Verification misc

Agenda

Papers

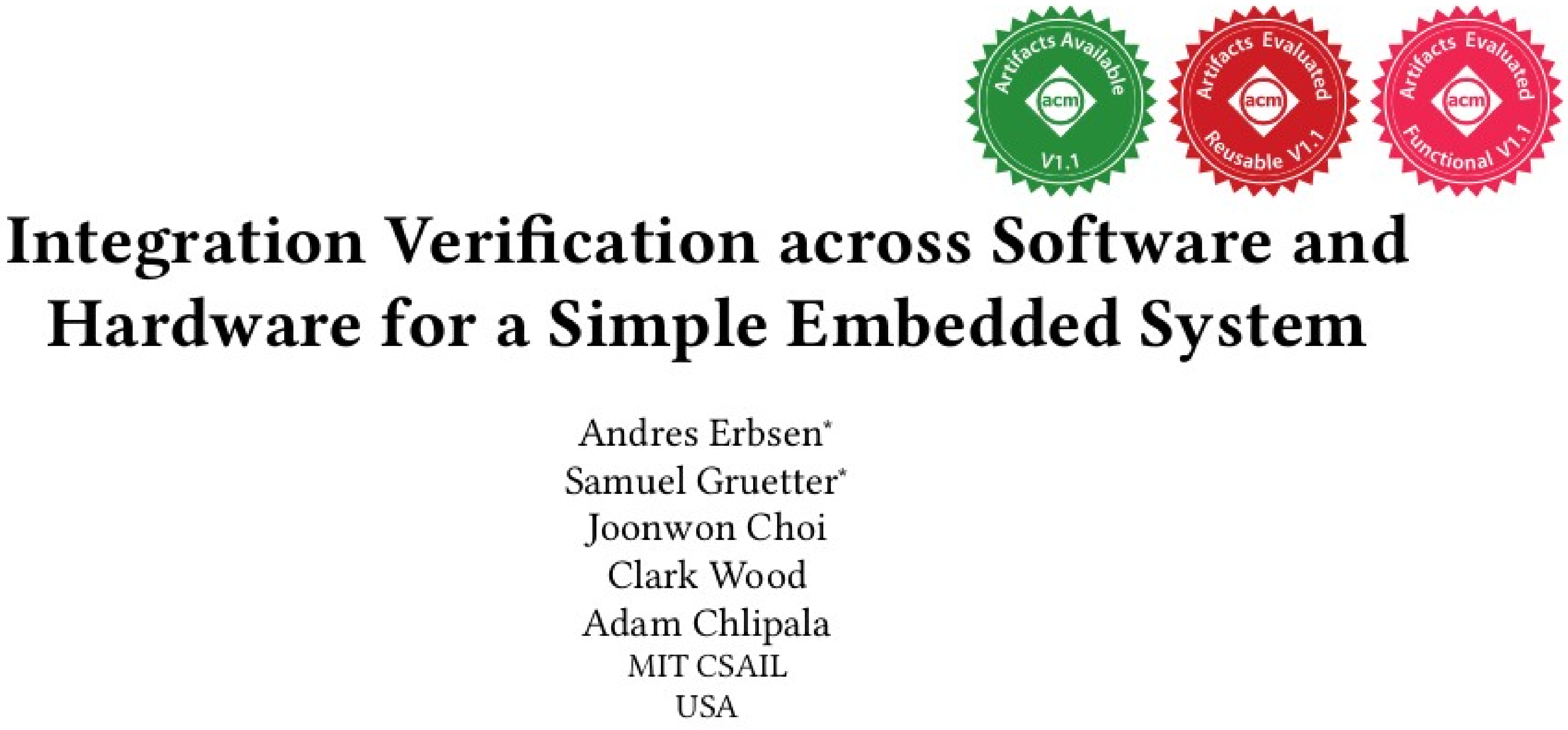
- Integration verification (PLDI’21)
- Network functions (SOSP’19)

Projects

- Rust verification

Integration verification (PLDI'21)

Integration Verification: PLDI’21



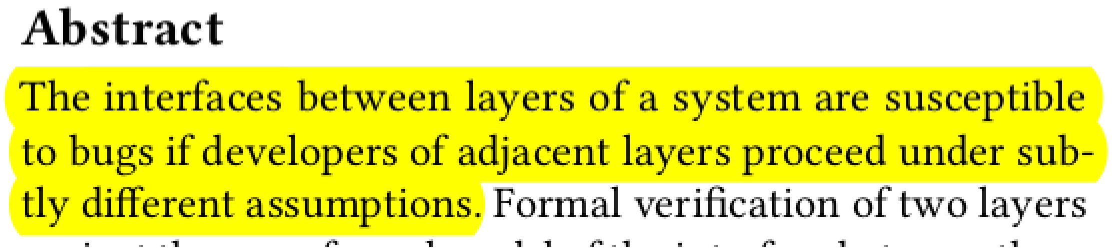
Integration Verification across Software and Hardware for a Simple Embedded System

Andres Erbsen*
Samuel Gruetter*
Joonwon Choi
Clark Wood
Adam Chlipala
MIT CSAIL
USA

Bedrock Problem

Integration verification

Whole system, spanning hardware → os + app → compiler



Abstract

The interfaces between layers of a system are susceptible to bugs if developers of adjacent layers proceed under subtly different assumptions. Formal verification of two layers

e.g. arithmetic overflow: wrap or trap?

Contributions

Integration verification on *realistic ISA* (RISC-V)

Modeling of MMIO

Bedrock overview

Target system

- Embedded RISC-V
  - Single-threaded
  - App does not need OS
- Custom processor
- Custom C-like language
  - Verified compiler
- The App
  - Light bulb controller via MMIO

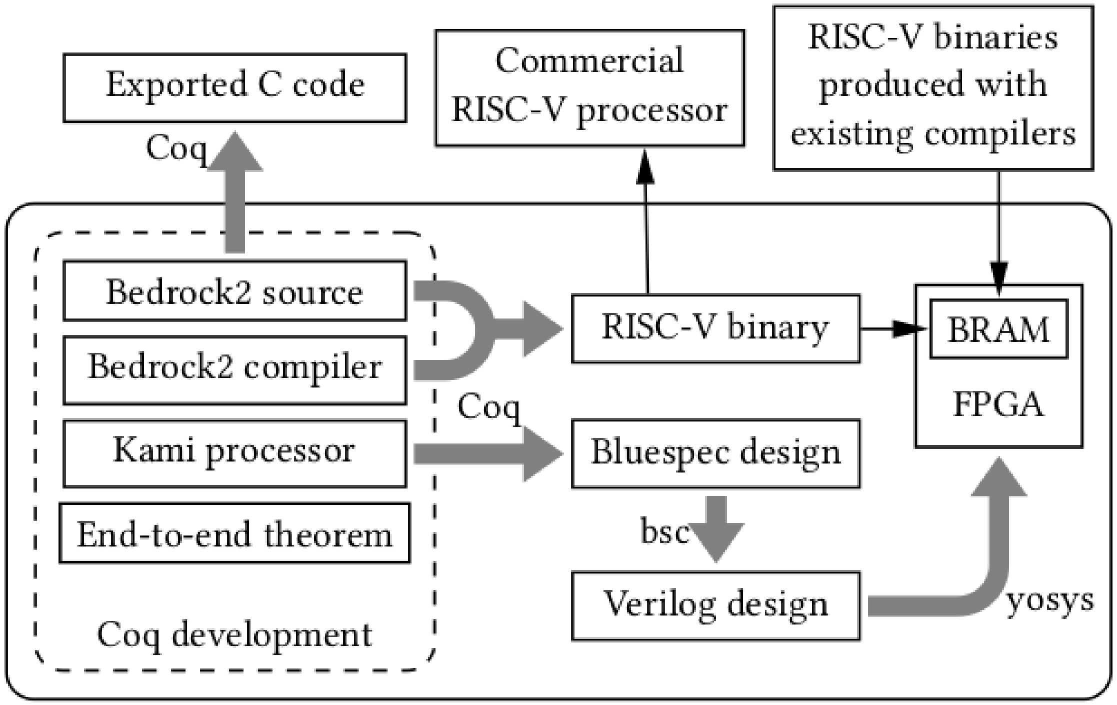


Figure 1. System overview. The top row highlights compat-ibility with existing interfaces and tools.

Bedrock overview

Target system

- Embedded **RISC-V**
  - Single-threaded
  - App does not need OS
- Custom **processor**
- Custom C-like **language**
  - Verified **compiler**
- The **App**
  - Light bulb controller via MMIO

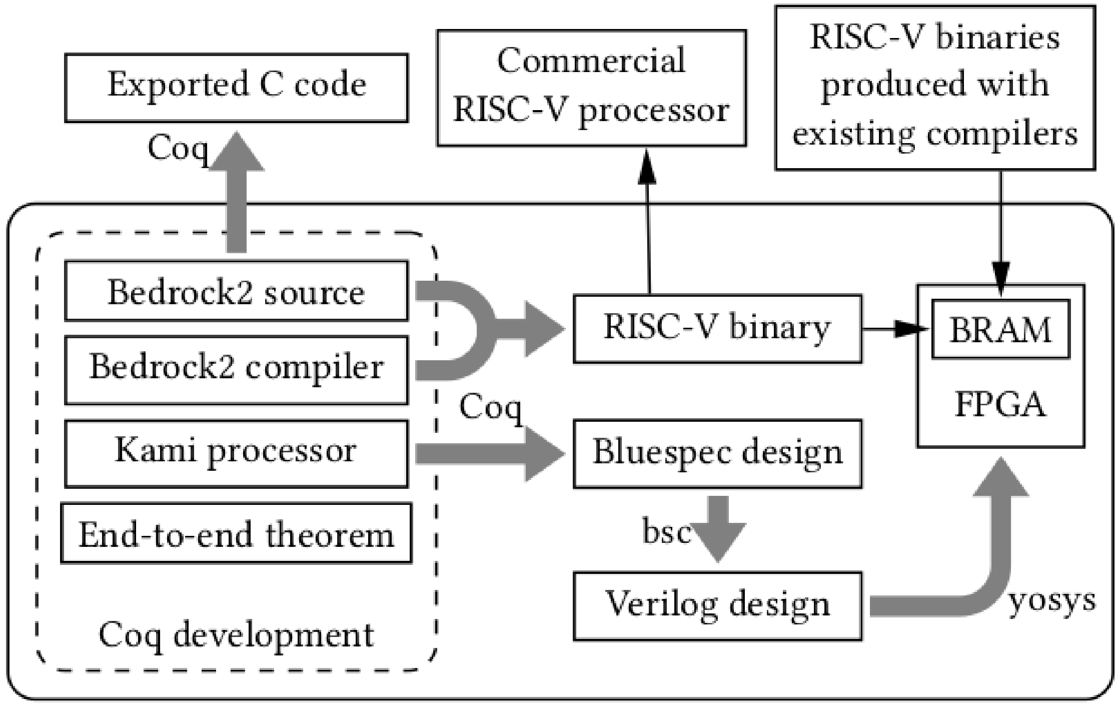


Figure 1. System overview. The top row highlights compatibility with existing interfaces and tools.

Bedrock specification

Topmost specification

Behavior of the whole system (from app to processor)

Specifies the legal I/O sequences in a regex fashion

```
Definition goodHlTrace :=
 BootSeq +++ ((EX b: bool, Recv b +++ LightbulbCmd b)
              ||| RecvInvalid ||| PollNone) ^*.
```

→ After boot, poll the I/O.

→ Each poll either 1. returns none; 2. gets an invalid value; 3. gets a valid value which is then sent to the light bulb via I/O

Bedrock specification

RISC-V specification

Written against the RISC-V standard [7]

→ To an executive haskell spec

→ Then converted into coq

Defines RISC-V system state & semantics of instructions

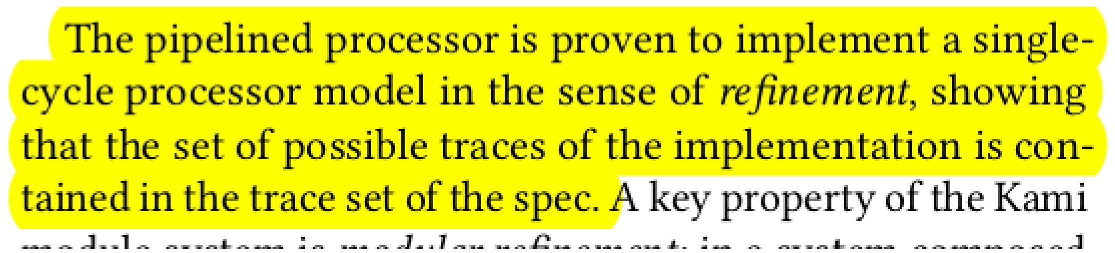
[7] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Andrew Wright, and Adam Chlipala. 2021. A Multipurpose Formal RISC-V Specification. arXiv:2104.00762 [cs.LO]

Bedrock specification

(The Kami) Processor specification

A 4 stage pipelined processor

Proven against: a multicycle (i.e. single stage) processor



The pipelined processor is proven to implement a single-cycle processor model in the sense of *refinement*, showing that the set of possible traces of the implementation is contained in the trace set of the spec. A key property of the Kami

Problem: matching hardware spec with software spec

Bedrock specification

The compiler

Pretty standard techniques as in *CompCert*, a well-established verified C compiler.

Spec: forward simulation between the source and compiled code

Bedrock specification

Interfacing hardware and software

Two models of RISC-V actually

1. Software-oriented assembly model → used for compiler, app
2. Single-cycle processor → used for processor

Prove they're "the same" RISC-V:

Standard simulation argument.

Others

"Continuation-passing style" semantics makes proof easier

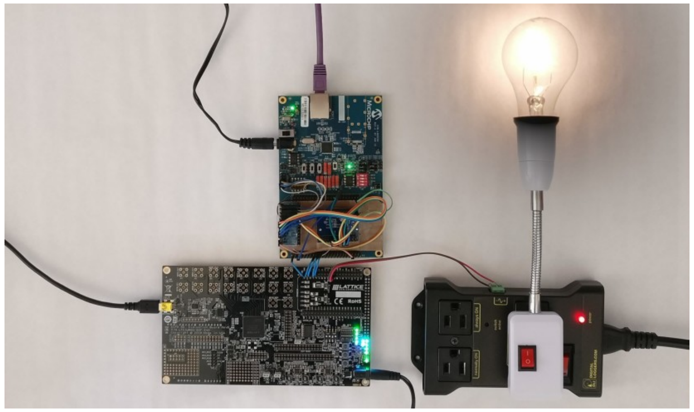
Implementation & Evaluation

Implementation

Processor, compiler & app all developed in coq.

Processor: extracted to HDL then compiled and burnt to FPGA

App: run coq compiler to RISC-V code.



Evaluation

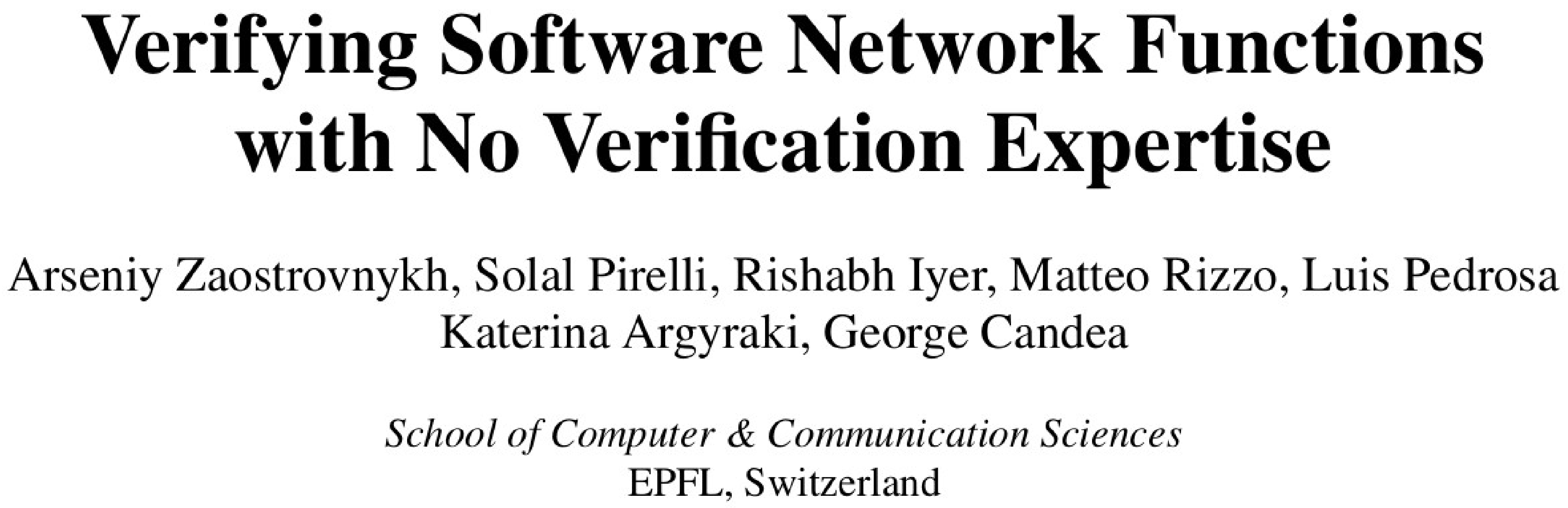
Light bulb app

10x slower than traditional dev (C + gcc + FE310 soc)

- 1.7x from implementation
- 2.1x from compiler
- 2.7x from processor

Network functions (SOSP’19)

NF Verification: SOSP’19



Verifying Software Network Functions with No Verification Expertise

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

School of Computer & Communication Sciences
EPFL, Switzerland

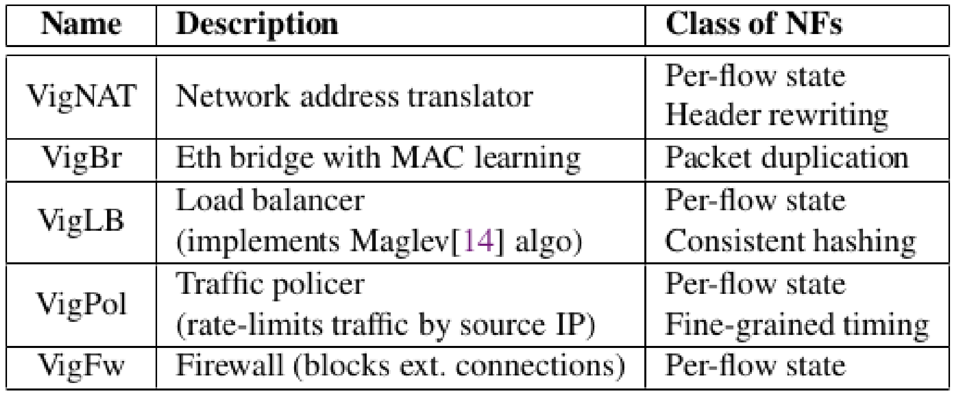
Vigor: Problem

Background: Network functions

e.g. NAT → previous work: VigNAT (SIGCOMM'17)

Specific modules within a network that implements specific functionalities.

→ SDN slang "NF virtualization": NF originally implemented by proprietary hardware, should now be software-defined



Name	Description	Class of NFs
VigNAT	Network address translator	Per-flow state Header rewriting
VigBr	Eth bridge with MAC learning	Packet duplication
VigLB	Load balancer (implements Maglev[14] algo)	Per-flow state Consistent hashing
VigPol	Traffic policer (rate-limits traffic by source IP)	Per-flow state Fine-grained timing
VigFw	Firewall (blocks ext. connections)	Per-flow state

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

Vigor: Problem

Push-button verification of network functions

NF logic is in C

NF Spec is in python, and is provided NF developer

The proof is automatic

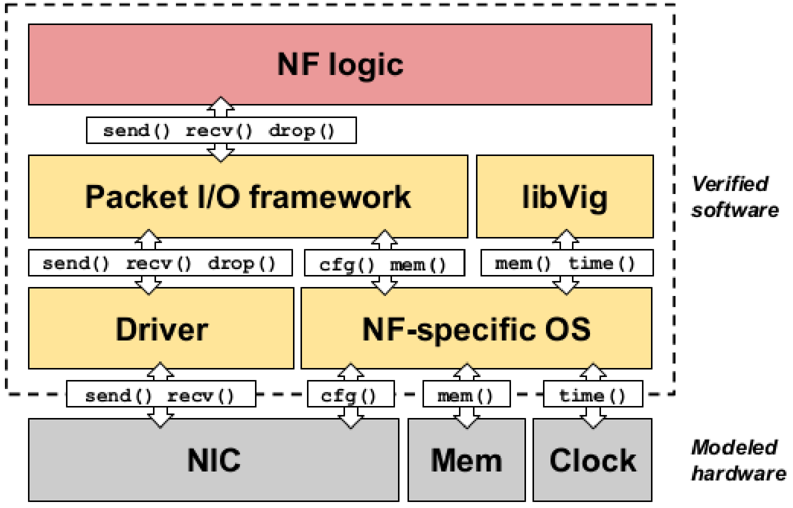


Figure 1. The Vigor stack for running NFs.

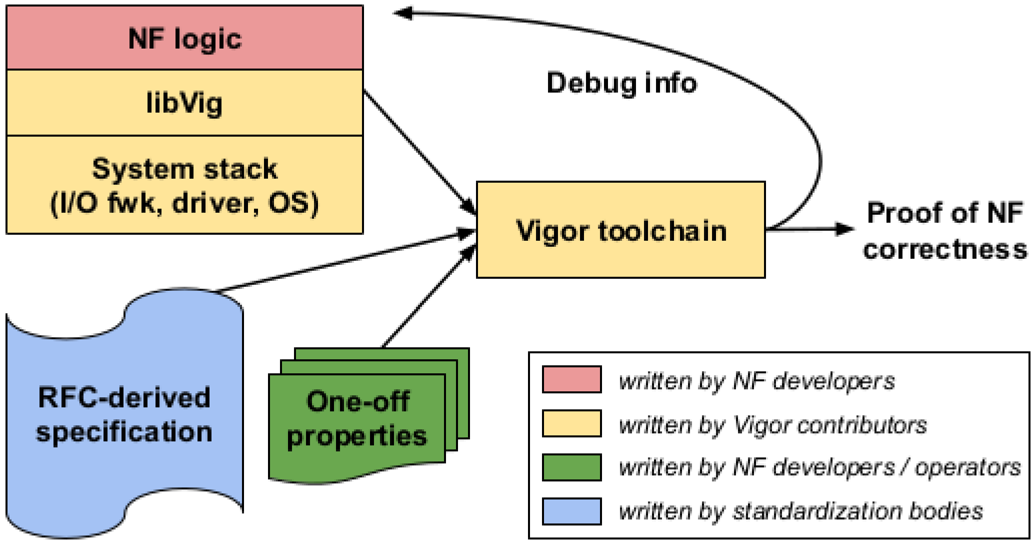

NF logic
libVig
System stack (I/O fwk, driver, OS)
Debug info
Vigor toolchain
Proof of NF correctness
RFC-derived specification
One-off properties
written by NF developers
written by Vigor contributors
written by NF developers / operators
written by standardization bodies

Figure 2. Vigor workflow: who writes what.

Vigor: Problem

Push-button verification of network functions

Contribution

1. *Push-button verification*: actually combining Theorem Proving and Automatic Symbolic Execution
2. *Full-stack NF verification*: NF + DPDK + custom NFOS
3. *Pay-as-you-go verification*: prove early, cost less

Vigor: Full-stack verification

Why

DPDK & NIC drivers are not trustworthy

How

Model only hardware, not DPDK

Only a fraction of DPDK called, and mostly upon initialization

→ ASE

NFOS: needn't isolation (app are ASE'ed), needn't drivers (done in DPDK). 2000 loc.

Vigor: Push-button verification

Problem

- ASE style push-button verification is less expressive
- TP style requires heavy effort, at the least many annotations
- NF development uses lots of advanced functionalities
  - e.g. auto-expiry hash table → has lots of pointers making it impossible for ASE to prove

How

- Split the stateful & stateless
- Require states be stored within data structures from libVig

Vigor: Push-button verification

How

1. Do ASE to get traces → KLEE
2. Convert traces to C programs
3. Combine lemmas about libVig facilities
4. Do TP to verify each trace → VeriFast

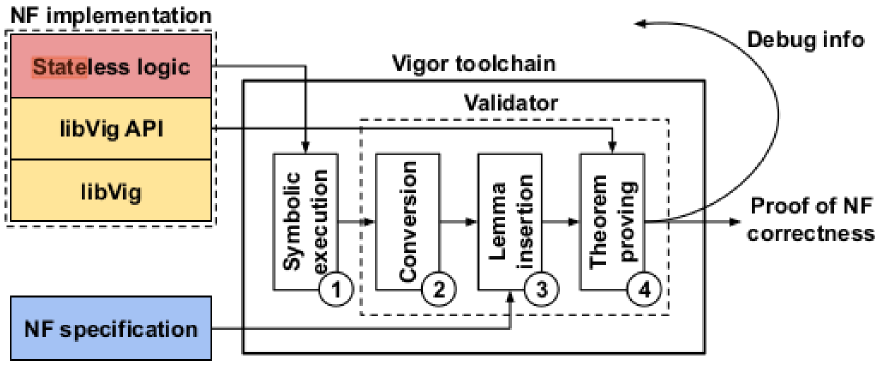


Figure 4. The Vigor verification process.

Why not all ASE?

ASE cannot go into libVig DS manipulation without getting lost

Vigor: pay-as-you-go proof

What’s contrary

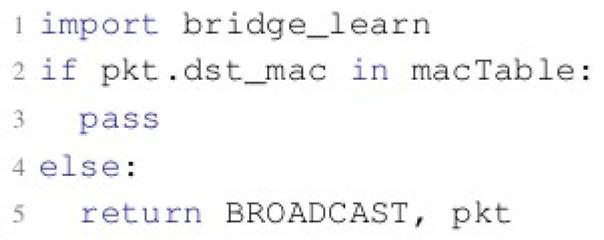
Traditional refinement (e.g. seL4, CertiKOS) requires an *abstract model* before proving any interesting properties

Example

Suppose we just focus on broadcasting behaviour

How

Just ASE

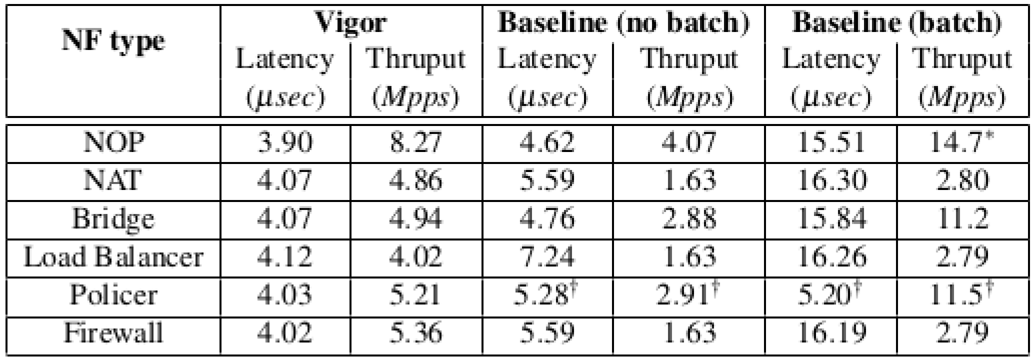


```
import bridge_learn
if pkt.dst_mac in macTable:
  pass
else:
  return BROADCAST, pkt
```

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

Implementation & Evaluation & Others

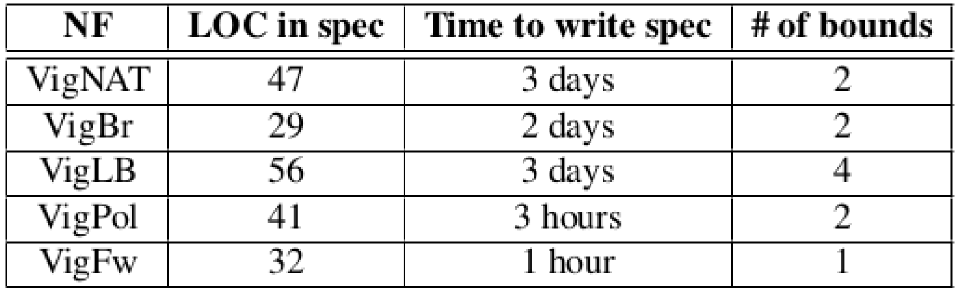
Overhead: acceptable (even better)



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

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

Proof burden



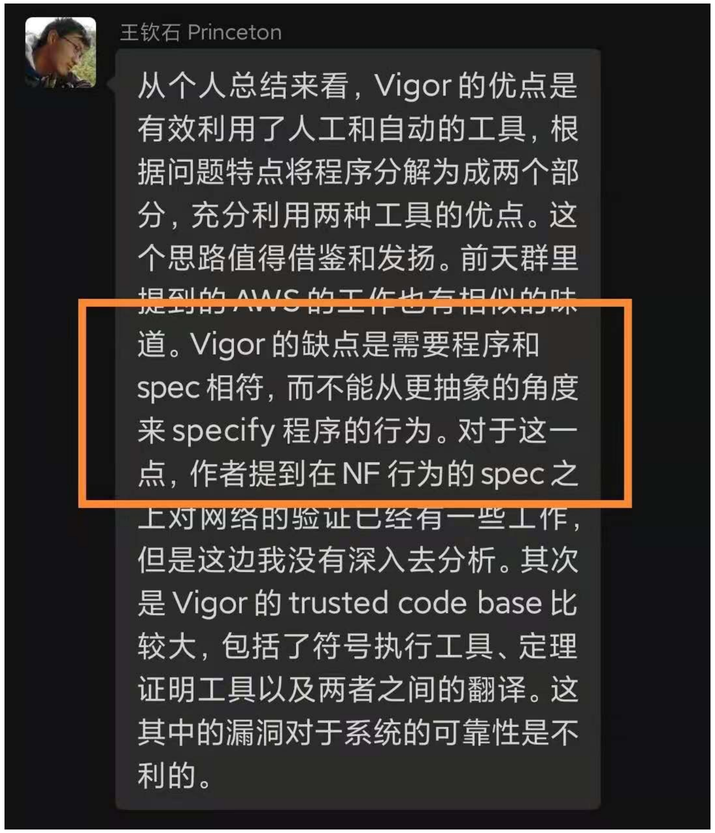
NF	LOC in spec	Time to write spec	# of bounds
VigNAT	47	3 days	2
VigBr	29	2 days	2
VigLB	56	3 days	4
VigPol	41	3 hours	2
VigFw	32	1 hour	1

Table 6. Statistics on writing NF specifications in Vigor.

Implementation & Evaluation & Others

Problems

- Large TCB
- Spec is code...
  - → hyperkernel



王钦石 Princeton

从个人总结来看，Vigor 的优点是有效利用了人工和自动的工具，根据问题特点将程序分解为成两个部分，充分利用两种工具的优点。这个思路值得借鉴和发扬。前天群里提到的 AWS 的工作也有相似的味道。Vigor 的缺点是需要程序和 spec 相符，而不能从更抽象的角度来 specify 程序的行为。对于这一点，作者提到在 NF 行为的 spec 之上对网络的验证已经有一些工作，但是这边我没有深入去分析。其次是 Vigor 的 trusted code base 比较大，包括了符号执行工具、定理证明工具以及两者之间的翻译。这其中的漏洞对于系统的可靠性是不利的。

Implementation & Evaluation & Others

Problems

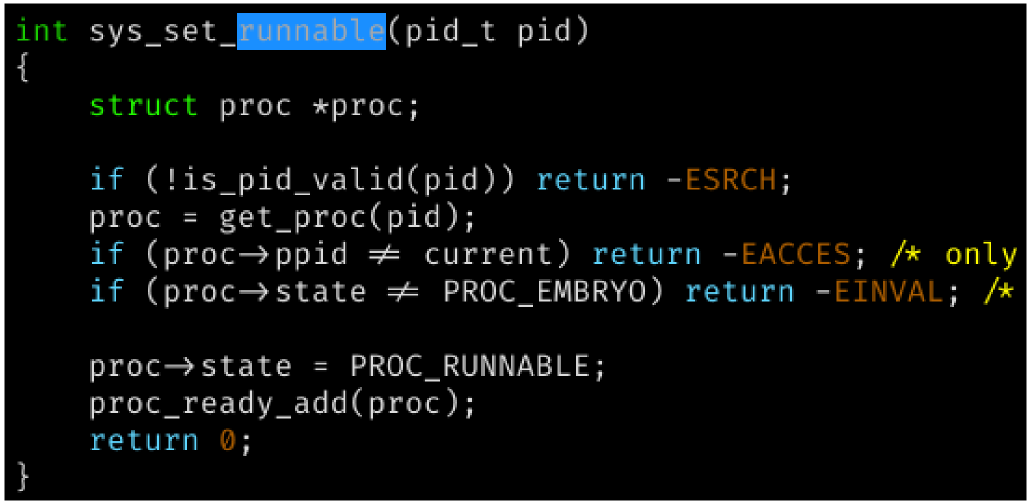
Large TCB

Spec is code…

→ e.g. hyperkernel

→ not inherent defect

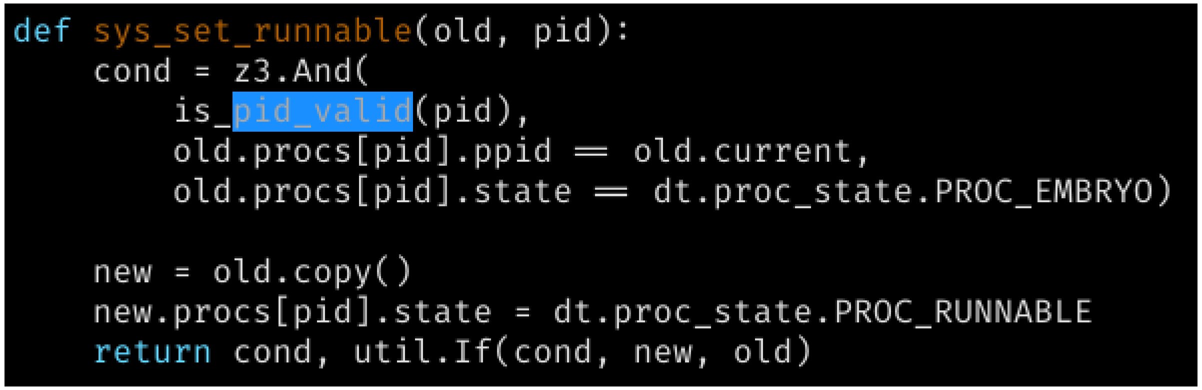
More than NF: Spec for whole SDN



```
int sys_set_runnable(pid_t pid)
{
    struct proc *proc;

    if (!is_pid_valid(pid)) return -ESRCH;
    proc = get_proc(pid);
    if (proc->ppid ≠ current) return -EACCES; /* only
    if (proc->state ≠ PROC_EMBRYO) return -EINVAL; /*

    proc->state = PROC_RUNNABLE;
    proc_ready_add(proc);
    return 0;
}
```



```
def sys_set_runnable(old, pid):
    cond = z3.And(
        is_pid_valid(pid),
        old.procs[pid].ppid == old.current,
        old.procs[pid].state == dt.proc_state.PROC_EMBRYO)

    new = old.copy()
    new.procs[pid].state = dt.proc_state.PROC_RUNNABLE
    return cond, util.If(cond, new, old)
```

Rust verification

Rust verification: Problem

Verification of software TEE

Hyperenclave by jyk

In Coq

with CertiK people and their techniques

Against the compiled MIR

Between surface rust & LLVM IR, the desugared IR

Why not other techniques

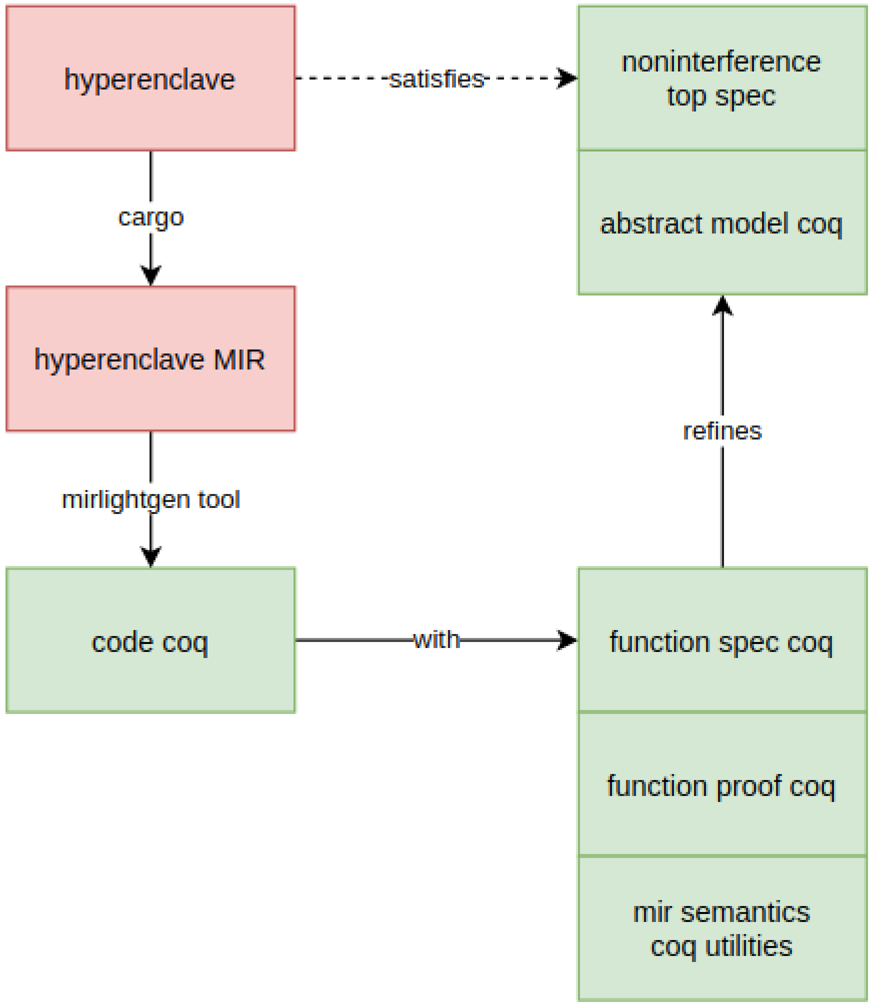
CRust (CBMC), Prusti (Boogie DV) ASE: too weak

Rust verification: Process

The good-old import way

→ Bedrock is export way

Same way as seL4, CertiKOS etc



Rust verification: Process

Verifying a single function

- The so-called code verification
- Pretty standard approach
- Not my expertise though

Rust verification: Noninterference

Definition

- ...

How

- Abstract machine definition
  - See code
- Transitions
- Lemmas
- Top-most theorem