VigNAT: A Formally Verified NAT

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

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

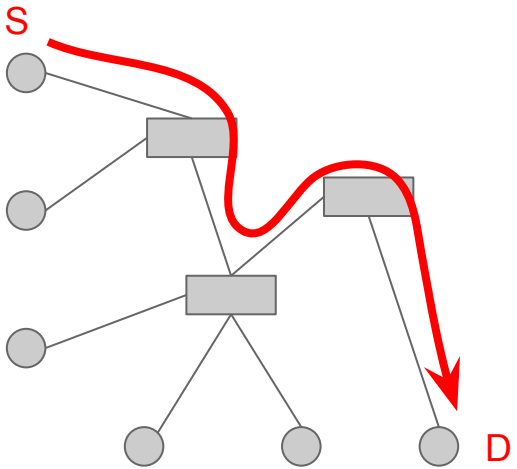
Testing: Easy but Incomplete

Testing: Easy but Incomplete

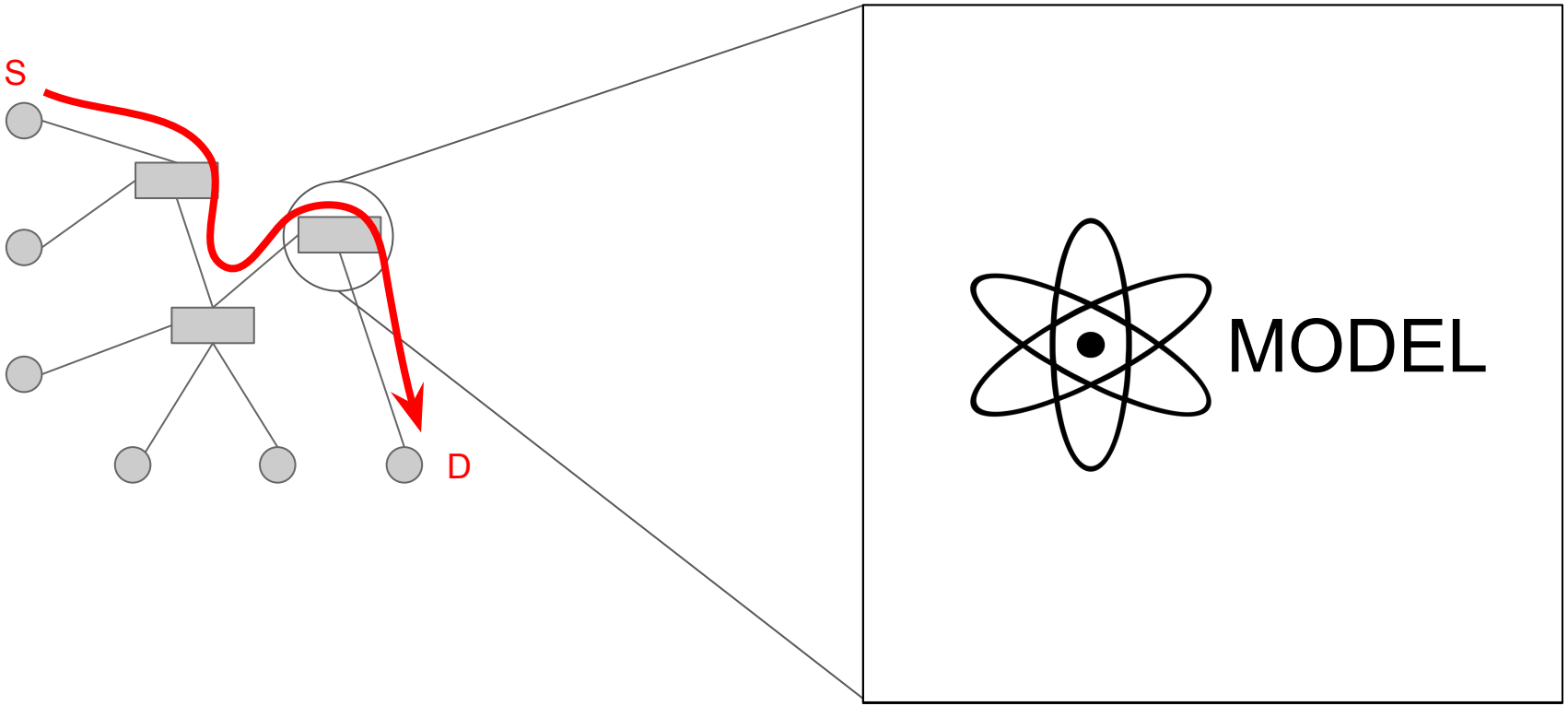
Formal Verification: Complete but Expensive

Formal Verification: Complete ~~but Expensive~~?

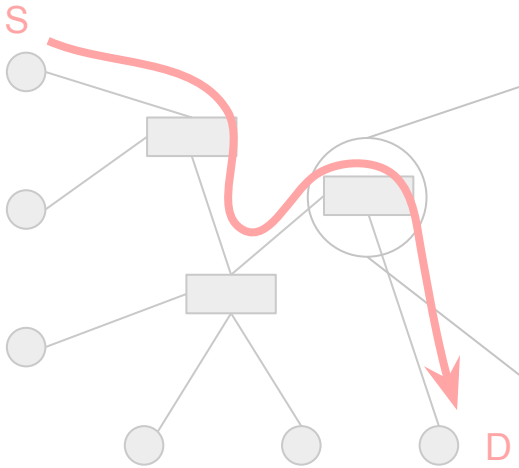
Network Verification



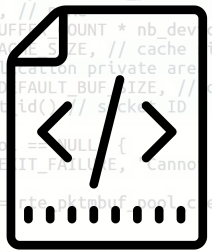
Network Verification



Network Verification ≠ NF **Code** Verification




```
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
);
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);
  }
}
```



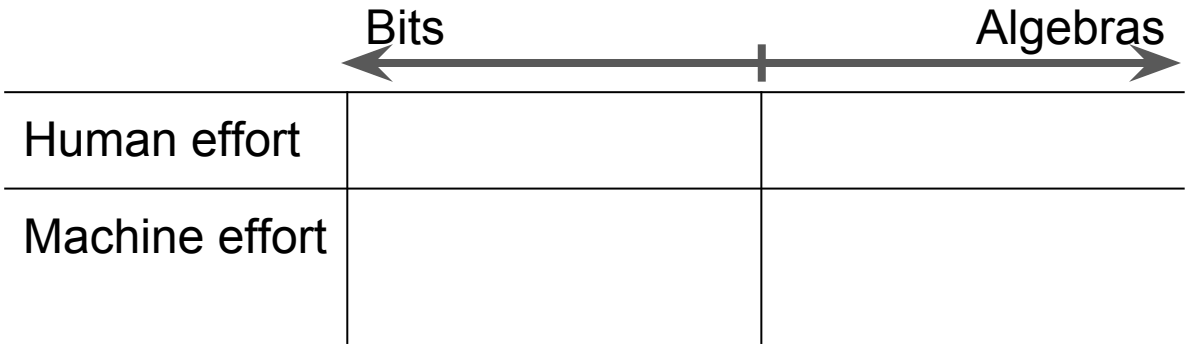
CODE

How to Verify an NF (Before Vigor)?

How to Verify an NF (Before Vigor)?

How to Verify an NF (Before Vigor)?

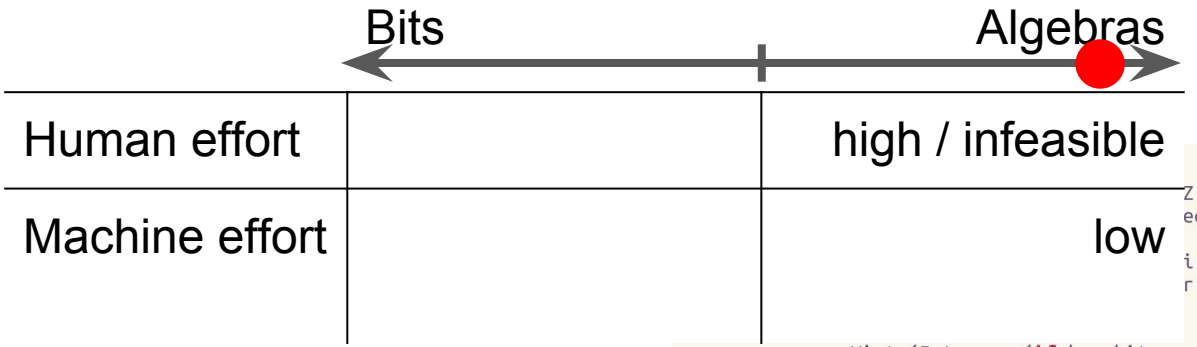
How to Verify an NF (Before Vigor)?



Theorem Proving

Human effort		high / infeasible
Machine effort		low

Theorem Proving



Too complicated

```
Z.eq_dec i z then val else values m i;
eq_dec i z then key else keys m i;

i z then true else busybits m i |}).
r (if busybits m x then 1 else 0)))
```

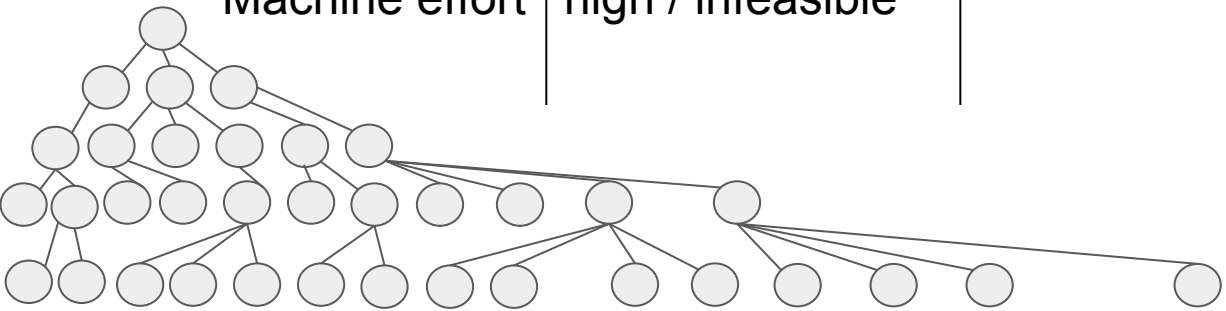
```
            Vint (Int.repr (if busybits ret x then 1 else 0)))) as BBEQ. {
  unfold upd;unfold ret;apply functional_extensionality;intro;simpl.
  unfold initial_world.EqDec_Z, zeq.
  destruct (Z.eq_dec z x), (Z.eq_dec x z);to_abstract;tauto.
}
assert ((upd (fun x : Z => Vint (Int.repr (keys m x))) z
             (Vint (Int.repr key))) =
        (fun x : Z => Vint (Int.repr (keys ret x)))) as KEYSEQ. {
  unfold upd;unfold ret;apply functional_extensionality;intro;simpl.
  unfold initial_world.EqDec_Z, zeq.
  destruct (Z.eq_dec z x), (Z.eq_dec x z);to_abstract;tauto.
}
assert ((upd (fun x : Z => Vint (Int.repr (values m x))) z
             (Vint (Int.repr val))) =
        (fun x : Z => Vint (Int.repr (values ret x)))) as VALSEQ. {
  unfold upd;unfold ret;apply functional_extensionality;intro;simpl.
  unfold initial_world.EqDec_Z, zeq.
  destruct (Z.eq_dec z x), (Z.eq_dec x z);to_abstract;tauto.
}
unfold amPut.
rewrite FE.
```


[1] Klein, Gerwin, et al. "seL4: Formal verification of an OS kernel." *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.

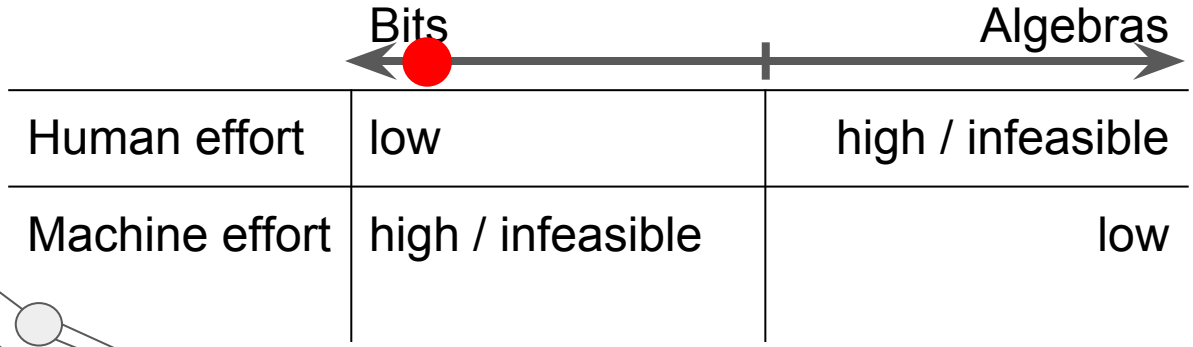
[2] Chen, Haogang, et al. "Using Crash Hoare logic for certifying the FSCQ file system." *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015.

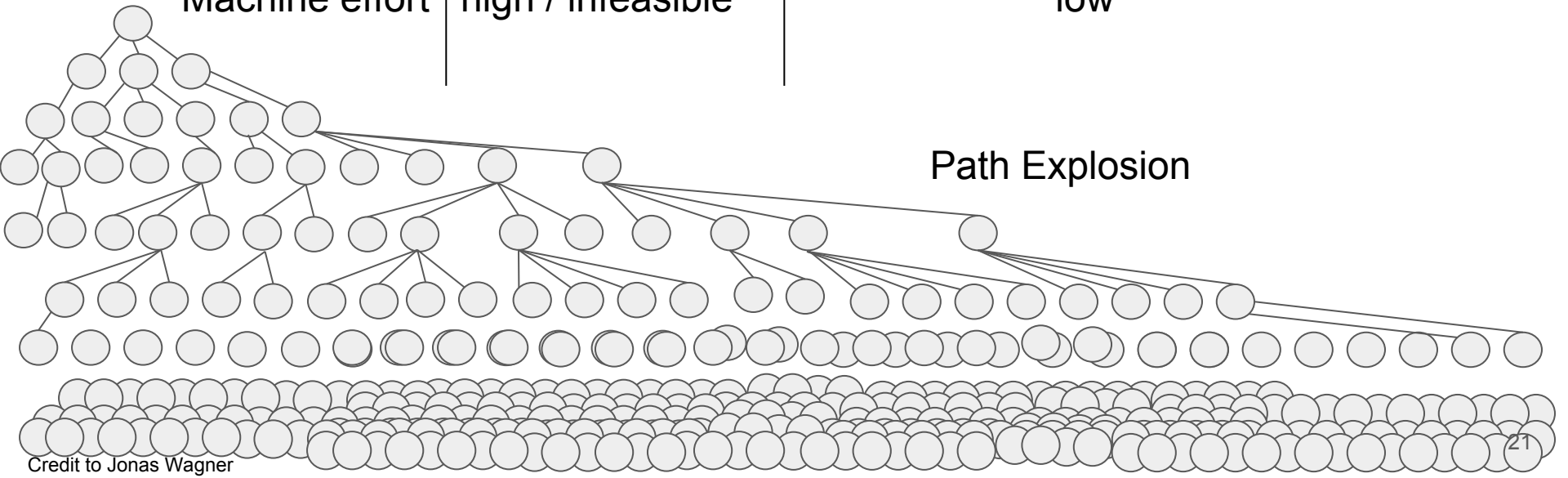
Exhaustive Symbolic Execution (SymbEx)

Human effort	low	high / infeasible
Machine effort	high / infeasible	low



Exhaustive Symbolic Execution (SymbEx)





Credit to Jonas Wagner

Vigor

Human effort	**low**	high / infeasible
Machine effort	high / infeasible	**low**

Vigor

Human effort	**low**	high / infeasible
Machine effort	high / infeasible	**low**

Plus runtime performance

Main Idea

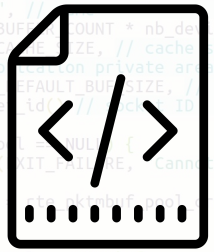
- Split the code into two parts
- Verify each part separately
- Stitch the proofs — key challenge

Outline

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

Vigor: split code | verify parts | stitch proofs

```
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
);
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);
  }
}
```



CODE

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

```
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
);
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

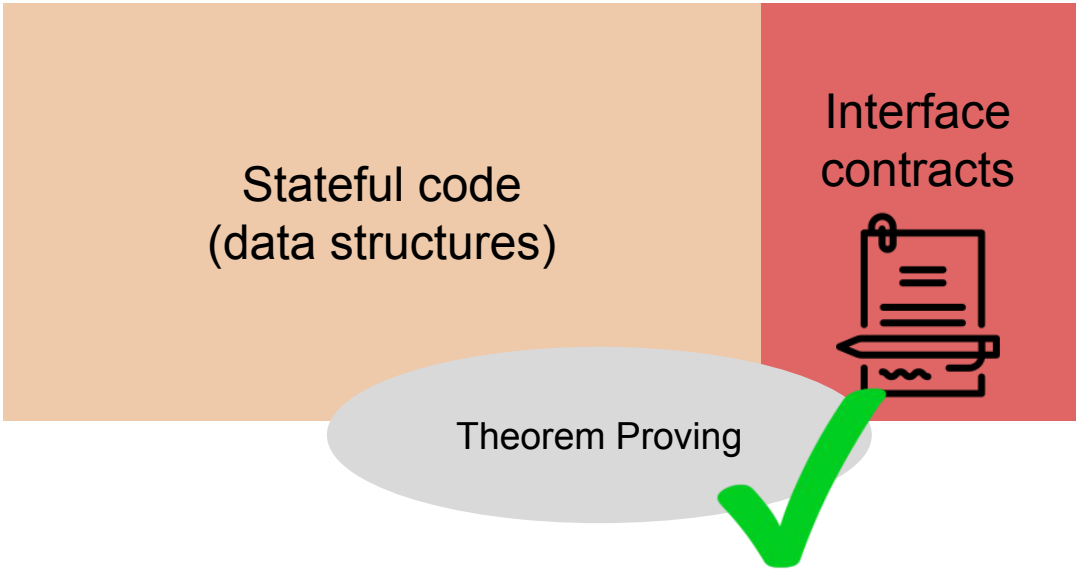
Vigor: **split code** | verify parts | stitch proofs

Stateful code
(data structures)

Interface
contracts

Stateless code
(application logic)

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



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

Stateful code
(data structures)

Interface
contracts

Stateless code
(application logic)

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

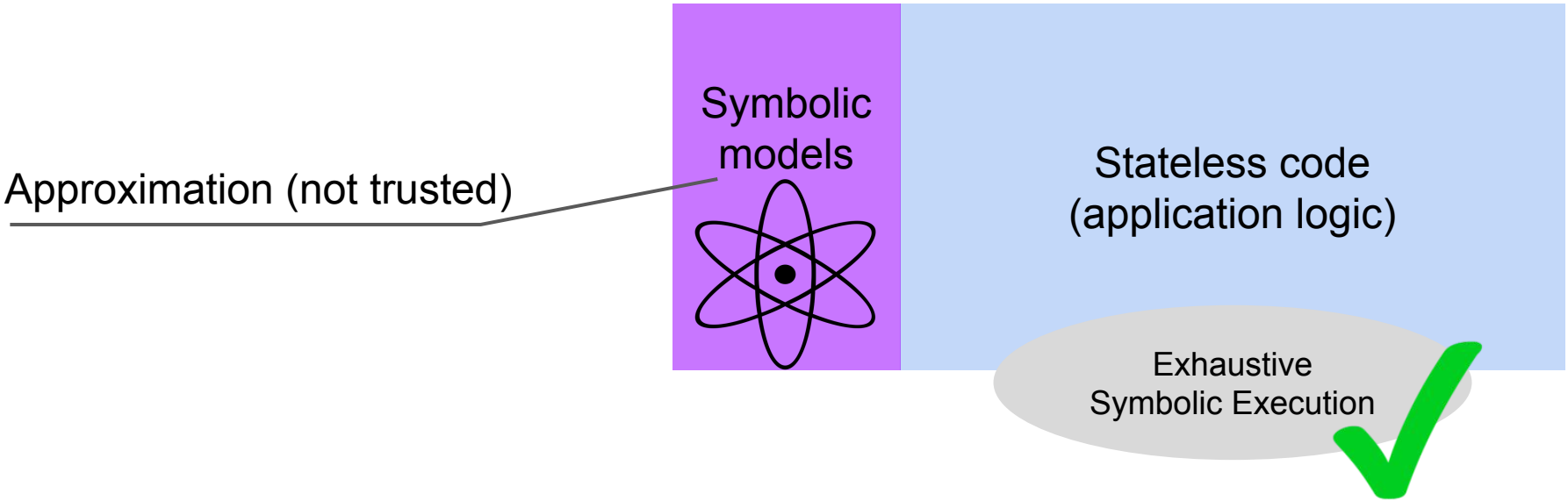
Stateful code
(data structures)

Interface
contracts

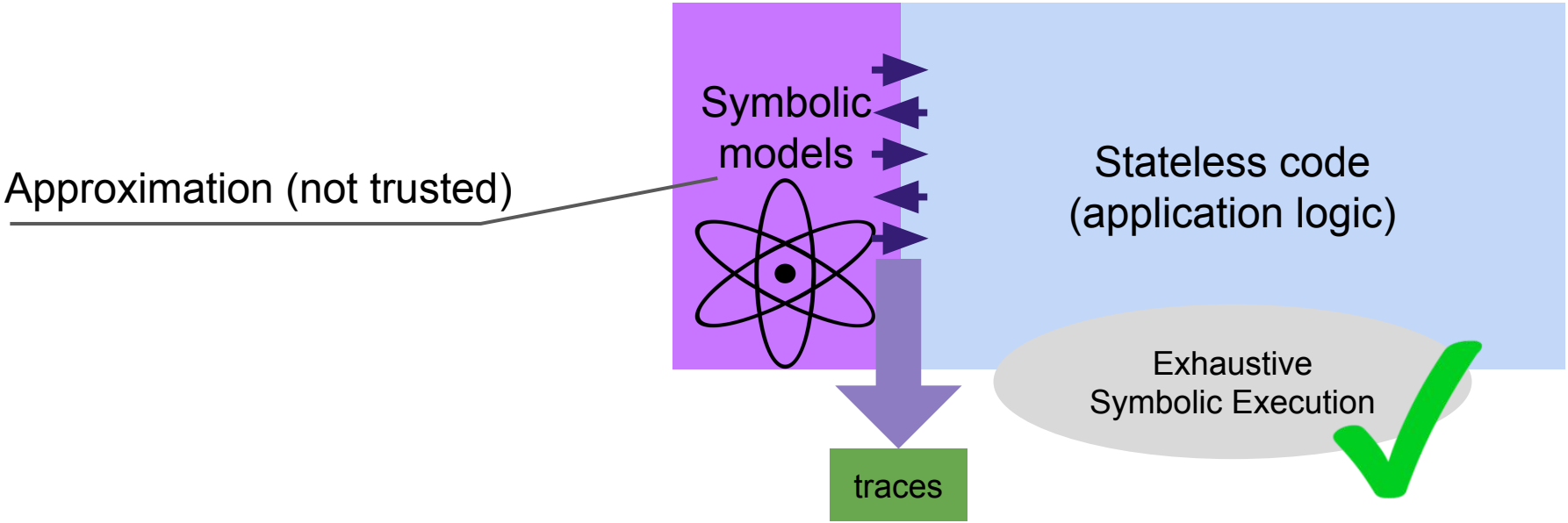
Stateless code
(application logic)

Exhaustive
Symbolic Execution

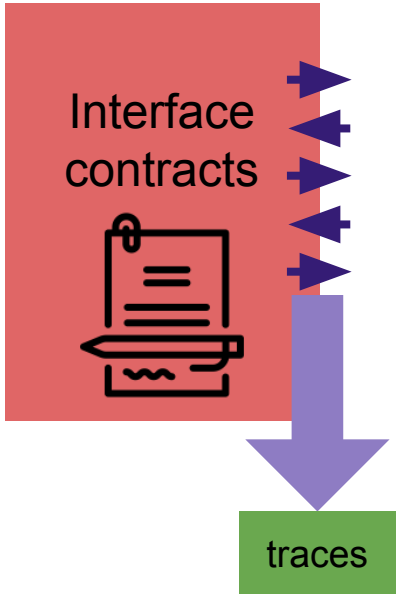
Vigor: split code | **verify parts** | stitch proofs



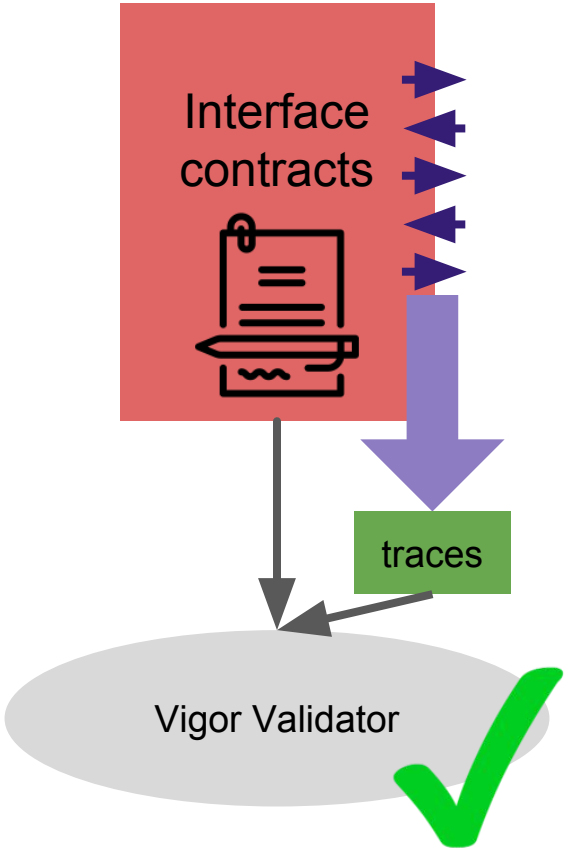
Vigor: split code | **verify parts** | stitch proofs



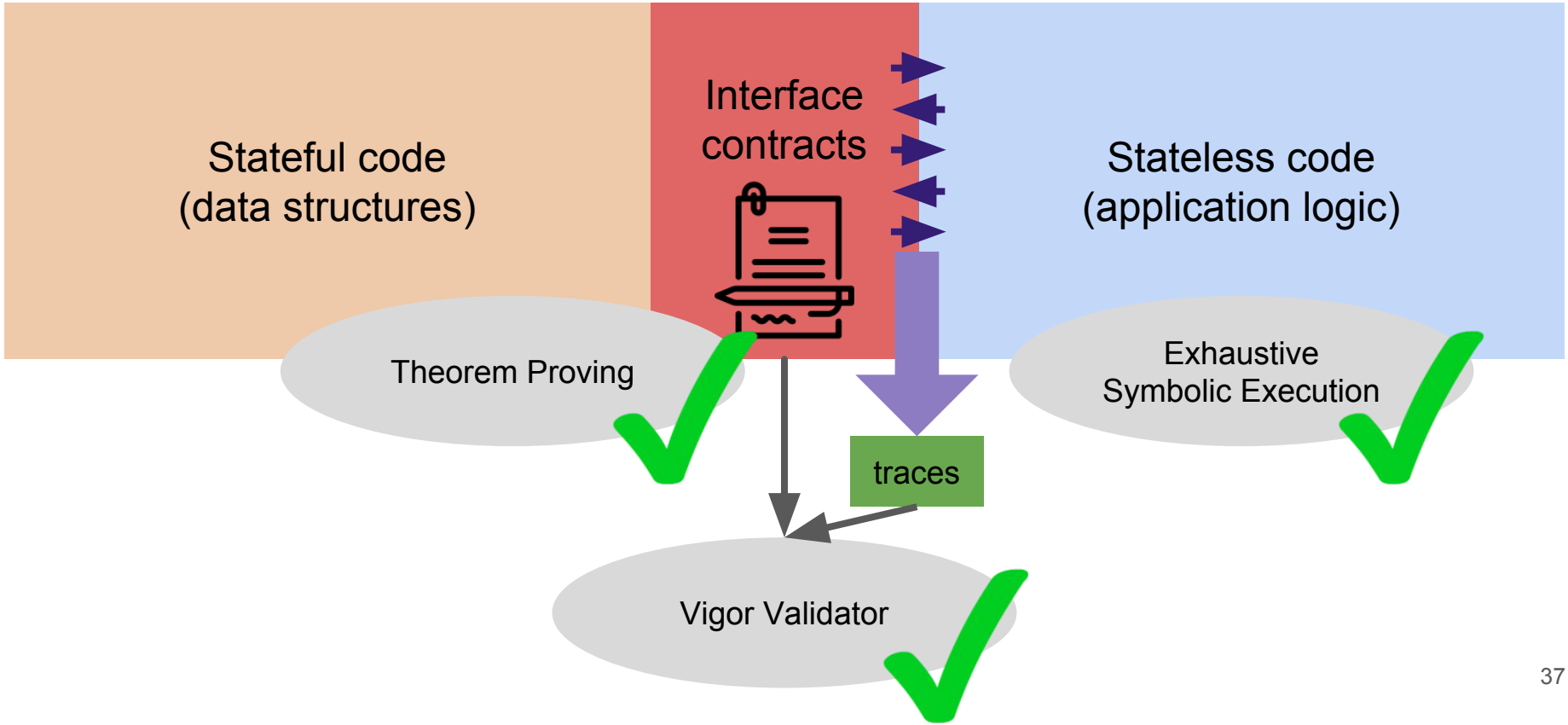
Vigor: split code | verify parts | **stitch proofs**



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



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



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

```
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);
}
```

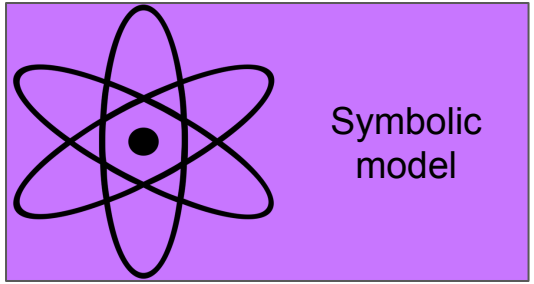
Example NF Code

```
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

```
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

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

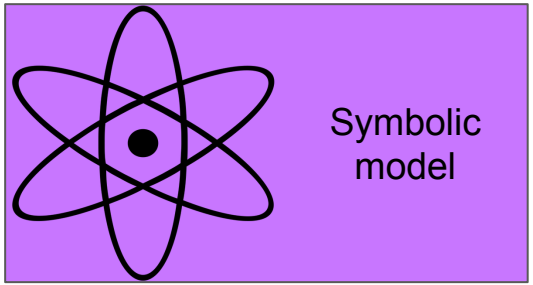
r is not empty and
p points to valid memory

---

r contains one packet less and
p points to a packet and p->port ≠ 9

Example: Symbolic Model

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

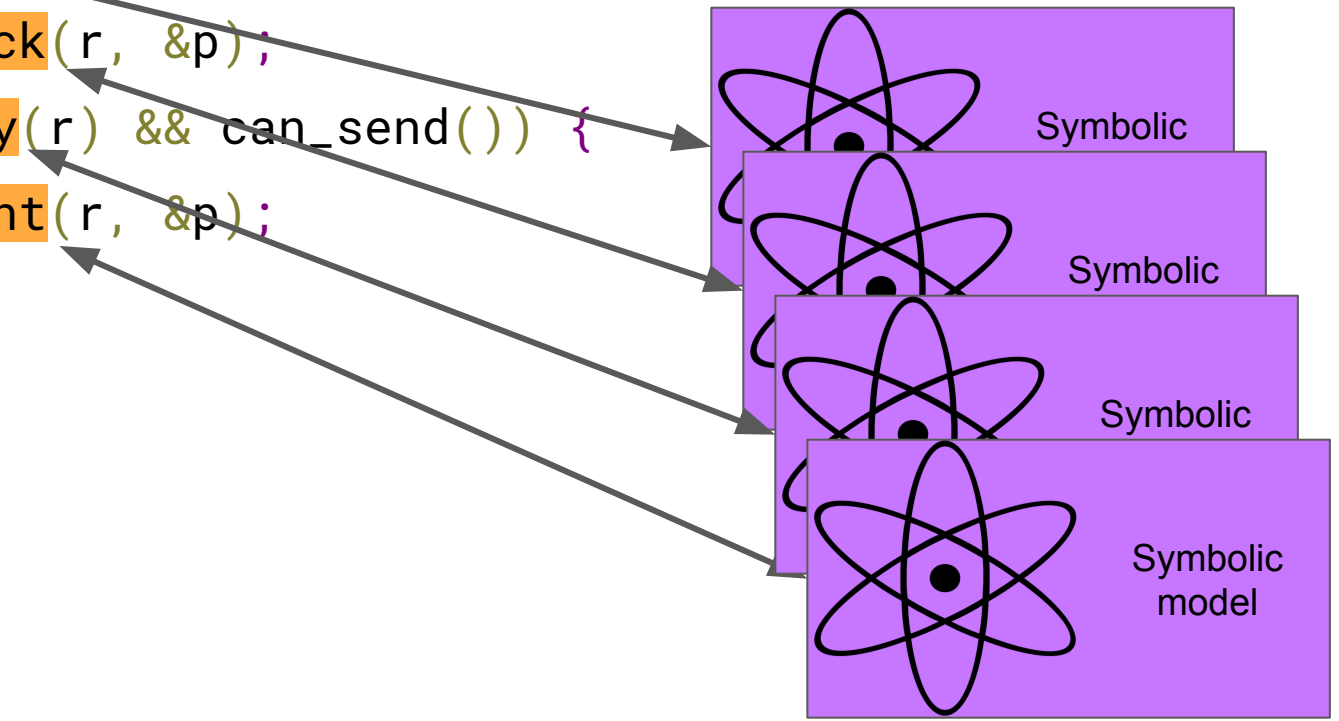


Example NF Code

```
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

```
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

```
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

```
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

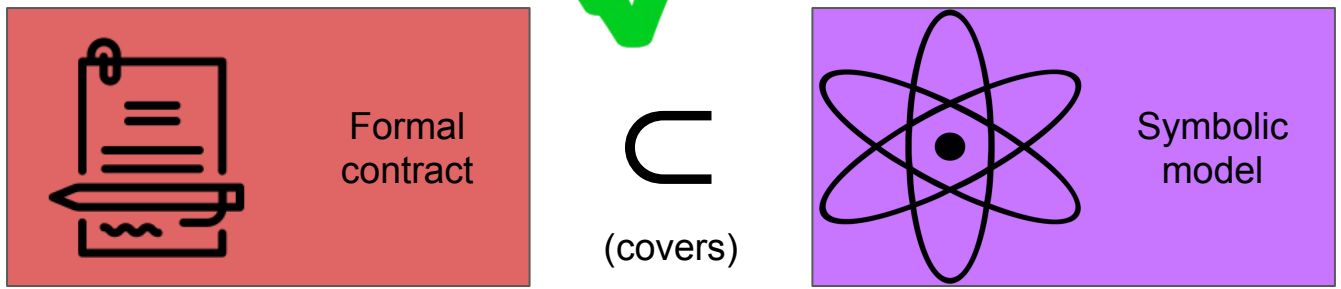
```
r1 = ring_full(r); assume(r1 == true);

r2 = ring_empty(r); assume(r2 == false);
ring_pop_front(r, &p);
assert(p.port ≠ 9);
```

Over-Approximation Proof

```
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

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

r2 = ring_empty(r); assume(r2 == false);
ring_pop_front(r, &p);
assert(p.port ≠ 9);
```



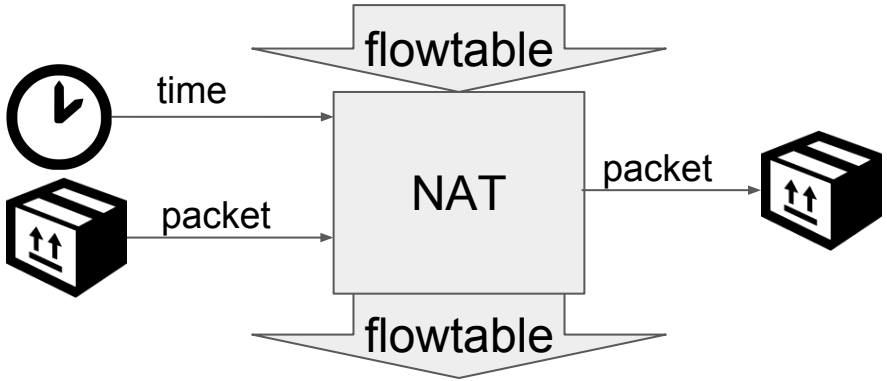
Outline

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

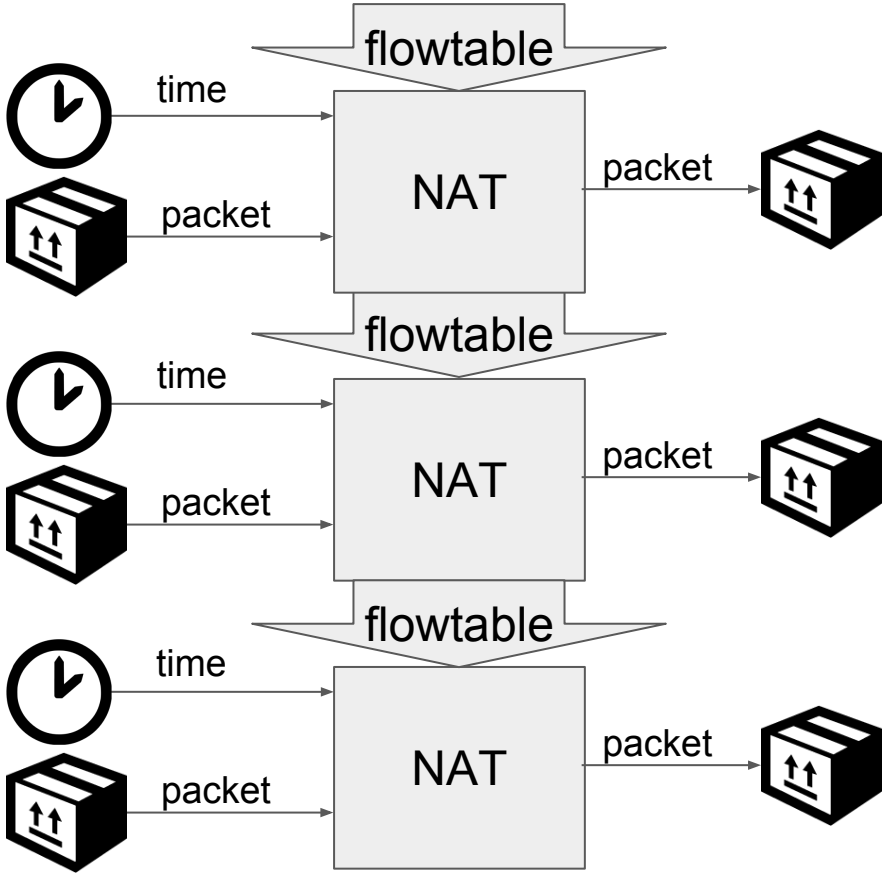
Formalization of the NAT RFC

- 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

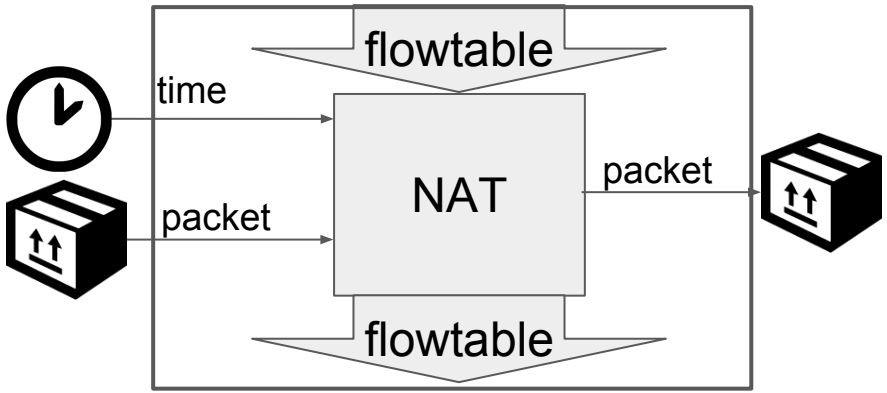


Formalization of the NAT RFC

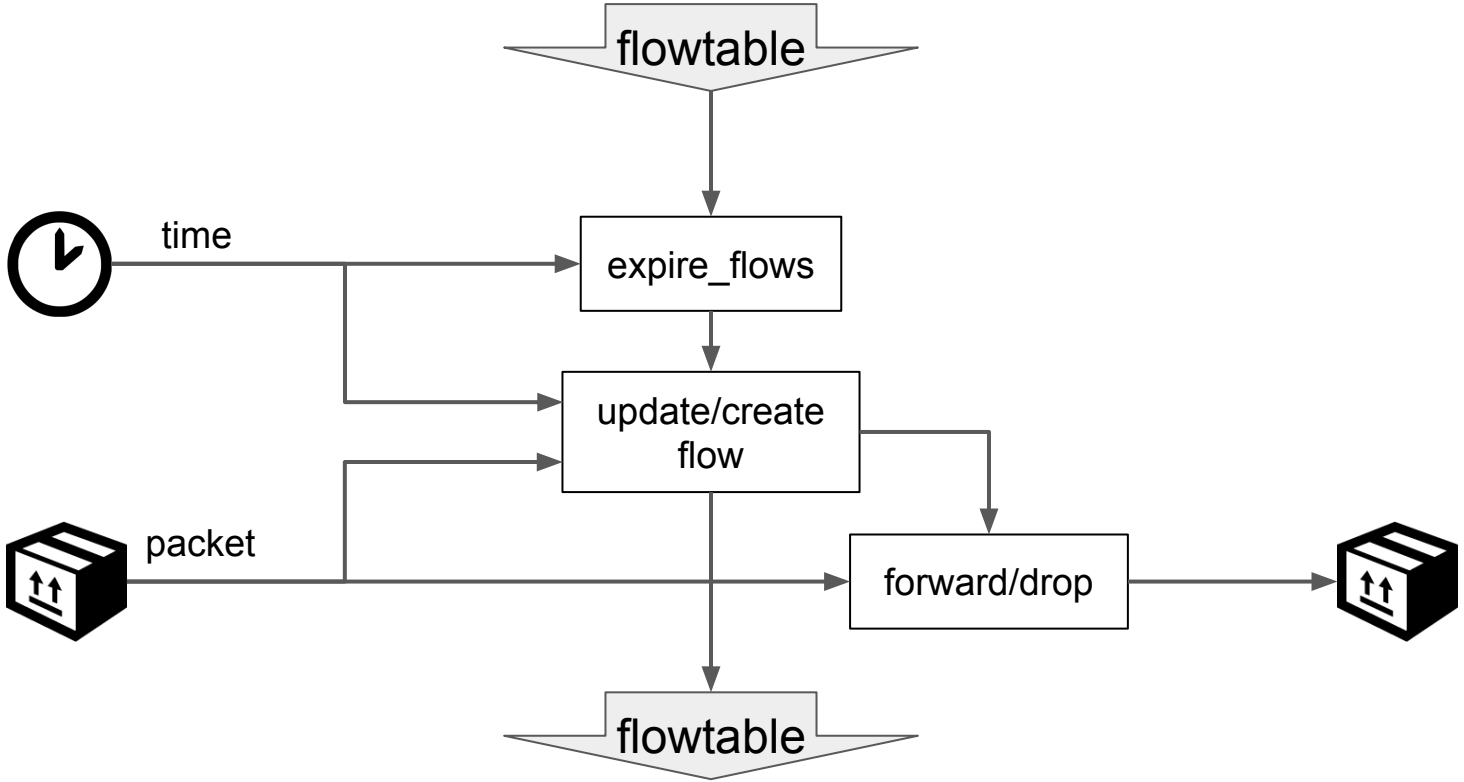


Formalization of the NAT RFC

$$\{\text{flowtable}_{\text{before}}, \text{time}, \text{packet}_{\text{in}}\} \rightarrow \{\text{flowtable}_{\text{after}}, \text{packet}_{\text{out}}\}$$



Formalization of the NAT RFC



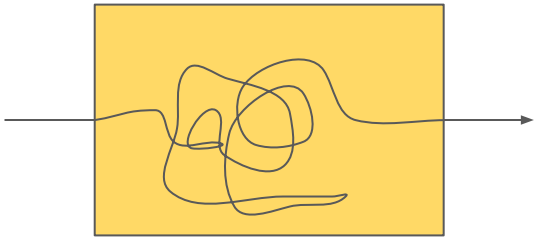
Outline

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

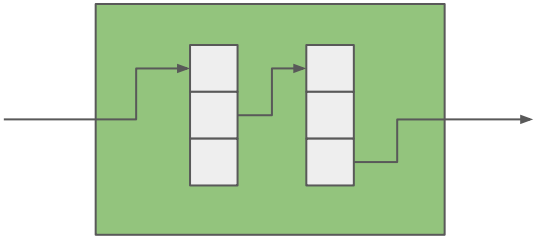
Performance

No-op
(DPDK)

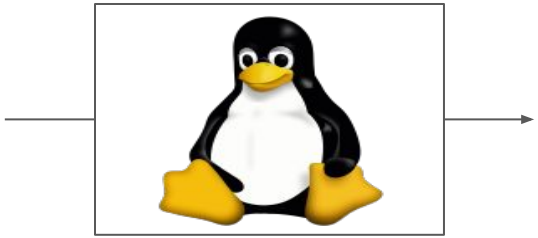
Unverified NAT
(DPDK)



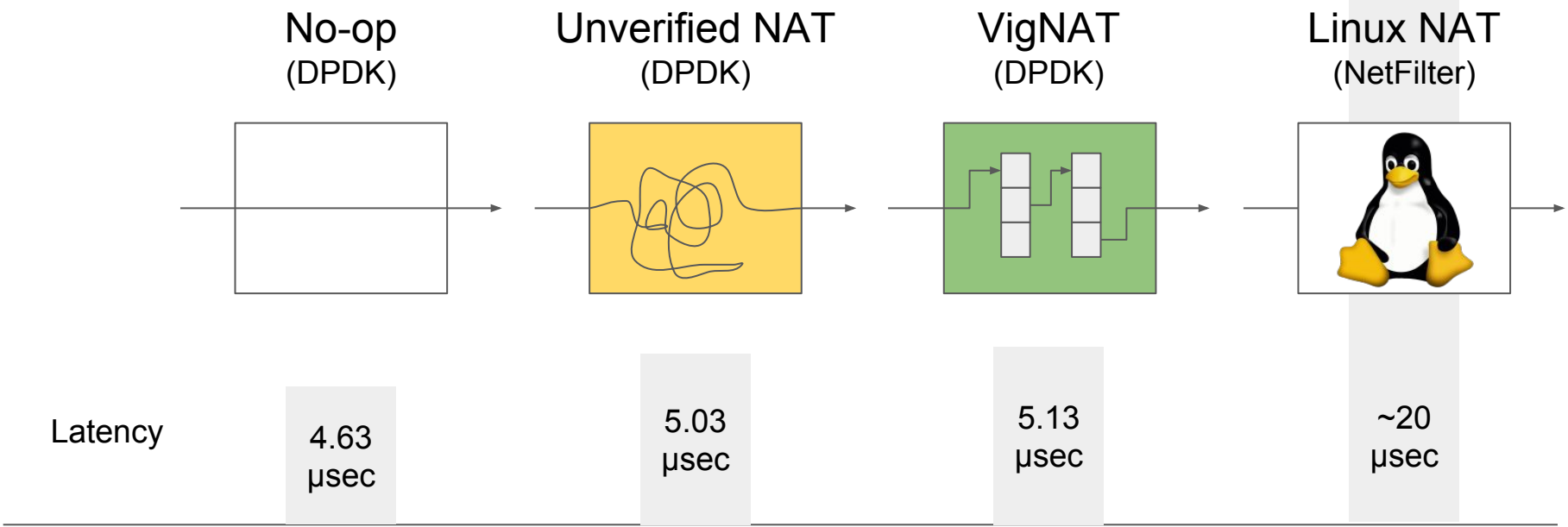
VigNAT
(DPDK)



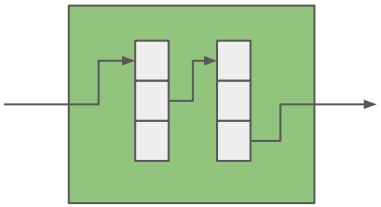
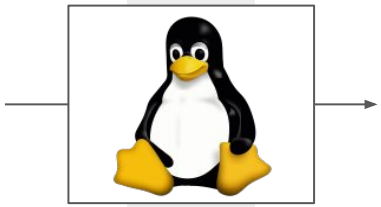
Linux NAT
(NetFilter)



Performance



Performance

	No-op (DPDK)	Unverified NAT (DPDK)	VigNAT (DPDK)	Linux NAT (NetFilter)
				
Latency	4.63 µsec	5.03 µsec	5.13 µsec	~20 µsec
Throughput	3.2 Mpps	2.0 Mpps	1.8 Mpps	0.6 Mpps

Human Effort (in Lines of Code)

VigNAT Code		Proof	
Stateless code	Stateful code (data structures)	Symbolic models	Proofs of data structure library
800	1 000	400 + 325 (unvalidated DPDK)	23 000

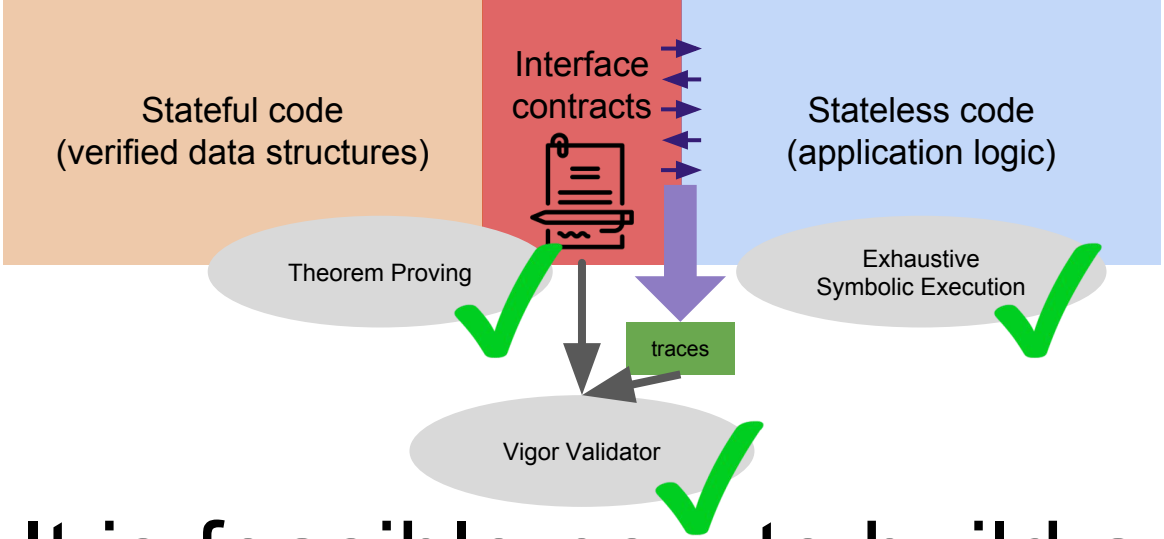
Expect to reuse across many NFs

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



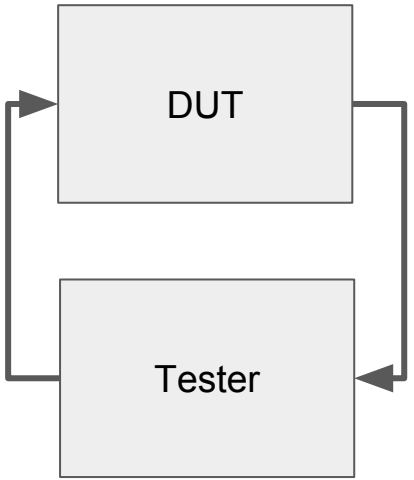
It is feasible now to build a **stateful** NF that have both **competitive performance** and formally verified **semantic properties** with reasonable effort

Additional material

Index

Performance Experiment

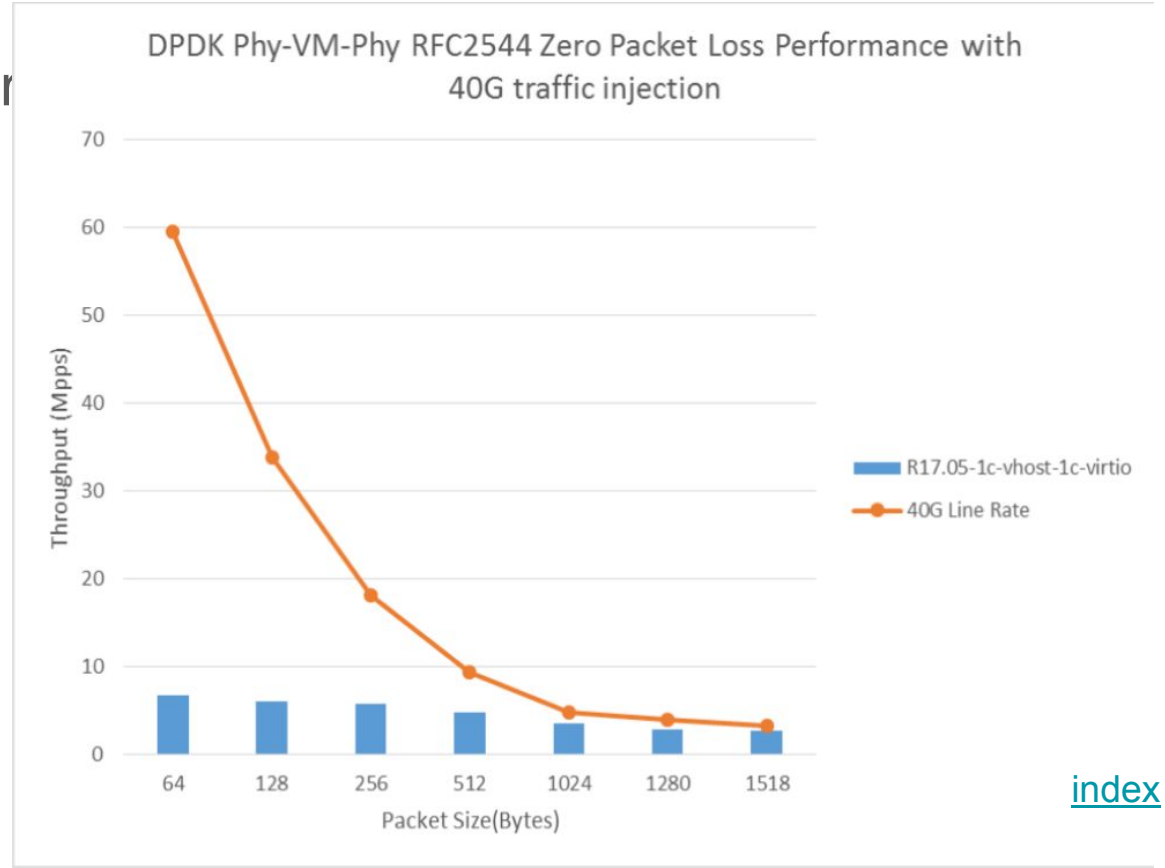
- RFC 2544
- Intel Xeon E5-2667 v2 @ 3.30 GHz
- 32 GB of DRAM
- 82599ES 10 Gbps



index

Performance is comparable

<DPDK perf r



index

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 + …)

index

RFC 3022 Formalization

```
1  Packet P arrives at time t → P is accepted
2                             → expire_flows(t)
3                             → update_flow(P, t)
4                             → forward(P)
5
6  expire_flows(t) :=  ∀G ∈ 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) ∈ flow_table) {
11                         ∀G ∈ 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) ∈ flow_table) {
21                     if (P.iface = internal) {
22                         → S.data = P.data
23                         → S.iface = external
24                         → S.dst_ip = P.dst_ip
25                         → S.dst_port = P.dst_port
26                         → S.src_ip = EXT_IP
27                         → S.src_port = F(P).ext_port
28                         → send packet S
29                     } else {
30                         → S.data = P.data
31                         → S.iface = internal
32                         → S.dst_ip = F(P).int_ip
33                         → S.dst_port = F(P).int_port
34                         → S.src_ip = P.src_ip
35                         → S.src_port = P.src_port
36                         → send packet S
37                     }
38                 } else {
39                     drop packet
40                 }
```

index

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.

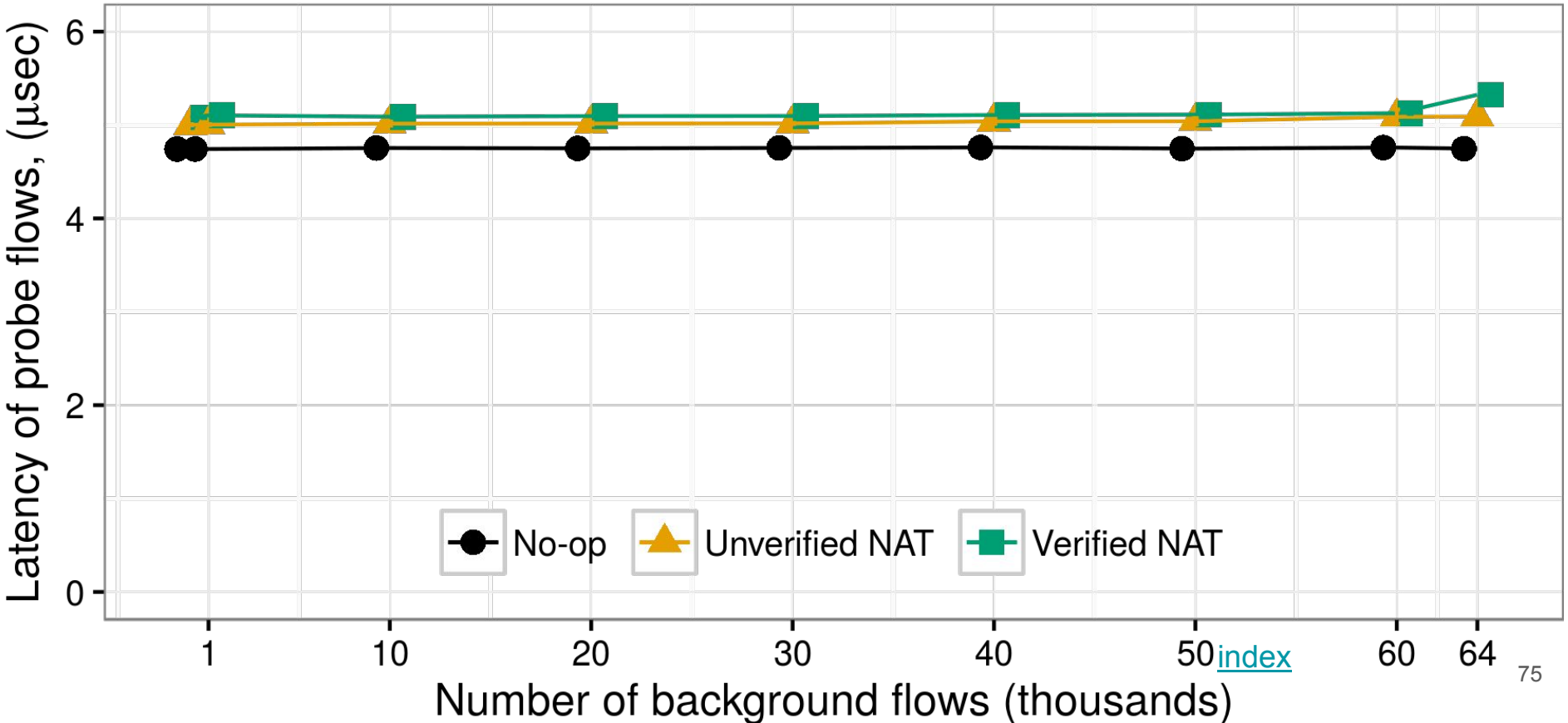
index

Challenges

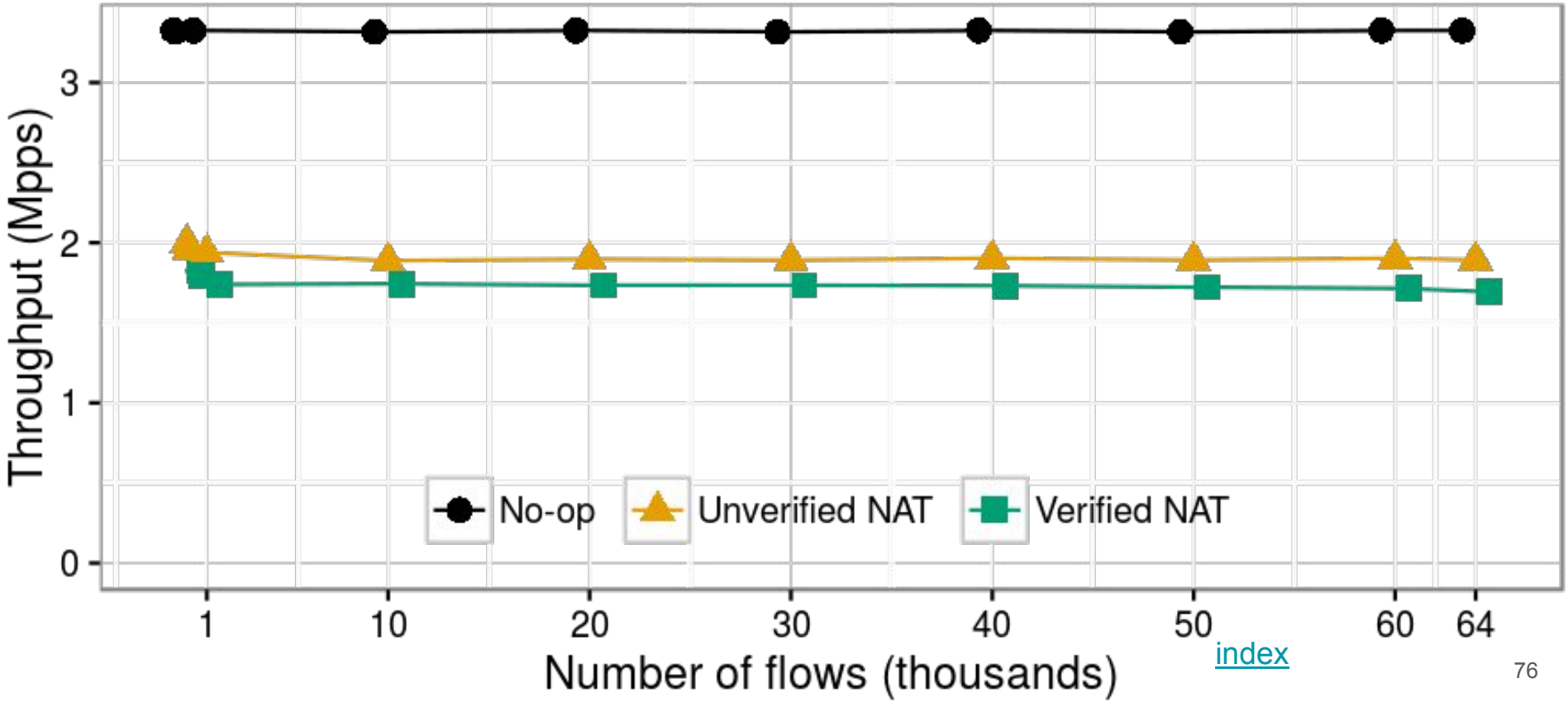
- Code
  - complexity from SymbEx viewpoint
  - unbounded number of events
  - arbitrary external interactions
- Formalize the RFC in machine readable language
- Integrate symbolic execution and theorem proving

index

Latency


Latency of probe flows, (µsec)
6
4
2
0
No-op
Unverified NAT
Verified NAT
1
10
20
30
40
50
index
60
64
Number of background flows (thousands)

Throughput



index

Example: NF Code

```
#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;
}
```

index

SymbEx→ Theorem Proving: Trace

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

```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;
}
```

index

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
```

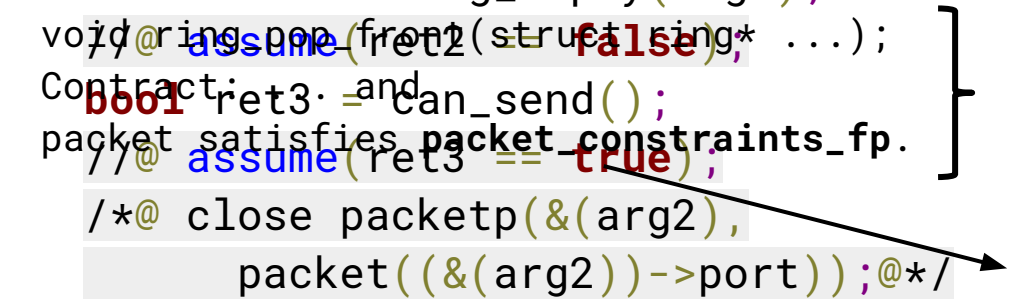
```
struct ring* arg1;
struct packet arg2;

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

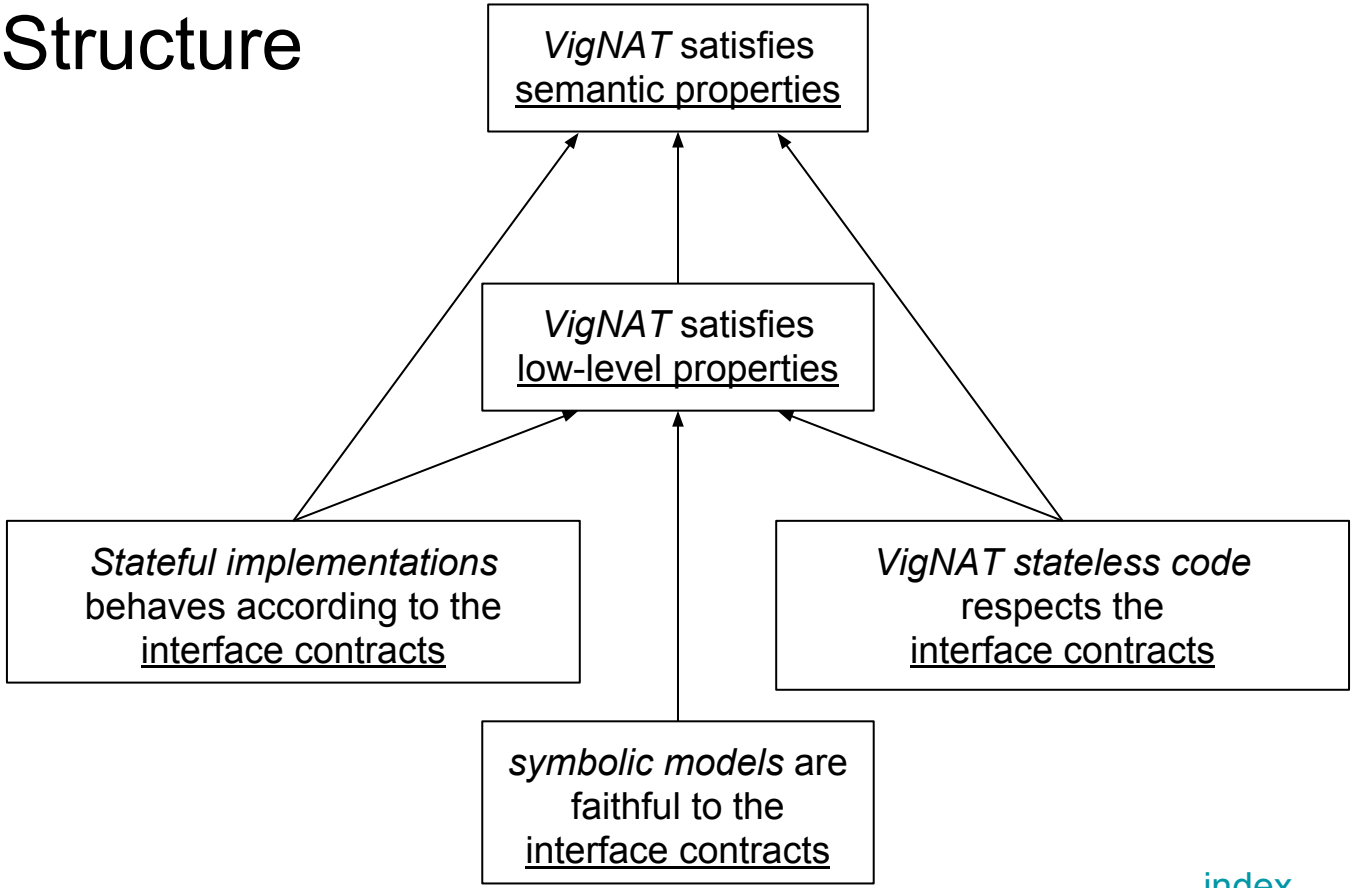
```
struct ring* arg1;
struct packet arg2;

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));
```

void ring_pop_front(struct ring* ...);
Contract: ... and
packet satisfies **packet_constraints_fp**.



index

Proof Structure



index