The Science of Deep Specification

Benjamin C. Pierce
University of Pennsylvania

POST / ETAPS
April, 2018

Penn
UNIVERSITY of PENNSYLVANIA

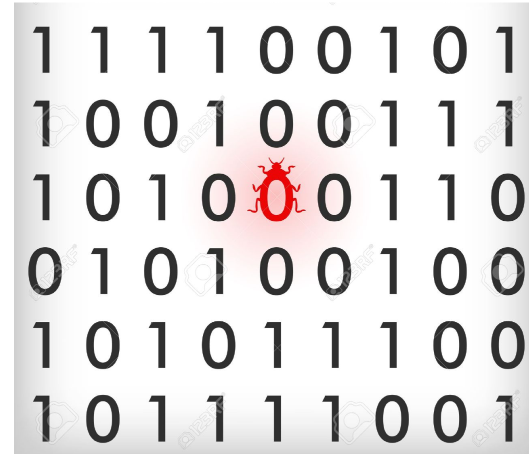
deep
spec

Toward a

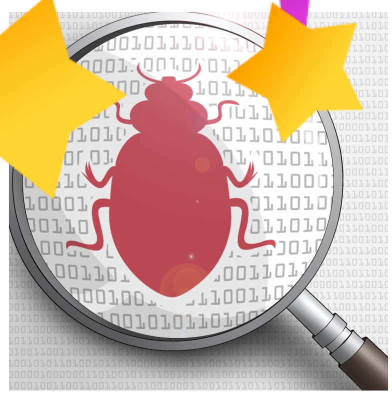
~~The~~ Science of Deep Specification

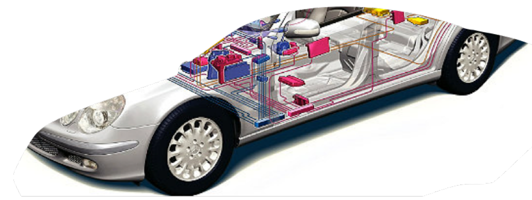
Benjamin C. Pierce
University of Pennsylvania

POST / ETAPS
April, 2018

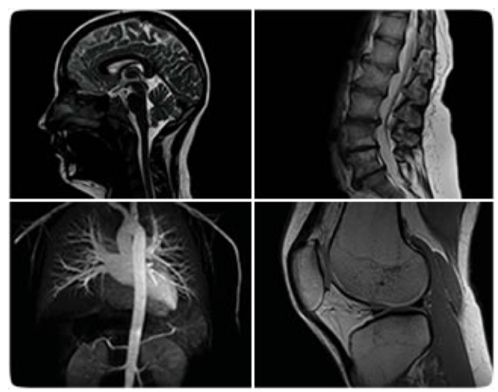


"We can't build software that works!"





Or…?

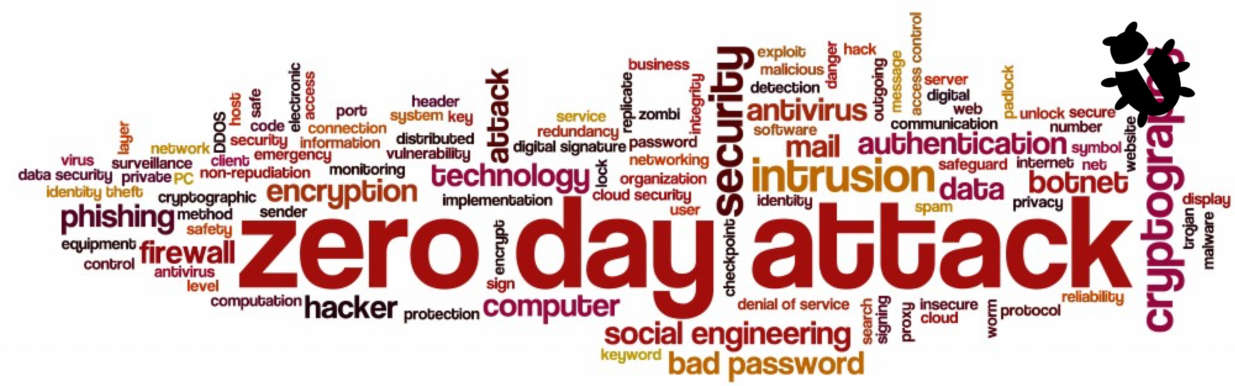


How did that happen?

- Better programming languages
  - Powerful mechanisms for *abstraction* and *modularity*
- Better software development methodology
  - Agile workflows, unit testing, …
- Stable platforms and frameworks
  - Posix, Win32, Android, iOS, apache, DOM/JS, …


Are we done?
No

What about secure software?



Grounds for hope…

- Better programming languages :-)
  - Basic *safety guarantees* built in
- Better understanding of risks and vulnerabilities
- Better system architectures for security
  - Separation kernels, hypervisors, sandboxing, TPMs, …
- Success stories of formal specification and machine-checked verification of critical software at scale
  - Not a panacea (side channels, etc.)
  - But a promising step in the right direction!

A Short Story

about a tiny compiler

and its specification(s)…

A datatype of stack machine instructions

```
Inductive instr : Type :=
| PUSH : nat -> instr
| PLUS : instr
| MINUS : instr
| MULT : instr.

Definition my_favorite_instructions
                                  : list instr :=
  [PUSH 10; PUSH 4; MULT; PUSH 2; PLUS].
```

An example instruction sequence

(All examples in Gallina, the language of the Coq proof assistant)

Starting stack

Program

Final stack

```
Fixpoint execute (s : list nat) (p : list instr) : list nat :=
  match (s, p) with
  | (_,        nil)              => s
  | (_,        (PUSH n) ::p') => execute (n    ::s)  p'
  | (m::n::s', PLUS      ::p') => execute ((m+n)::s') p'
  | (m::n::s', MINUS     ::p') => execute ((m-n)::s') p'
  | (m::n::s`, MULT      ::p') => execute ((m*n)::s') p'
  | (_,        _         ::p') => execute s          p'
  end.
```

Operational semantics of the stack machine

A datatype of arithmetic expressions

```
Inductive exp : Type :=
  | Num : nat -> exp
  | Plus : exp -> exp -> exp
  | Minus : exp -> exp -> exp
  | Mult : exp -> exp -> exp.

Definition my_favorite_number : exp :=
  Plus (Mult (Num 10) (Num 4)) (Num 2).
```

An example value belonging to the type exp

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
  end.
```

A compiler from arithmetic expressions to stack instructions

Specifying our compiler…

An Informal Specification

Compiling an arithmetic expression should yield stack-machine instructions that compute the corresponding numeric result:

- (Plus e1 e2) means add the results of e1 and e2
- (Minus e1 e2) means subtract the results of e1 and e2
- (Mult e1 e2) means multiply the results of e1 and e2

Formal	✗
Live	✗
Rich	

A (Very) Simple Formal Specification

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
  end.
```

Types!

Formal
Live
Rich ✗

Another Simple Formal Specification

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
  end.

Example e1 : assert (eq (compile (Num 42))
                        [PUSH 42]).

Example e2 : assert (eq (compile (Plus (Num 2) (Nu
                        [PUSH 2; PUSH 2; PLUS]).
```

Unit tests

Formal

Live

Rich /✗

```
Fixpoint compile (e : exp) : lis
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1
  | Minus e1 e2 => compile e1
  | Mult e1 e2 => compile e1 ++ c
  end.

Example e1 : assert (eq (compile (Num 42))
                        [PUSH 42]).

Example e2 : assert (eq (compile (Plus
                        [PUSH 2; PUSH 2; P
```

We don't really care what instructions we generate: we just want executing them to give the right answer!

…which raises the question: What *is* the "right answer"?

For Coq savants:

```
Definition assert b := (b = true).
```

```
Fixpoint eval (e : exp) : nat :=
  match e with
  | Num n => n
  | Plus e1 e2 => (eval e1) + (eval e2)
  | Minus e1 e2  => (eval e1) - (eval e2)
  | Mult e1 e2 => (eval e1) * (eval e2)
  end.

Definition compiles_correctly (e : exp) : bool :=
  eq (execute [] (compile e)) [eval e].
```

"Executing the compiled code in an empty stack…

yields a stack containing the result of `eval`*uating the original expression."*

Operational semantics of the source language

```
Example e3 :
  assert (compiles_correctly (Plus (Num 2) (Num 2))).


Example e4 :
  assert (compiles_correctly (Plus (Num 5) (Num 3))).


Example e5 :
  assert (compiles_correctly (Mult (Num 0) (Num 3))).


Example e6 :
  assert (compiles_correctly (Mult (Num 2) (Num 2))).
```

```
Example e7 :
  assert (compiles_correctly (Mult (Num 3) (Num 1))).
```

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e1 ++ [MULT]
  end.
```

```
Example e7 :
  assert (compiles_correctly (Mult (Num 3) (Num 1))).
```

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
  end.
```

Enumerative

etc.

Specification-Based Testing

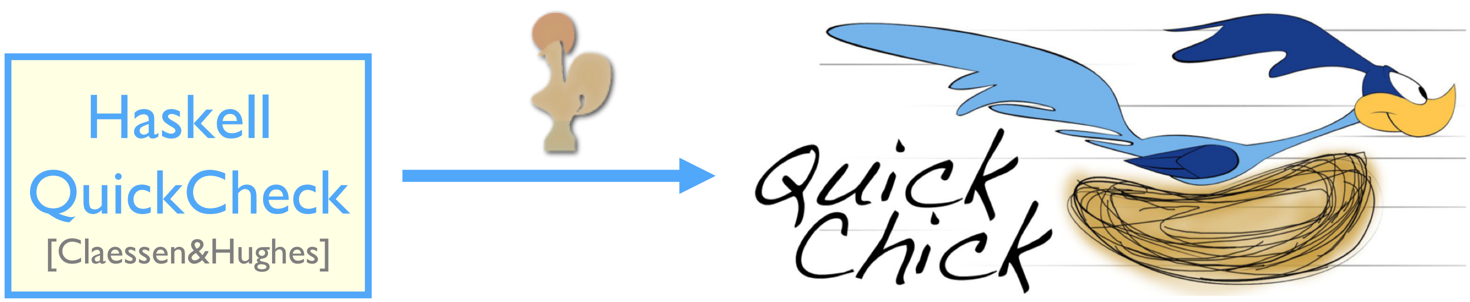
Random

Concolic

etc.

Specification-Based *Random* Testing

- Generate lots of random expressions
- For each, see if `compiles_correctly` returns `true`
- If a failing example is found, “shrink” it (by greedy search) to a minimal failing example



```
QuickChick compiles_correctly.
```

Counterexample found after 4 tests and 8 shrinks:

Minus (Num

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
  end.
```

QuickChick

Counterexa

shrinks:

Minus (Num

```
Fixpoint execute (s : list nat) (p : list instr) : list nat :=
  match (s, p) with
  | (_,         nil)          => s
  | (_,         (PUSH n) ::p') => execute (n    ::s)  p'
  | (m::n::s', PLUS      ::p') => execute ((m+n)::s') p'
  | (m::n::s', MINUS     ::p') => execute ((m-n)::s') p'
  | (m::n::s', MULT      ::p') => execute ((m*n)::s') p'
  | (_,         _        ::p') => execute s           p'
  end.
```

```
Fixpoint compile (e : exp) : list instr :=
  match e with
  | Num n => [PUSH n]
  | Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
  | Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
  | Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
  end.
```

`compile` leaves the results of subexpressions in the wrong order on the stack!

Beyond Testing…

What else can we do with a specification?

- Synthesize programs that satisfy it
- Build run-time monitors that check for violations
- Prove that an implementation satisfies it

```
Theorem compile_correct : forall e,
  assert (compiles_correctly e).
```

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.

Lemma execute_eval_comm : forall e stack,
  execute stack (compile e) = eval e :: stack.

Theorem compile_correct : forall e,
  assert (compiles_correctly e).
```

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.

Lemma execute_eval_comm : forall e stack,
  execute stack (compile e) = eval e :: stack.

Theorem compile_correct : forall e,
  assert (compiles_correctly e).
```

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
    - reflexivity.
    - destruct a.
      + intros. simpl. rewrite IHp1.
        reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
Qed.
```

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
    - reflexivity.
    - destruct a.
      + intros. simpl. rewrite IHp1.
        reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
Qed.
```

No automation

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
    - reflexivity.
    - destruct a; simpl; intros;
      destruct stack as [|x [|y stack']];
      try rewrite IHp1; reflexivity.
Qed.
```

Simple automation

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
    - reflexivity.
    - destruct a.
      + intros. simpl. rewrite IHp1.
        reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
      + intros. simpl.
        destruct stack as [|x [|y stack']].
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
        * rewrite IHp1. reflexivity.
Qed.
```

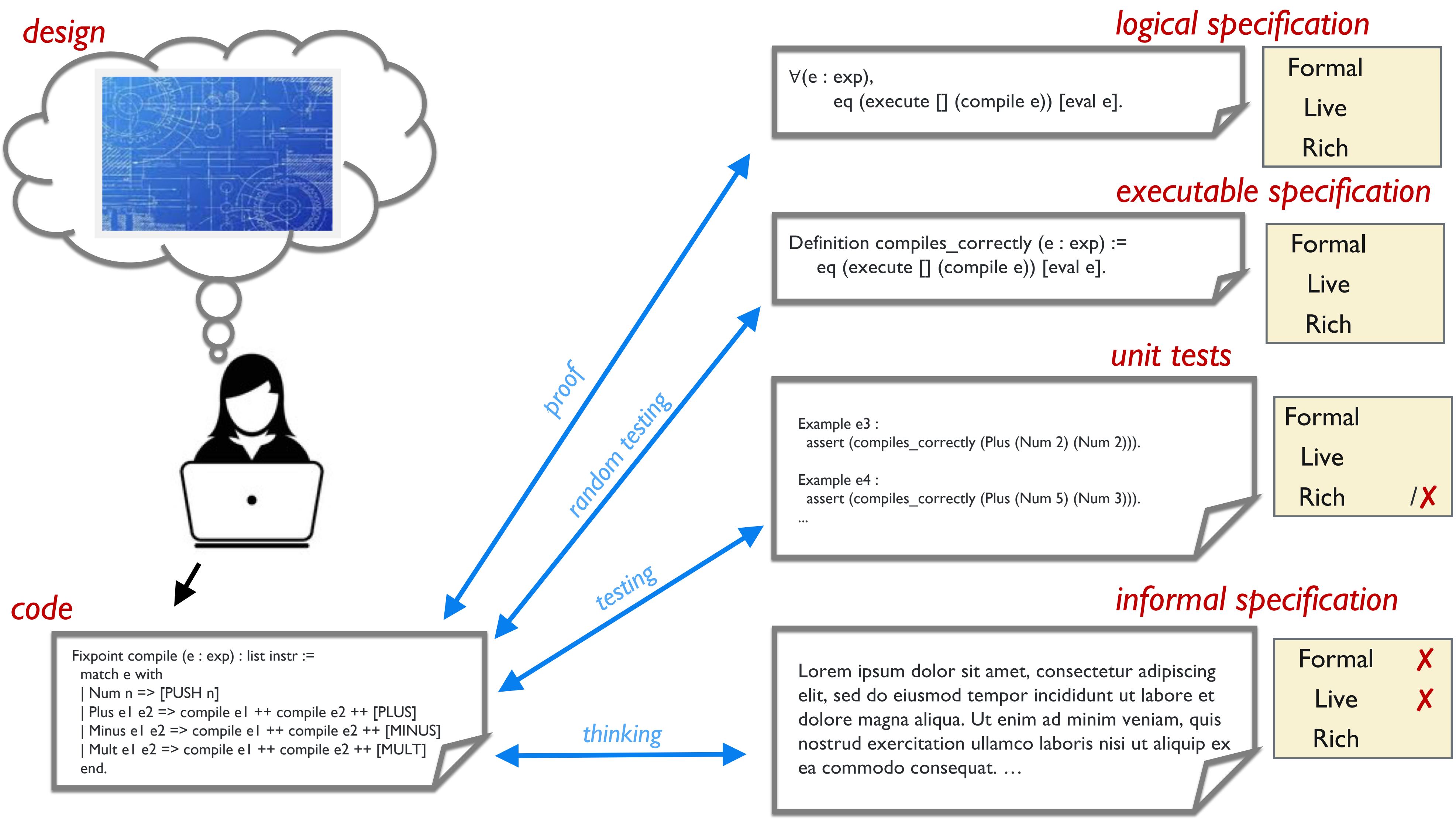
No automation

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1.
    - reflexivity.
    - destruct a; simpl; intros;
      destruct stack as [|x [|y stack']];
      try rewrite IHp1; reflexivity.
Qed.
```

Simple automation

```
Lemma execute_app : forall p1 p2 stack,
    execute stack (p1 ++ p2)
  = execute (execute stack p1) p2.
Proof.
  induction p1;
    try (destruct a);
    try (destruct stack
             as [|x [|y stack']]);
    crush.
Qed.
```

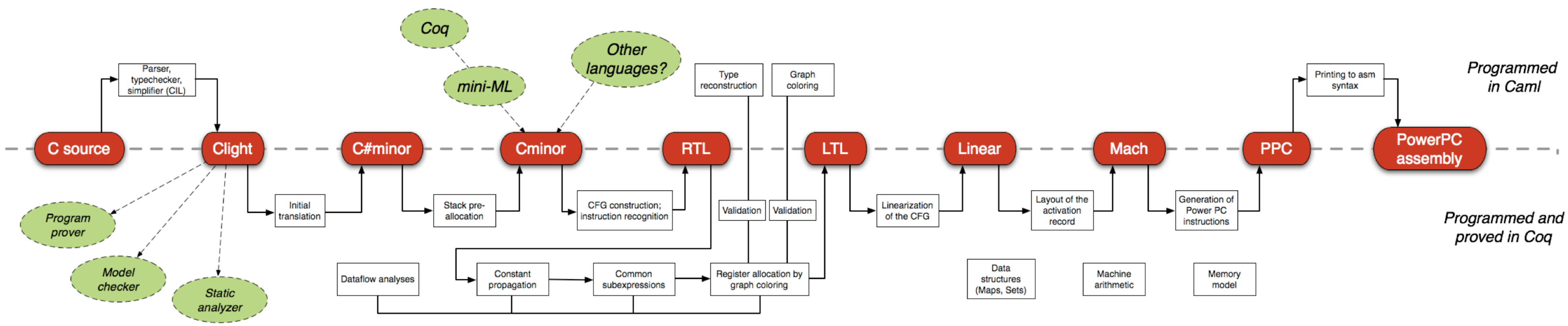
Chlipala automation


design
code
Fixpoint compile (e : exp) : list instr :=
match e with
| Num n => [PUSH n]
| Plus e1 e2 => compile e1 ++ compile e2 ++ [PLUS]
| Minus e1 e2 => compile e1 ++ compile e2 ++ [MINUS]
| Mult e1 e2 => compile e1 ++ compile e2 ++ [MULT]
end.
proof
random testing
testing
thinking
logical specification
∀(e : exp),
eq (execute [] (compile e)) [eval e].
Formal
Live
Rich
executable specification
Definition compiles_correctly (e : exp) :=
eq (execute [] (compile e)) [eval e].
Formal
Live
Rich
unit tests
Example e3 :
assert (compiles_correctly (Plus (Num 2) (Num 2))).
Example e4 :
assert (compiles_correctly (Plus (Num 5) (Num 3))).
...
Formal
Live
Rich /✗
informal specification
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. …
Formal ✗
Live ✗
Rich

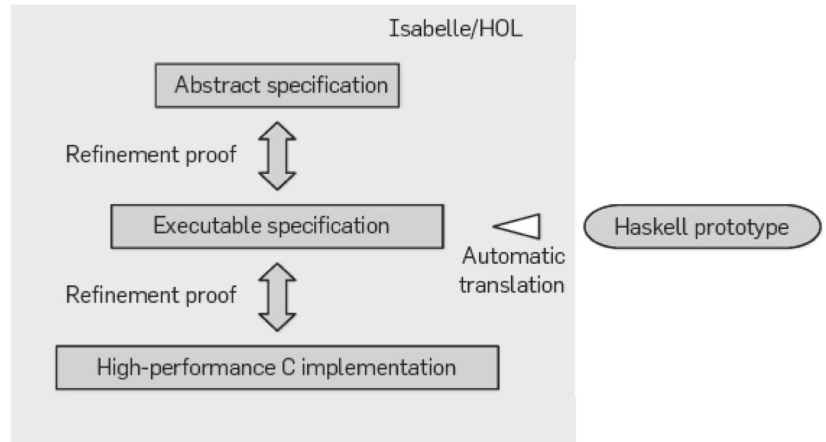
nice story

does it scale?

COMPCERT



- Accepts most of ISO C 99
- Produces machine code for PowerPC, ARM, x86 (32-bit), and RISC-V architectures
- 90% of the performance of GCC (v4, opt. level 1)



- Real-world operating-system kernel
- With an end-to-end proof of implementation correctness and security enforcement
- Verified down to machine code

NICTA

CertiKOS

Certified Kit Operating System

YALE

Certified OS Kernels

Clean-slate design with end-to-end guarantees on extensibility, security, and resilience. Without Zero-Day Kernel Vulnerabilities.

Layered Approach

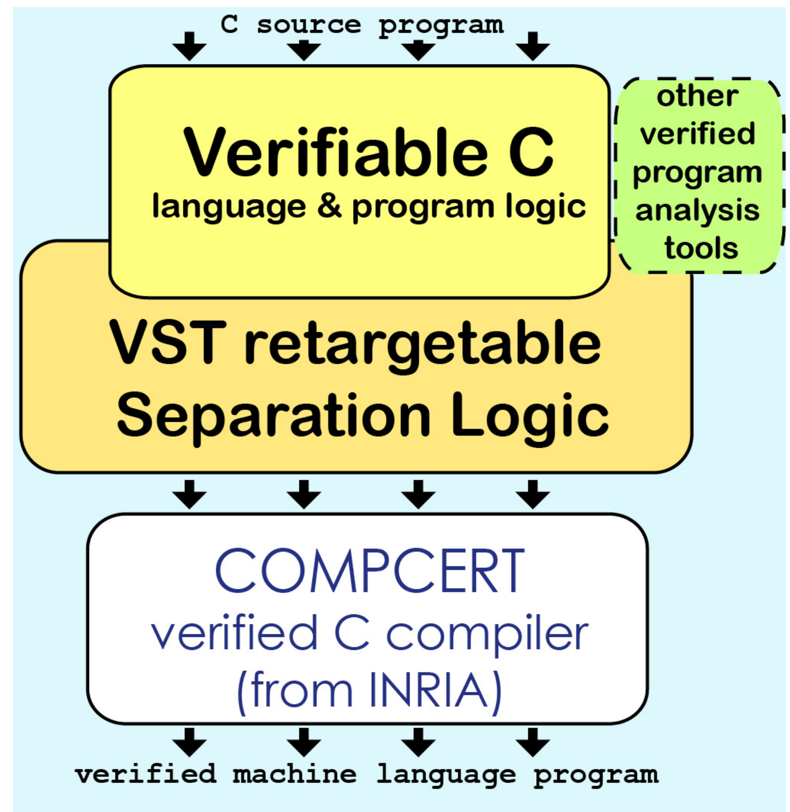
Divides a complex system into multiple certifeid abstraction layers, which are deep specifications of their underlying implementations.

Languages and Tools

New formal methods, languages, compilers and other tools for developing, checking, and automating specs and proofs.

Verified
Software
Toolchain

- C verification framework based on higher-order separation logic in Coq
- Verified implementations of OpenSSL-HMAC and SHA-256
- working on additional cryto primitives (HMAC-based Deterministic Random Byte Generation, AES), parts of TweetNaCL



And many, many more!

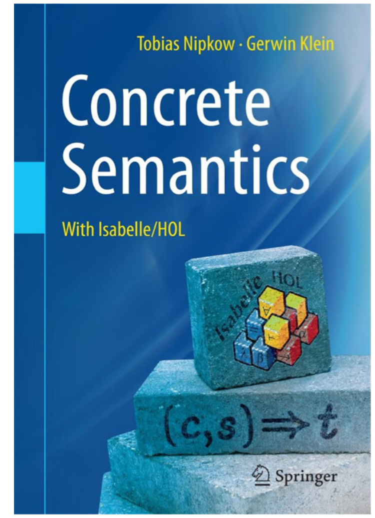
- Bedrock system
- Ur/Web compiler
- CompCert TSO compiler
- CompCert static analysis tools
- Jitk and Data6 verified filesystems
- Fscq file system from MIT
- Verdi distributed system framework
- Testable formal spec for AutoSAR
- CakeML compiler
- Vellvm: Verified LLVM optimizations
- IronClad Apps
- Full-scale formal specifications of critical system interfaces
  - X86 instruction set
  - TCP protocol suite
  - Posix file system interface
  - Weak memory consistency models for x86, ARM, PowerPC
  - ISO C / C++ concurrency
  - Elf loader format
  - C language (Cerberus – also see Krebbers, K semantics, …)

Kami

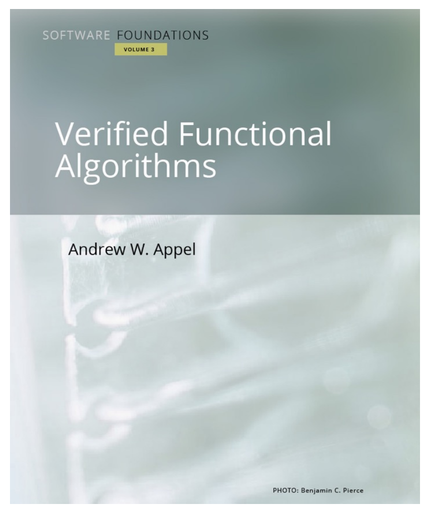
- Coq framework for implementing, specifying, verifying, and compiling Bluespec-style hardware components.
- E.g., a RISC-V implementation (w 4-stage pipeline), fully verified down to RTL

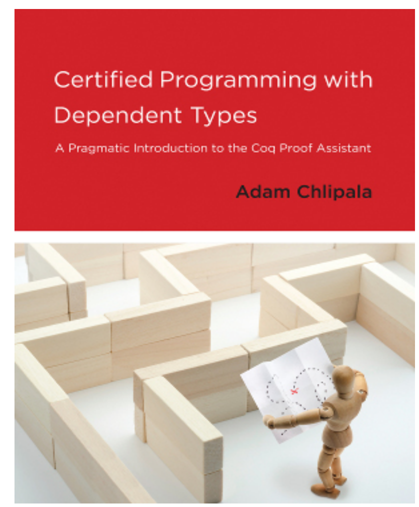
Verified Textbooks!

Isabelle



Coq





Why now?

Urgent need for increased confidence

+

Diminishing value of “paper proofs”

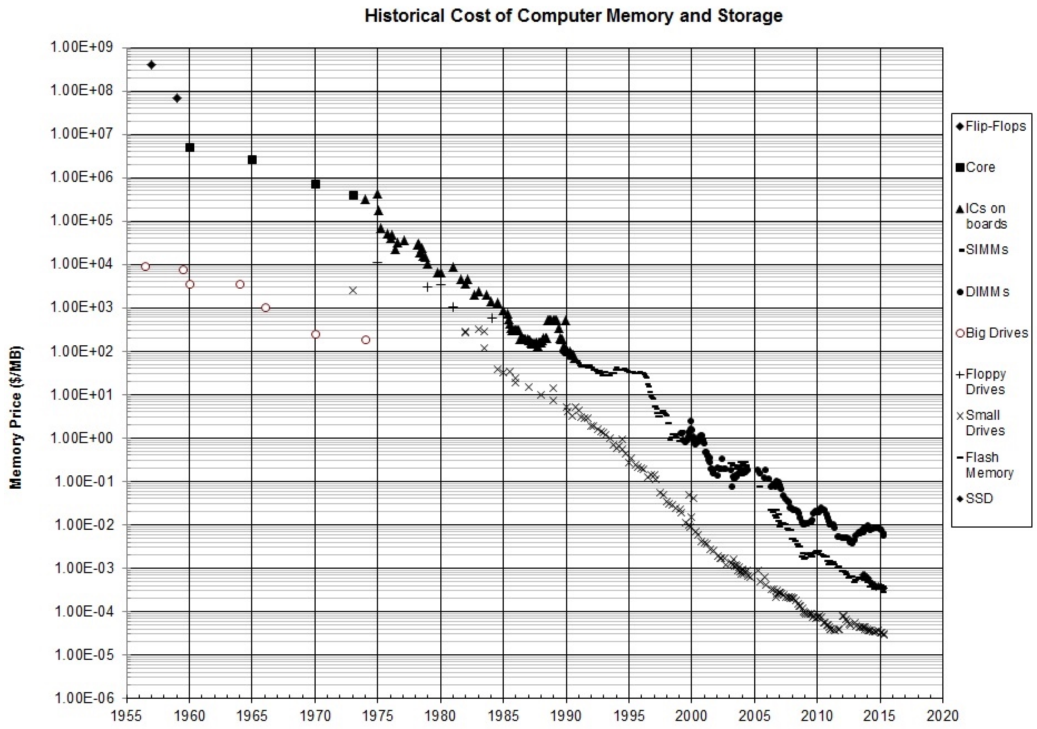
+

Progress on enabling technologies

Enabling Technologies

- Logics
  - Concurrent separation logic, …
- Proof assistants
  - Coq, Isabelle, ACL2, Twelf, HOL-light, …
- Testing tools and methodologies
  - QuickCheck, QuickChick, …
- DSLs for writing specifications
  - OTT, Lem, Redex, …
- Languages with integrated specifications
  - Dafny, Boogie, JML, F*, Liquid Types, Verilog PSL, Dependent Haskell, ...

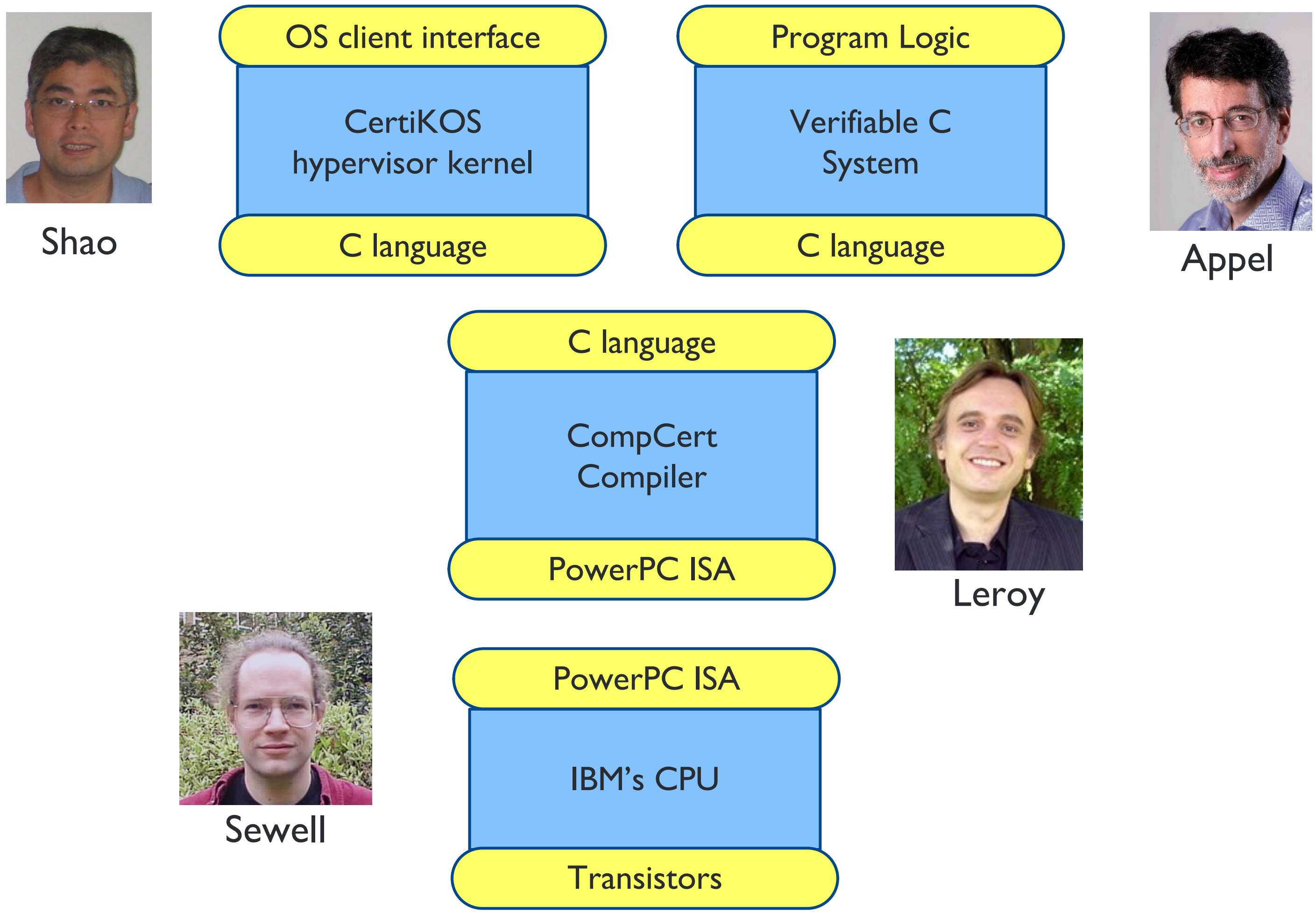
Enabling Technologies



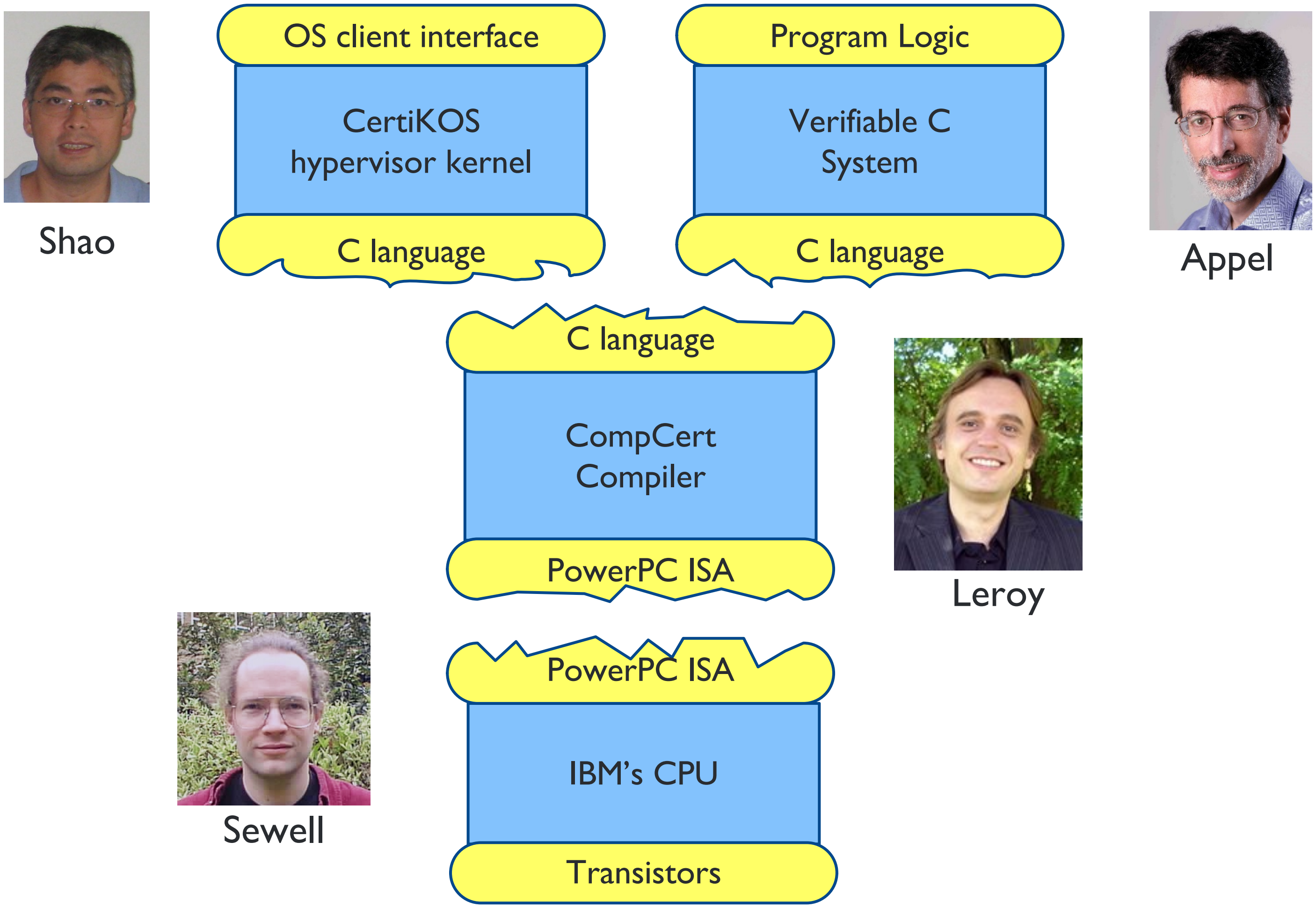
Are we done?

Nope.

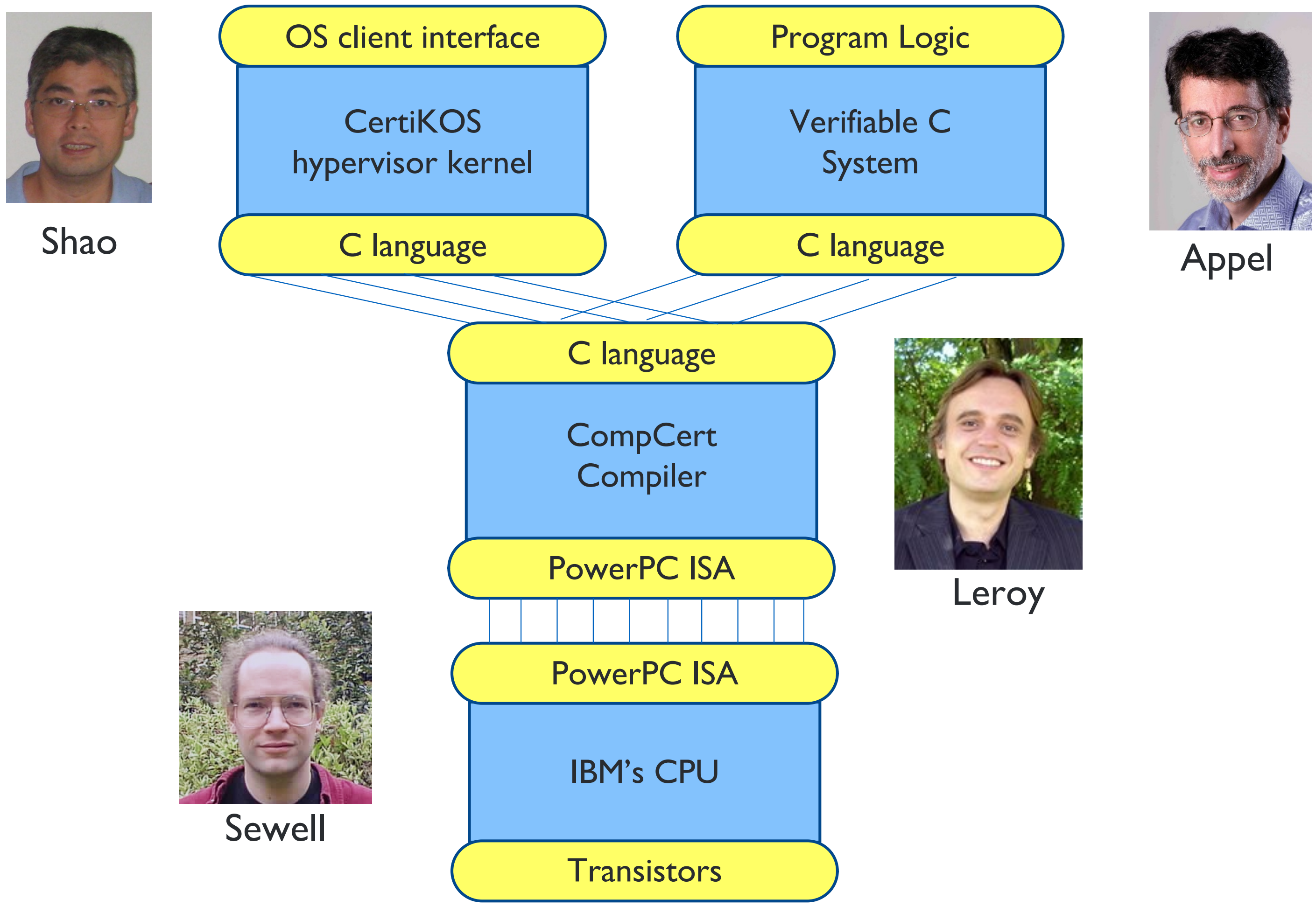
Lessons from CompCert



Lessons from CompCert



Lessons from CompCert

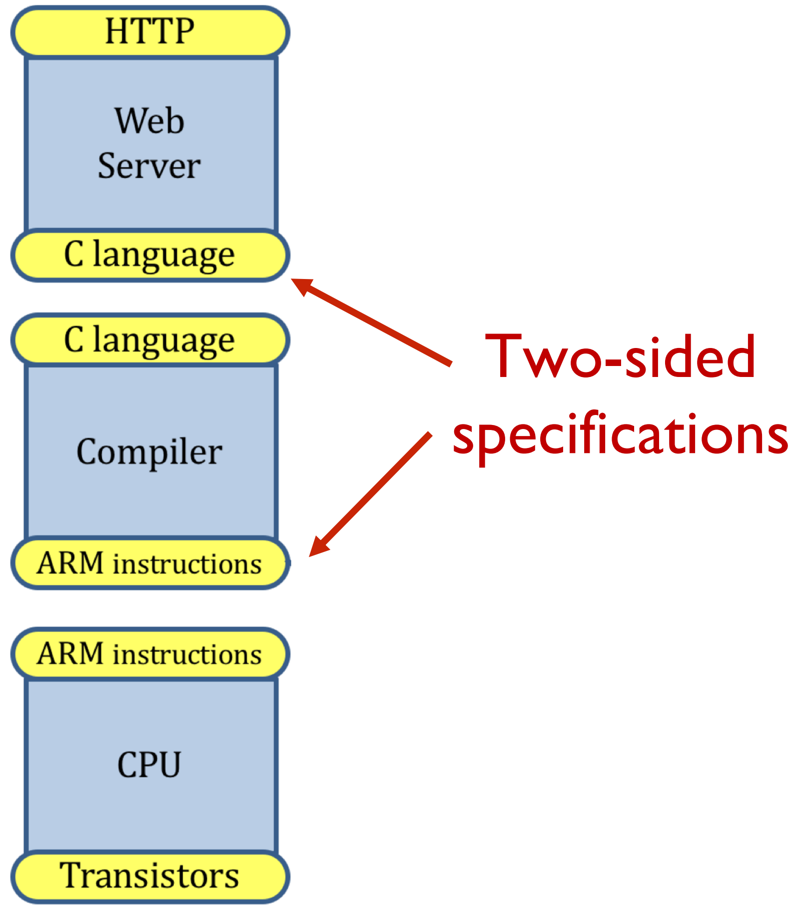


Lessons from seL4

- Original specification and correctness proof for seL4 kernel took **~20 person years**
- Later, the same team added a tool for setting up secure system configurations
  - where processes at different security levels were guaranteed not to interfere
- Proving correctness of this tool took ~4 person years, of which **1.5 years** were devoted to **upgrading the kernel specification** (and proof) to eliminate unwanted nondeterminism

Two-sided specifications

Verified components must connect at specification boundaries



“Deep” specifications:

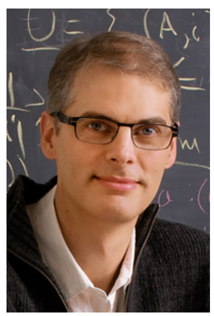
Formal	mathematically rigorous
Rich	precisely expressing intended behavior of complex software
Live	automatically checked against actual code (not just a model)
Two-sided	exercised by both “implementors” and “clients”

The Science of Deep Specification

Andrew Appel
Princeton

Steve Zdancewic
University of Pennsylvania

Adam Chlipala
MIT



Yours truly
University of Pennsylvania

Zhong Shao
Yale

Stephanie Weirich
University of Pennsylvania

And more importantly…

Andres Erbsen
Antal Spector-Zabusky
Antoine Voizard
Benjamin Sherman
Christine Rizkallah
David Costanzo
David Kaloper Meršinjak
Dmitri Garbuzov
Hernán Vanzetto
Jade Philipoom
Jason Gross
Ji-Yong Shin
Jieung Kim
Joachim Breitner
Joonwon Choi
Joshua Lockerman
Jérémie Koenig

Lennart Beringer
Leonidas Lampropoulos
Li-yao Xia
Lionel Rieg
Lucas Paul
Matthew Weaver
Mengqi Liu
Mirai Ikebuchi
Murali Vijayaraghavan
Nick Giannarakis
Olivier Savary Belanger
Pedro Henrique Avezedo de Amorim
Pierre Wilke
Qinxiang Cao
Quentin Carbonneaux
Richard Zhang

Ronghui Gu
Samuel Gruetter
Santiago Cuellar
Unsung Lee
Vilhelm Sjöberg
William Mansky
Wolf Honore
Xiongnan (Newman) Wu
Yao Li
Yishuai Li
Yuanfeng Peng
Yuting Wang
Zoe Paraskevopoulou

Goal:

Move from

point success stories

to

sustainable engineering practice at industrially relevant scale

Many parts

CERTIKOS

Verified Software Toolchain

Kami

deep spec crypto

Core Spec

Certi Coq

Vellvm verified LLVM

Quick Chick

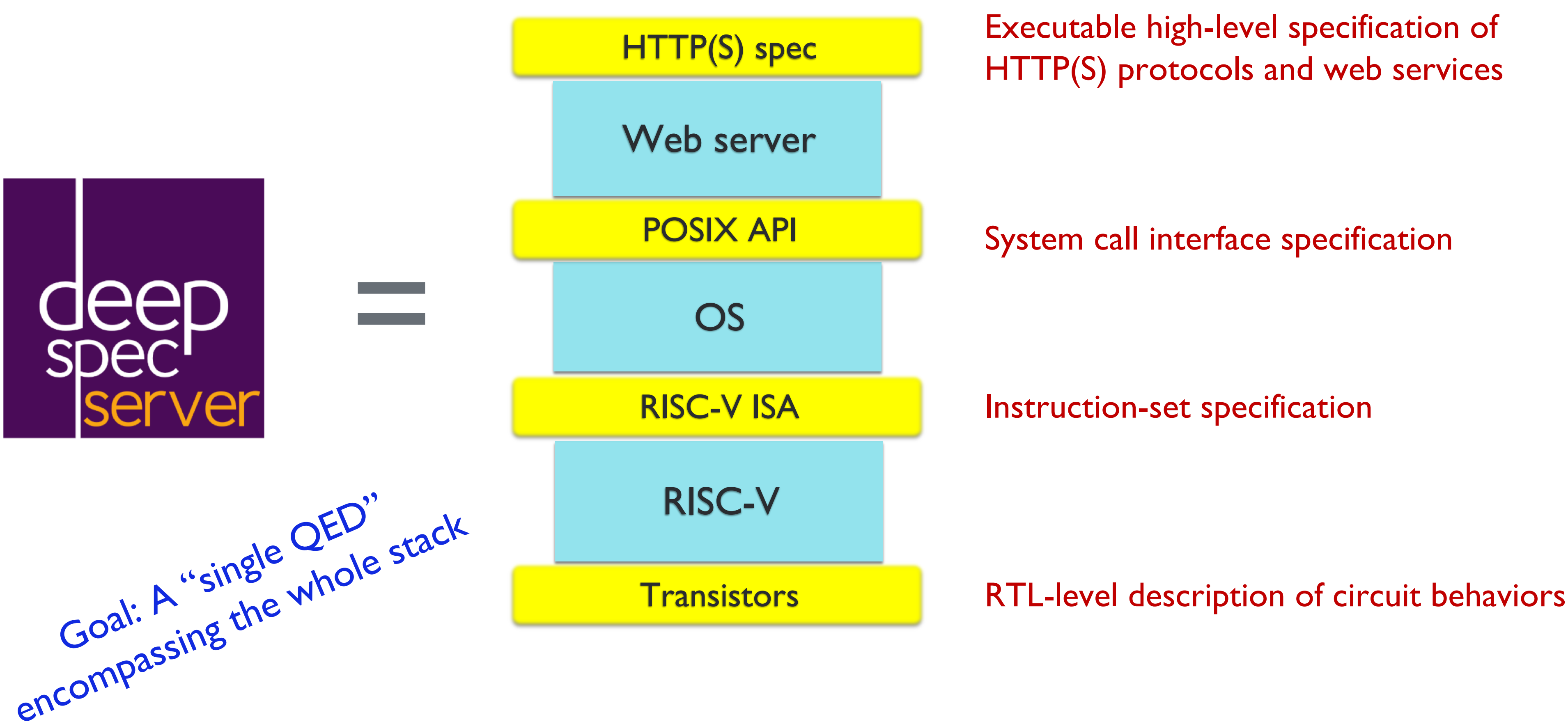
One whole

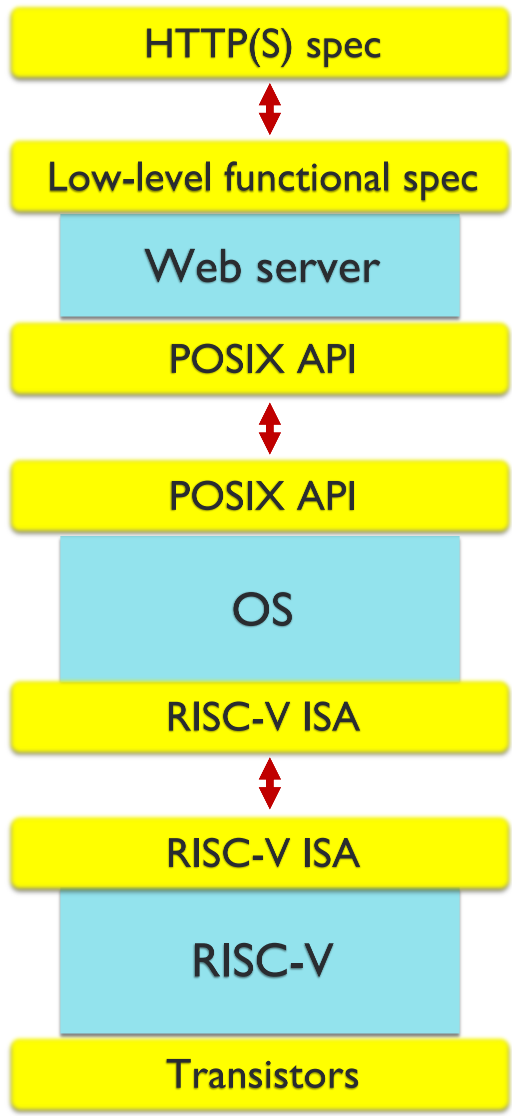
deep spec server

The DeepSpec Web Server

"Securing the Internet of Things"

- Based on popular `libmicrohttpd` library
  - Clean separation between core HTTP-level functionality (and specs) and the specifics of particular web services
- Aimed at embedded web servers
  - E.g. IoT device controllers
- Current state = simple first version
  - Parsing / printing of core HTTP formats
  - Basic GET / PUT functionality
  - ETag support for concurrency control
- Later:
  - Broader coverage of HTTP standard documents
  - TLS authentication
  - Support for database-backed web services


deep spec server
=
HTTP(S) spec
Web server
POSIX API
OS
RISC-V ISA
RISC-V
Transistors
Executable high-level specification of HTTP(S) protocols and web services
System call interface specification
Instruction-set specification
RTL-level description of circuit behaviors
Goal: A "single QED" encompassing the whole stack



Executable high-level specification of HTTP(S) protocols and web services

Functional program with same observable behavior as C web server

System call interface specification
(separation logic Hoare triples)

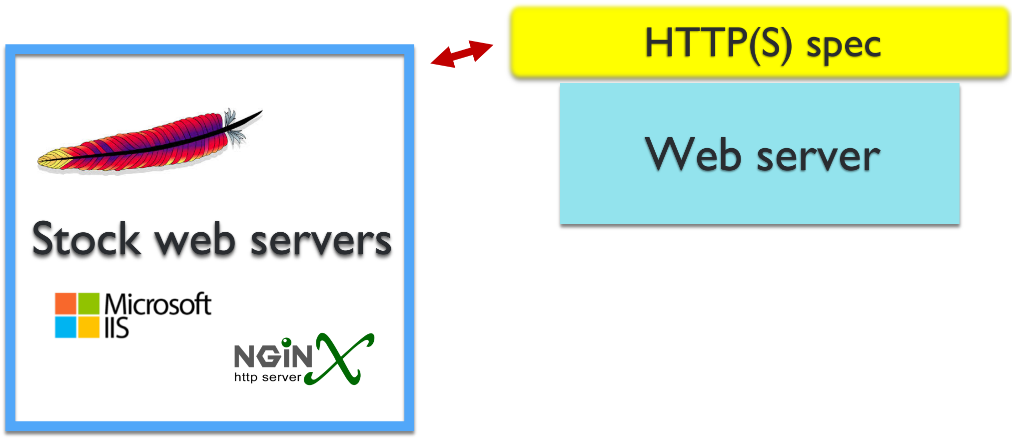
System call interface specification
(CertiKOS "layer interface")

Instruction-set specification
(assembly level, structured memory model)

Instruction-set specification
(machine-code level, flat memory model)

RTL-level description of circuit behaviors

Challenge:
A Testable High-Level Specification


HTTP(S) spec
Web server
Stock web servers
Microsoft
IIS
NGiNX
http server

Strategy:

Write specification in the form of an acceptance tester: a functional program that interacts with a server and accepts / rejects traces.

Status:

- Core HTTP(S) header formats
- Basic GET / PUT commands
- ETag commands for bandwidth reduction / concurrency control

Early results: Testing stock web servers

- Nginx
  - Passes all tests so far
- Apache
  - Nonstandard responses:
    - For GET requests that expect 200 OK, Apache sometimes closes connection before sending the full response
    - For GET requests that expect 404 Not Found, Apache sometimes responds 403 Forbidden
  - Wrong behavior:
    1. Unconditional PUT, return 204 No Content
    2. Unconditional GET, return 200 OK with ETag
    3. Conditional If-Match PUT with ETag from 2, return 412 Precondition Failed
    4. Unconditional GET, return 200 OK with content from 3

I.e.,. The server said it was rejecting our PUT, but actually executed it.

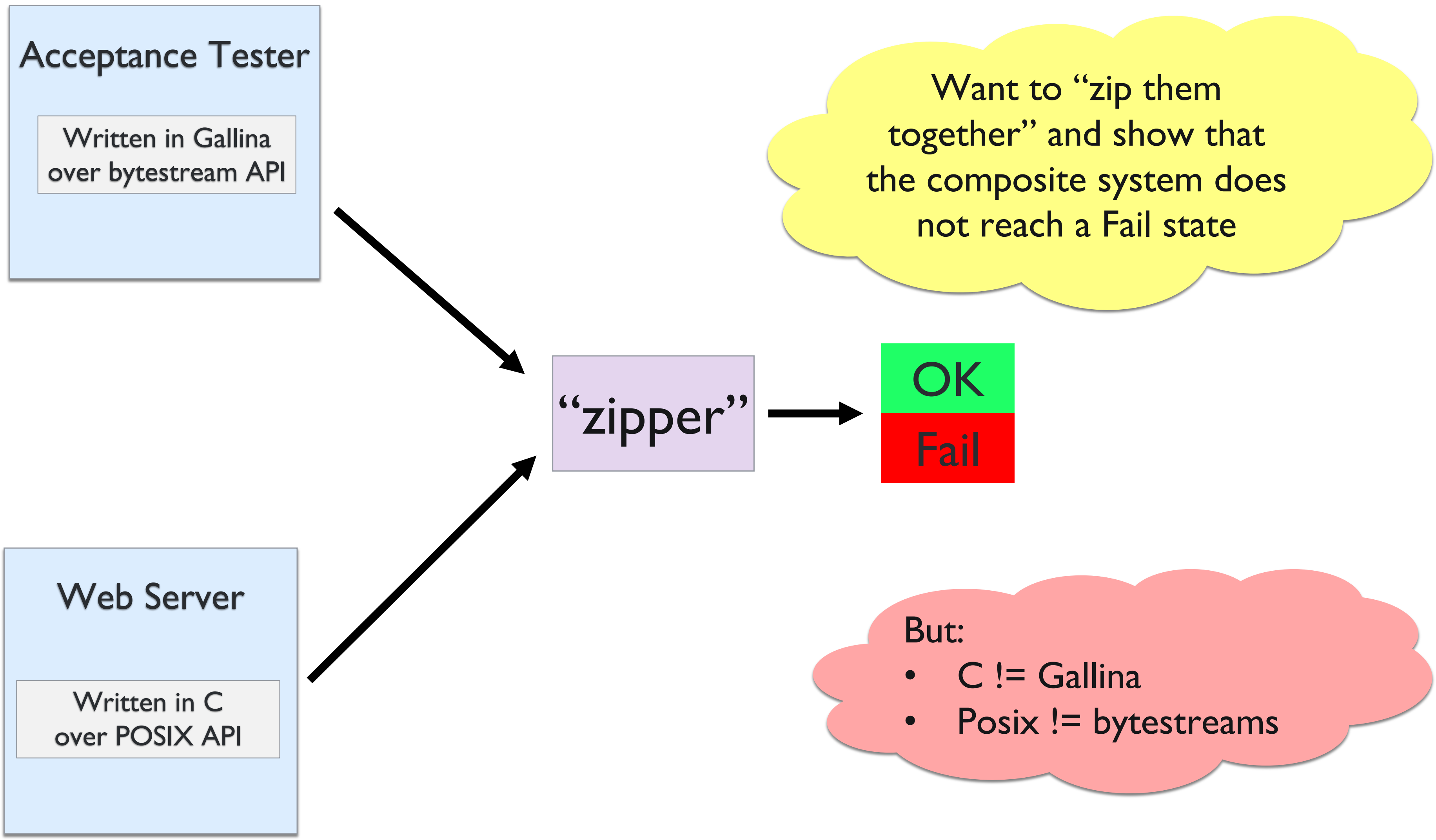
Ongoing Work

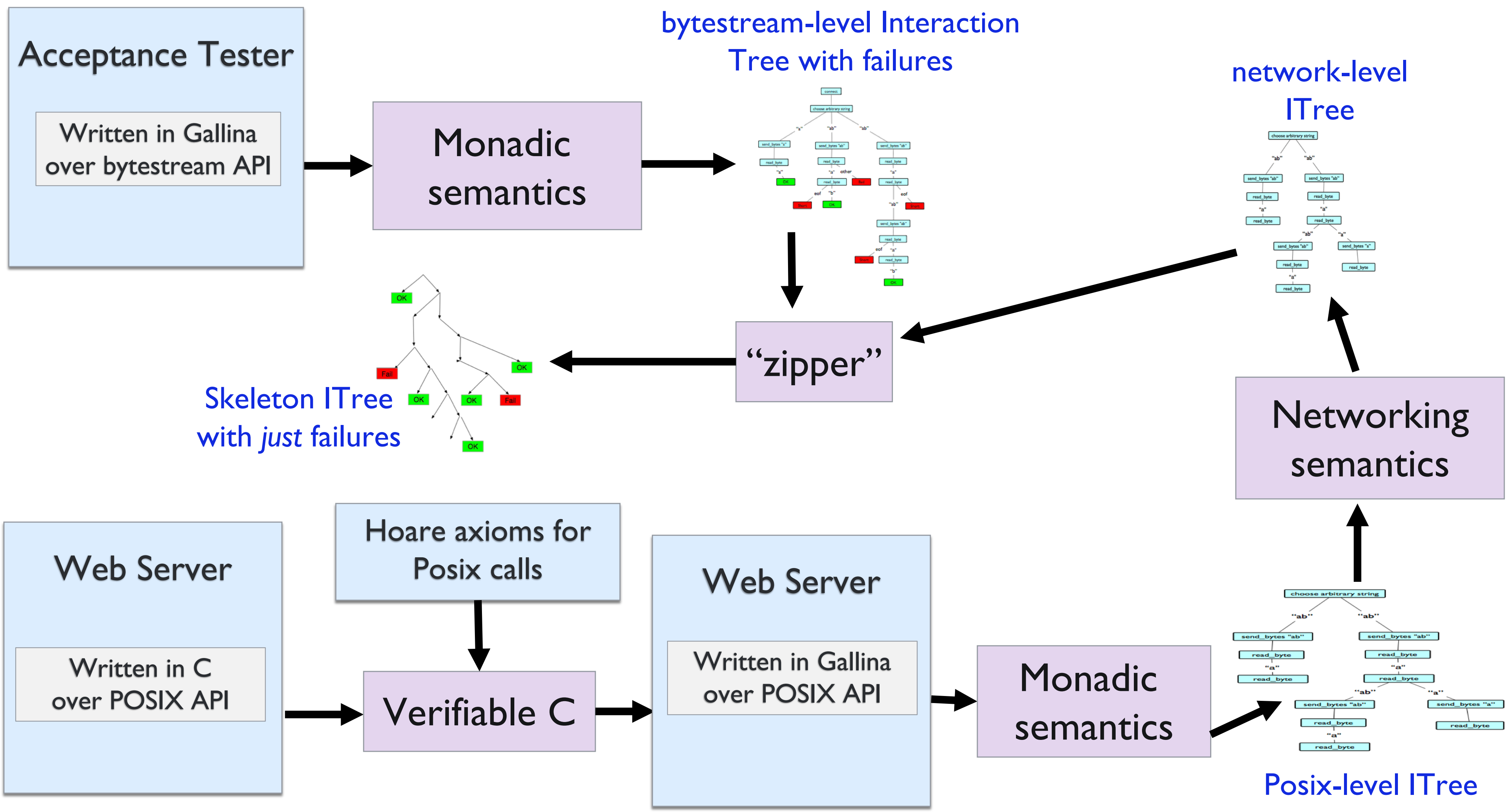
- More features of HTTP
  - Cookies
  - Authentication and encryption
  - Streaming
  - Etc., etc.
- Deeper testing of stock web servers
- More extensive “mutation testing”
  - to confirm that the test framework is able to detect manually inserted bugs

Challenge:
Unifying Specification Styles

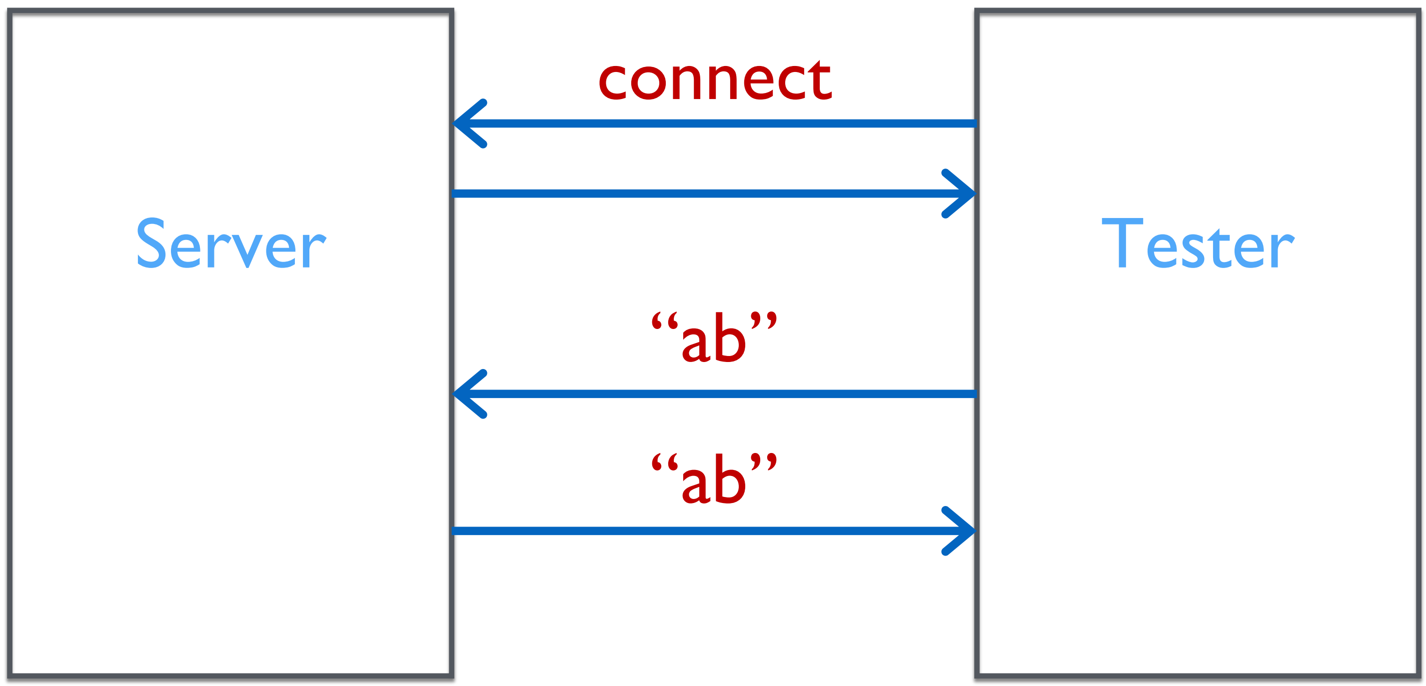
Too many metalanguages!

- Network-level HTTP spec
  - Acceptance tester (functional program)
- Web server implementation
  - CompCert “observation traces”
- VST C verification tool
  - Hoare triples in separation logic
- CertiKOS
  - “Layer interfaces”


Acceptance Tester
Written in Gallina
over bytestream API
Web Server
Written in C
over POSIX API
"zipper"
OK
Fail
Want to "zip them together" and show that the composite system does not reach a Fail state
But:
• C != Gallina
• Posix != bytestreams


Acceptance Tester
Written in Gallina over bytestream API
Monadic semantics
bytestream-level Interaction Tree with failures
network-level ITree
"zipper"
Skeleton ITree with *just* failures
Networking semantics
Web Server
Written in C over POSIX API
Hoare axioms for Posix calls
Verifiable C
Web Server
Written in Gallina over POSIX API
Monadic semantics
Posix-level ITree

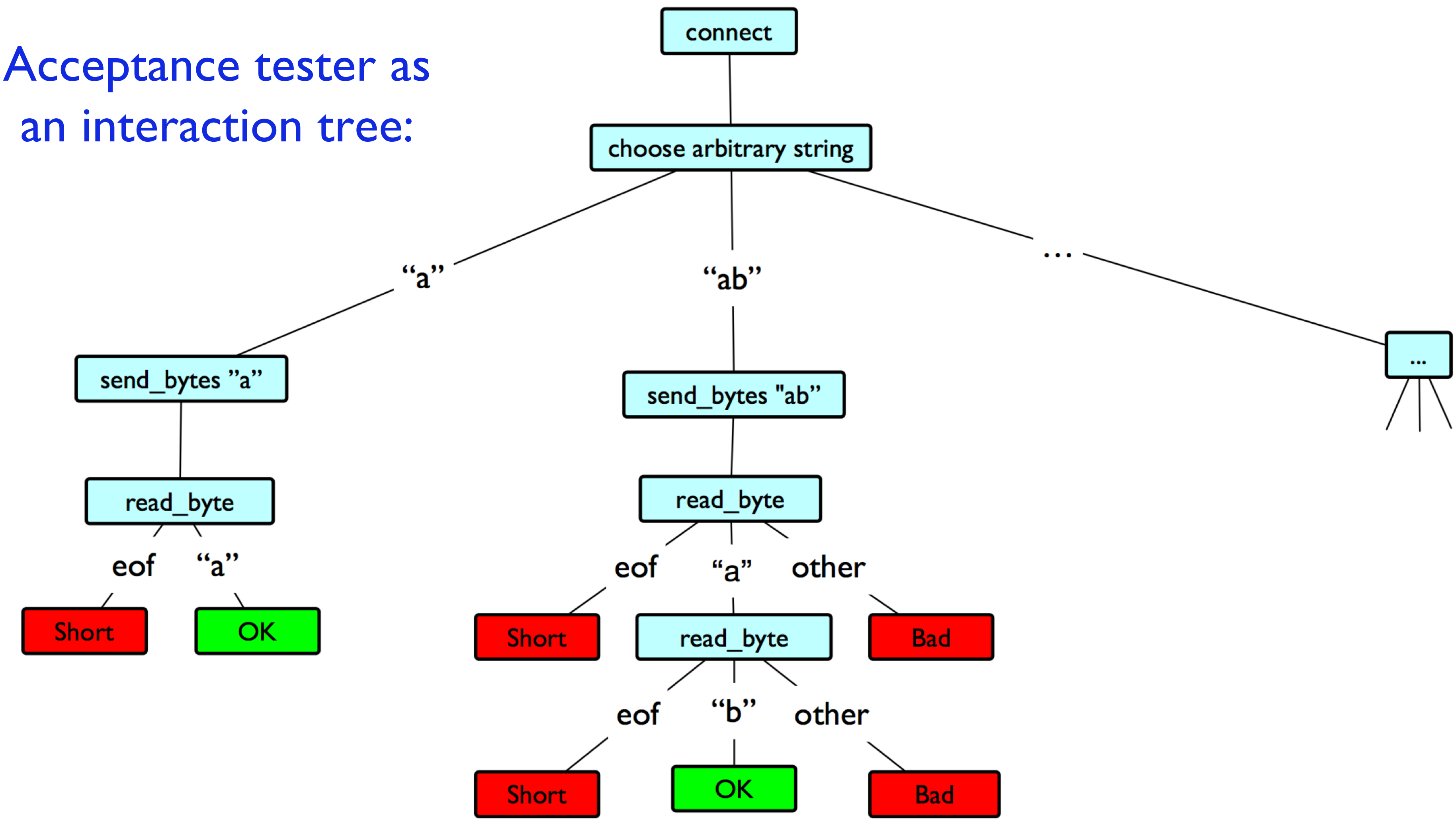
An “Echo Server”



Acceptance Tester

```
Definition echo_test :=
  c <- open_conn;;                  (* Open connection *)
  bytes <- arbitrary;;              (* Choose a string to send *)
  send_bytes c bytes;;              (* Send string *)
  for_each bytes (fun b =>          (* Check response byte by byte *)
    ob <- read_byte c ;;
    match ob with
    | None =>                        fail "Short"
    | Some b' => when (b != b') (fail "Bad")
    end).
```

Acceptance tester as an interaction tree:



More formally…

```
CoInductive M (Event : Type -> Type) X :=
| Ret (x:X)
| Tau (k: M Event X).
| Vis {Y: Type} (e : Event Y) (k : Y -> M Event X)
```

An `M E X` is the denotation of a program as a possibly infinite (coinductive) tree, parameterized over a type `Event` of observable events where:

- leaves correspond to final results labeled with `X`,
- internal nodes node are either
  - internal events (labeled `Tau`), or
  - observable events (labeled `Vis`, with a child for every element of the event's result type `Y`).

Network events

```
Inductive networkE : Type -> Type :=
| OpenConn : networkE connection
| CloseConn : connection -> networkE unit
| ReadByte : connection -> networkE (option byte)
| WriteByte : connection -> byte -> networkE unit.

Definition read_byte conn : M networkE (option byte) :=
  Vis (ReadByte conn) Ret.
```

Posix socket events

```
Inductive SocketAPI : Type -> Type :=
| Socket (domain : Z) (type : Z) (protocol : Z): SocketAPI (SocketError + sockfd)
| Close (fd : sockfd): SocketAPI (SocketError + unit)
| BindAndListen (fd : sockfd) : SocketAPI (SocketError + unit)
| Accept (fd : sockfd) : SocketAPI (SocketError + sockfd)
| Recv (fd : sockfd) (num_bytes : Z): SocketAPI (SocketError + string)
| Send (fd : sockfd) (msg : string): SocketAPI (SocketError + unit)
| Select (read_set : list sockfd): SocketAPI (SocketError + list sockfd).
```

Failure events

```
Definition failureE : Type -> Type :=
| Fail : string -> failureE void.

Definition fail reason : M failureE X :=
  Vis (Fail reason) ...
```

Nondeterminism events

```
Inductive arbitraryE : Type -> Type :=
| Arb : forall `{Show X} `{Arbitrary X}, arbitraryE X.
```

Status

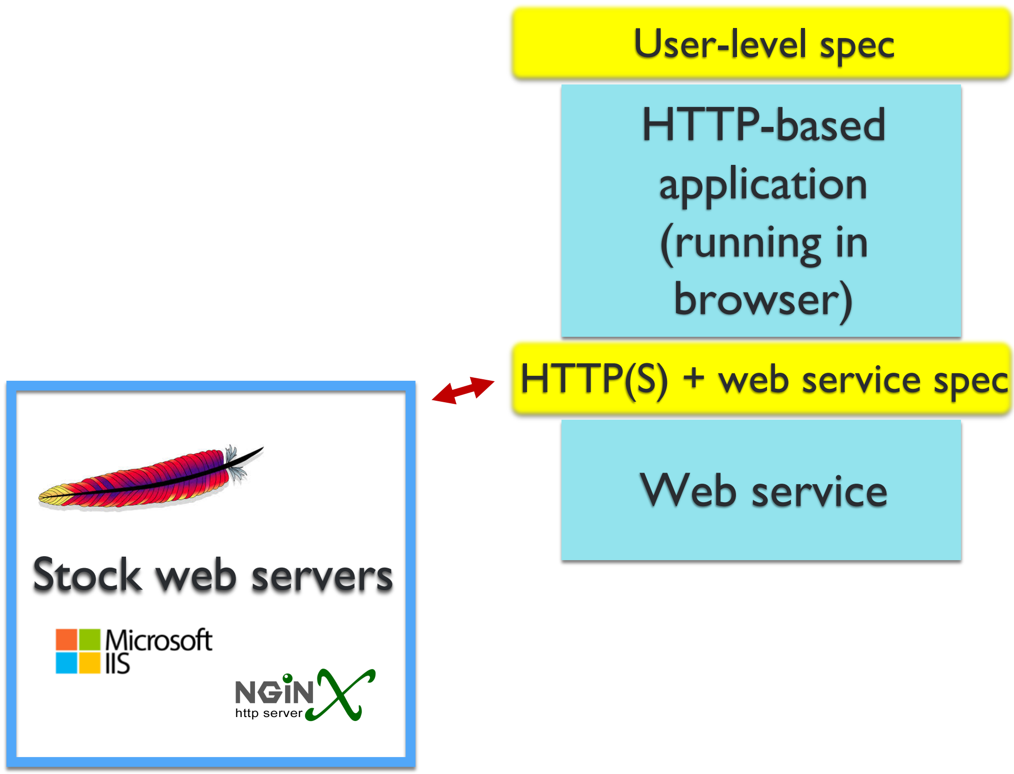
- “Echo server” correctness proof almost complete

Next steps

  - Prove that CertiKOS implementation of POSIX socket API satisfies the axioms
  - Scale proofs up to web server…

Component	Approximate LOC
Common: axioms for socket API	500
C code for echo server	**140**
Interaction tree for echo server	100
Hoare triples for functions in echo server	200
VST proofs of C-to-OS-level-spec	400
Coq proofs of OS-level-to-network-level	1000-2000 ?
Total	**1500-2500ish**
C code for web server	**2880**
Interaction tree for web server	2000?
Hoare triples for web server	4000?
VST proofs of C-to-OS-level-spec	8000?
Coq proofs of OS-level-to-network-level	20-40k?
Total	**30-50k ???**

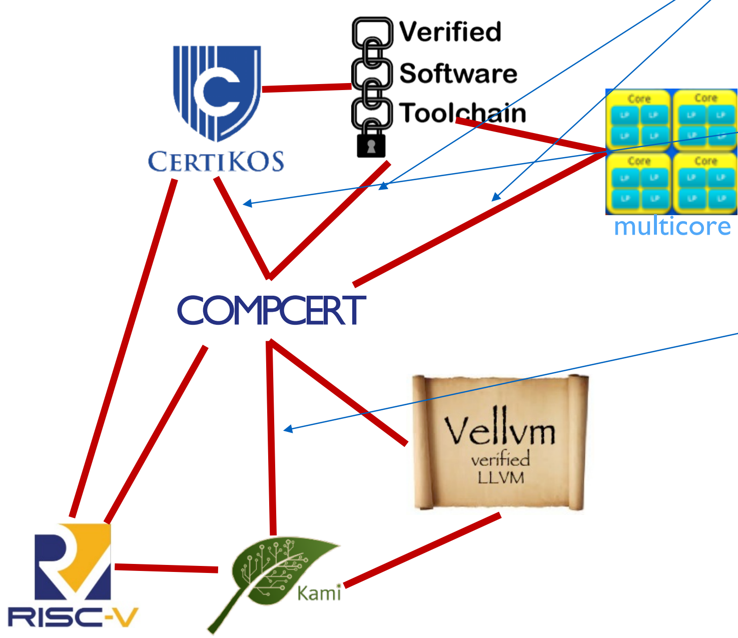
Challenge:
Exercising the HTTP specification
from both sides


User-level spec
HTTP-based application (running in browser)
HTTP(S) + web service spec
Web service
Stock web servers
Microsoft IIS
NGiNX
http server

Challenge:
Upgrading CompCert

Present-day CompCert is proved correct only for single-module, single-thread (sequential) programs; and only down to assembly language (not machine language); and only down to a block-structured memory model, not the flat address space of a real ISA.

Ongoing Work



Specifying and proving that CompCert is correct on **shared-memory concurrent** programs.

New semantic approaches to **separate compilation**

Assembly-to-machine-language and **structured-memory-model-to-flat-memory-model** specifications and proofs

Join us!

Thank you!

(any (more) questions?)

Teaching materials

Technical workshops
(next one @ PLDI 2018)

Summer schools

visitors program

PhD and postdoc positions

Visit deepspec.org to see what's happening and join our mailing list