tbody>
</table>

- Human effort: quick (workable)
- Machine effort: slow (infeasible)
Theorem Proving

<table>
<thead>
<tr>
<th></th>
<th>Bits</th>
<th>Algebras</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human effort</td>
<td>low</td>
<td>high / infeasible</td>
</tr>
<tr>
<td>Machine effort</td>
<td>low</td>
<td>low</td>
</tr>
</tbody>
</table>
Theorem Proving

Human effort

Machine effort

Bits

Algebras

high / infeasible

low


Exhaustive Symbolic Execution (SymbEx)

<table>
<thead>
<tr>
<th>Human effort</th>
<th>Bits</th>
<th>Algebras</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td></td>
<td>high / infeasible</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine effort</th>
<th>Bits</th>
<th>Algebras</th>
</tr>
</thead>
<tbody>
<tr>
<td>high / infeasible</td>
<td></td>
<td>low</td>
</tr>
</tbody>
</table>
### Exhaustive Symbolic Execution (SymbEx)

<table>
<thead>
<tr>
<th></th>
<th>Bits</th>
<th>Algebras</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human effort</td>
<td>low</td>
<td>high / infeasible</td>
</tr>
<tr>
<td>Machine effort</td>
<td>high / infeasible</td>
<td>low</td>
</tr>
</tbody>
</table>

Path Explosion

Credit to Jonas Wagner
## Vigor

<table>
<thead>
<tr>
<th>Human effort</th>
<th>Bits</th>
<th>Algebras</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td></td>
<td>high / infeasible</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine effort</th>
<th>Bits</th>
<th>Algebras</th>
</tr>
</thead>
<tbody>
<tr>
<td>high / infeasible</td>
<td></td>
<td>low</td>
</tr>
</tbody>
</table>
### Vigor

<table>
<thead>
<tr>
<th></th>
<th>Bits</th>
<th>Algebras</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human effort</td>
<td>low</td>
<td>high / infeasible</td>
</tr>
<tr>
<td>Machine effort</td>
<td>high / infeasible</td>
<td>low</td>
</tr>
</tbody>
</table>

Plus runtime performance
Main Idea

- Split the code into two parts
- Verify each part separately
- Stitch the proofs — key challenge
Outline

● Problem Statement

● VigNAT Formal Proof
  ○ General Idea
  ○ Proof Stitching Example

● RFC Formalization

● Performance
int ret = rte_eal_init(argc, argv);
if (ret < 0) {
    rte_exit(EXIT_FAILURE, "Error with EAL initialization, ret=%d\n", ret);
}
argc -= ret;
argv += ret;

nf_config_init(argc, argv);
nf_print_config();

// Create a memory pool
unsigned nb_devices = rte_eth_dev_count();
struct rte_mempool *mempool = rte_pktmbuf_pool_create("MEMPOOL", nb_devices, 128, 0, 0, 0, rte_socket_id);
// #elements
MEMPOOL_BUCKET_SIZE = 1, // cache size
0, // alignment private elem size
RTE_MBUFSIZE = 64, // data buffer size
rte_socket_id);
if (mbuf_pool_resource < nb_devices) {
    rte_exit(EXIT_FAILURE, "Cannot create mbuf pool\n",
}
cREATE_POOL("clone_pool", MEMPOOL_CLONE_COUNT, 32, 0, 0, rte_socket_id());
if (clone_pool_resource < nb_devices) {
    rte_exit(EXIT_FAILURE, "Cannot create mbuf clone pool\n”,
}
yerror(rte_errno));

// Initialize all devices
for (uint8_t device = 0; device < nb_devices; device++) {
    if (nf_init_device(device, mbuf_pool) == 0) {
        NF_INFO("Initialized device %" PRIu8 ", device");
    } else {
        rte_exit(EXIT_FAILURE, "Cannot init device %" PRIu8 ", device");
    }
}
Vigor: split code | verify parts | stitch proofs

```c
int ret = rte_eal_init(argc, argv);
if (ret < 0) {
    rte_exit(EXIT_FAILURE, "Error with EAL initialization, ret=%d\n", ret);
}
argc -= ret;
argv += ret;

nf_config_init(argc, argv);
nf_print_config();

// Create a memory pool
unsigned nb_devices = rte_eth_dev_count();
struct rte_mempool* mbuf_pool = rte_pktmbuf_pool_create(
    "MEMPOOL", // name
    MEMPOOL_BUFFER_COUNT * nb_devices, // #elements
    MEMPOOL_CACHE_SIZE, // cache size
    0, // application private area size
    RTE_MBUF_DEFAULT_BUF_SIZE, // data buffer size
    rte_socket_id() // socket ID
); // end of for
if (mbuf_pool == NULL) {
    rte_exit(EXIT_FAILURE, "Cannot create mbuf pool\n");
}
clone_pool = rte_pktmbuf_pool_create("clone_pool", MEMPOOL_CLONE_COUNT,
    32,
    0, 0, rte_socket_id());
if (clone_pool == NULL) {
    rte_exit(EXIT_FAILURE, "Cannot create mbuf clone pool: %s\n",
        rte_strerror(rte_errno));
}

// Initialize all devices
for (uint8_t device = 0; device < nb_devices; device++) {
    if (nf_init_device(device, mbuf_pool) == 0) {
        NF_INFO("Initialized device %", PRIu8 ".", device);
    } else {
        rte_exit(EXIT_FAILURE, "Cannot init device %", PRIu8 ",", device);
    }
}
```
Vigor: **split code** | verify parts | stitch proofs
Vigor: split code | verify parts | stitch proofs

Stateful code (data structures)

Interface contracts

Stateless code (application logic)
Vigor: split code | **verify parts** | stitch proofs

Stateful code (data structures)

Interface contracts

Theorem Proving
Vigor: split code | **verify parts** | stitch proofs

- **Stateful code** (data structures)
- **Interface contracts**
- **Stateless code** (application logic)
Vigor: split code | verify parts | stitch proofs

- Stateful code (data structures)
- Interface contracts
- Stateless code (application logic)
- Exhaustive Symbolic Execution
Vigor: split code | **verify parts** | stitch proofs

Approximation (not trusted)

Stateless code (application logic)

Symbolic models

Exhaustive Symbolic Execution
Vigor: split code | verify parts | stitch proofs

Approximation (not trusted) --> Symbolic models

Stateless code (application logic)

Exhaustive Symbolic Execution

traces
Vigor: split code | verify parts | stitch proofs
Vigor: split code | verify parts | stitch proofs
Vigor: split code | verify parts | stitch proofs

Stateful code (data structures)

Interface contracts

Stateless code (application logic)

Theorem Proving

Exhaustive Symbolic Execution

traces

Vigor Validator
Outline

● Problem Statement
● VigNAT Formal Proof
  ○ General Idea
  ○ Proof Stitching Example
● RFC Formalization
● Performance
Proof Stitching: SymbEx + Theorem Proving

- Stateful code: theorem proving
- Stateless code: exhaustive symbolic execution

1. Use symbolic models — rough interpretations of contracts
   - Symbolic models are written in C

2. Replay call traces in a proof checker to check contracts
Example NF Code

```c
if (!ring_full(r) && receive(&p) && p.port != 9)
    ring_push_back(r, &p);

if (!ring_empty(r) && can_send()) {
    ring_pop_front(r, &p);
    send(&p);
}
```
Example NF Code

```c
if (!ring_full(r) && receive(&p) && p.port != 9)
    ring_push_back(r, &p);
if (!ring_empty(r) && can_send()) {
    ring_pop_front(r, &p);
    send(&p);
}
```
Example NF Code

```c
if (!ring_full(r) && receive(&p) && p.port != 9)
    ring_push_back(r, &p);
if (!ring_empty(r) && can_send()) {
    ring_pop_front(r, &p);
    send(&p);
}
```
For Each API Function ...
Example: Formal Contract

```c
void ring_pop_front(struct ring* r, struct packet* p);
```

- `r` is not empty and
- `p` points to valid memory

____________________________________

- `r` contains one packet less and
- `p` points to a packet and `p->port ≠ 9`
Example: Symbolic Model

```c
void ring_pop_front(struct ring* r, struct packet* p) {
    FILL_SYMBOLIC(p, sizeof(struct packet), "popped_packet");
    ASSUME(p->port != 9);
}
```
Example NF Code

```c
if (!ring_full(r) && receive(&p) && p.port != 9)
    ring_push_back(r, &p);
if (!ring_empty(r) && can_send()) {
    ring_pop_front(r, &p);
    send(&p);
}
```
Use Symbolic Models

```c
if (!ring_full(r) && receive(&p) && p.port != 9)
    ring_push_back(r, &p);
if (!ring_empty(r) && can_send()) {
    ring_pop_front(r, &p);
    send(&p);
}
```
if (!ring_full(r) && receive(&p) && p.port != 9)  
    ring_push_back(r, &p);
if (!ring_empty(r) && can_send()) {
    ring_pop_front(r, &p);
    send(&p);
}
Execution Trace

```c
if (!ring_full(r) && receive(&p) && p.port != 9)
    ring_push_back(r, &p);
if (!ring_empty(r) && can_send())
    ring_pop_front(r, &p); after(p.port != 9)
    send(&p);
```
Over-Approximation Proof

r1 = ring_full(r); assume(r1 == true);

r2 = ring_empty(r); assume(r2 == false);
ring_pop_front(r, &p);
assert(p.port ≠ 9);
Over-Approximation Proof

\[
\text{r1 = ring_full(r); assume(r1 == true);} \\
\text{r2 = ring_empty(r); assume(r2 == false);} \\
\text{ring_pop_front(r, &p);} \\
\text{assert(p.port ≠ 9);} \\
\]
Over-Approximation Proof

\[ r_1 = \text{ring\_full}(r); \ \text{assume}(r_1 == \text{true}); \]

\[ r_2 = \text{ring\_empty}(r); \ \text{assume}(r_2 == \text{false}); \]
\text{ring\_pop\_front}(r, \ &p);
\text{assert}(p.\text{port} \neq 9);
Outline

- Problem Statement
- VigNAT Formal Proof
  - General Idea
  - Proof Stitching Example
- RFC Formalization
- Performance
Formalization of the NAT RFC

- Everything happens at packet arrival
- Abstract flow table summarizes history of previous interactions
- Packet arrival timestamps — the only source of time
Formalization of the NAT RFC
Formalization of the NAT RFC
Formalization of the NAT RFC

\{\text{flowtable} \text{ before}, \text{ time}, \text{ packet}_\text{in}\} \rightarrow \{\text{flowtable} \text{ after}, \text{ packet}_\text{out}\}
Formalization of the NAT RFC

- flowtable
- expire_flows
- update/create flow
- forward/drop
- flowtable
- packet
- time
Outline

● Problem Statement

● VigNAT Formal Proof
  ○ General Idea
  ○ Proof Stitching Example

● RFC Formalization

● Performance
Performance

No-op (DPDK)

Unverified NAT (DPDK)

VigNAT (DPDK)

Linux NAT (NetFilter)
Performance

No-op (DPDK)

Unverified NAT (DPDK)

VigNAT (DPDK)

Linux NAT (NetFilter)

Latency

4.63 μsec

5.03 μsec

5.13 μsec

~20 μsec
Performance

Latency
- No-op (DPDK): 4.63 μsec
- Unverified NAT (DPDK): 5.03 μsec
- VigNAT (DPDK): 5.13 μsec
- Linux NAT (NetFilter): ~20 μsec

Throughput
- No-op (DPDK): 3.2 Mpps
- Unverified NAT (DPDK): 2.0 Mpps
- VigNAT (DPDK): 1.8 Mpps
- Linux NAT (NetFilter): 0.6 Mpps
### Human Effort (in Lines of Code)

<table>
<thead>
<tr>
<th>VigNAT Code</th>
<th>Proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateless code (data structures)</td>
<td>Symbolic models</td>
</tr>
<tr>
<td>800</td>
<td>1 000</td>
</tr>
<tr>
<td></td>
<td>400 + 325 (unvalidated DPDK)</td>
</tr>
<tr>
<td></td>
<td>23 000</td>
</tr>
</tbody>
</table>

Expect to reuse across many NFs
Verification Friendliness of NF Code

- Low complexity
  - No long/unbounded loops (except main loop)

- Well defined data structures

- Often implements widely adopted standards
Summary

- Vigor = symbolic execution + theorem proving
  - Stitching them is our primary contribution
- VigNAT is formally verified to comply with RFC 3022
  - Competitive performance
  - Tractable verification effort
It is feasible now to build a **stateful** NF that have both **competitive performance** and **formally verified semantic properties** with reasonable effort.
Additional material
Index

- Experimental setup
- DPDK performance report
- Future work
- NAT RFC formalization
- Related work
- Plots
- Stitching details
- Proof Structure
Performance Experiment

- RFC 2544
- Intel Xeon E5-2667 v2 @ 3.30 GHz
- 32 GB of DRAM
- 82599ES 10 Gbps
Performance is comparable

<DPDK perf report>
Future Work

● Certify more NFs
  ○ bridge with mac-learning,
  ○ DMZ

● Improve automation (to reduce the effort and TCB)
  ○ Invariant induction
  ○ Symbolic model selection/generation

● Support Concurrency

● Full system verification (Vigor + CompCert + seL4 + …)
RFC 3022 Formalization

Packet $P$ arrives at time $t$ → $P$ is accepted
  → expire_flows($t$)
  → update_flow($P, t$)
  → forward($P$)

$\text{expire_flows}(t) := \forall G \in \text{flow_table}
  \text{ s.t. } G.\text{timestamp} + T_{exp} \leq t
  \Rightarrow \text{remove } G \text{ from flow_table}$

$update\_flow(P, t) := \text{if } (F(P) \in \text{flow_table}) \text{ then }
  \forall G \in \text{flow_table} \text{ s.t. } F(P) = G
  \Rightarrow \text{set } G.\text{timestamp} = t
  \text{else }
  \text{if } (P.\text{iface} = \text{internal}) \text{ then }
    \text{if } (\text{size(flow_table)} < \text{CAP}) \text{ then }
      \text{insert } F(P) \text{ in flow_table}
    \text{else }
      \text{send packet } S
  \text{else }
    \text{send packet } S
  \text{end if}
$\text{else}$
  \text{drop packet } S
$\text{end if}$
Related work

- **System software verification**
  seL4, CompCert, IronFleet, FSCQ, Beringer et al.

- **Interoperability testing / protocol specification**
  Musuvathi et al., Bishop et al., Kuzniar et al., PIC

- **Network configuration verification / testing**
  SymNet {+NF testing}, BUZZ, Batfish, HSA, VeriFlow, NoD, Anteater, Panda et al., Cocoon, Xie et al.

- **NF software verification** : Dobrescu et al.
Challenges

● Code
  ○ complexity from SymbEx viewpoint
  ○ unbounded number of events
  ○ arbitrary external interactions

● Formalize the RFC in machine readable language

● Integrate symbolic execution and theorem proving
Latency

Number of background flows (thousands)

Latency of probe flows, (μsec)

- No-op
- Unverified NAT
- Verified NAT
Example: NF Code

```c
#define CAP 512
int main() {
    struct packet p;
    struct ring *r = ring_create(CAP);
    if (!r) return 1;
    while(VIGOR_LOOP(1)) {
        loop_iteration_begin(&r);
        if (!ring_full(r))
            if (receive(&p) && p.port != 9)
                ring_push_back(r, &p);
        if (!ring_empty(r) && can_send()) {
            ring_pop_front(r, &p);
            send(&p);
        }
        loop_iteration_end(&r);
    }
    return 0;
}
```
#define CAP 512
int main() {
    struct packet p;
    struct ring *r = ring_create(CAP);
    if (!r) return 1;
    while (VIGOR_LOOP(1))
    {
        loop_iteration_begin(&r);
        if (!ring_full(r))
            if (receive(&p) && p.port != 9)
                ring_push_back(r, &p);
        if (!ring_empty(r) && can_send())
            if (!ring_pop_front(r, &p))
                send(&p);
        loop_iteration_end(&r);
    }
    return 0;
}
SymbEx→

Theorem Proving:

Replay

loop_iteration_begin(&X) => []
ring_full(&X) => true
ring_empty(&X) => false
can_send() => true
ring_pop_front(&X, &{.port == y} -> &{.port == z}) => []

——
z != 9

SymbEx→

Theorem Proving:

Replay

struct ring* arg1;
struct packet arg2;

loop_invariant_produce(&((arg1)));
//@ open loop_invariant(_);

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));
//@ open packetp(&(arg2), _);

//@ assert(arg2.port != 9);

index
SymbEx→
Theorem Proving:
Replay

```c
struct ring* arg1;
struct packet arg2;

loop_invariant_produce(&((arg1)));
//@ open loop_invariant(_);
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)));
```

Contract: ... and packet satisfies packet_constraints_fp.
Proof Structure

- VigNAT satisfies semantic properties
- VigNAT satisfies low-level properties
  - Stateful implementations behaves according to the interface contracts
  - VigNAT stateless code respects the interface contracts
  - Symbolic models are faithful to the interface contracts

index