The Science of Deep Specification

Benjamin C. Pierce
University of Pennsylvania

POST / ETAPS
April, 2018
Toward a 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)
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.
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
A (Very) Simple Formal Specification

Fixpoint \text{compile} (e : \text{exp}) : \text{list instr} :=
\begin{align*}
\text{match } e \text{ with} \\
| \text{Num } n &\Rightarrow \\text{[PUSH } n]\text{ ]} \\
| \text{Plus } e_1 e_2 &\Rightarrow \text{compile } e_1 ++ \text{compile } e_2 ++ \text{[PLUS]} \\
| \text{Minus } e_1 e_2 &\Rightarrow \text{compile } e_1 ++ \text{compile } e_2 ++ \text{[MINUS]} \\
| \text{Mult } e_1 e_2 &\Rightarrow \text{compile } e_1 ++ \text{compile } e_1 ++ \text{[MUL}T] \\
\end{align*}
\text{end.}
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) (Num 2))) [PUSH 2; PUSH 2; PLUS]).
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.

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

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

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 evaluating 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.
Specification-Based Testing

Random

Concolic

Enumerative

etc.
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

Haskell
QuickCheck
[Claessen&Hughes]
QuickChick compiles correctly.

Counterexample found after 4 tests and 8 shrinks:

```ocaml
Minus (Num 3) (Num 0)
```

```
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 compiles_correctly.

Counterexample found after 4 tests and 8 shrinks:

Minus (Num 3) (Num 0)

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.

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.

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
Lemma execute_app : \(\forall p1 \ p2 \ \text{stack}, \) 
\(\text{execute} \ \text{stack} \ (p1 ++ p2) = \text{execute} \ (\text{execute} \ \text{stack} \ p1) \ p2.\)

Lemma execute_eval_comm : \(\forall e \ \text{stack}, \) 
\(\text{execute} \ \text{stack} \ (\text{compile} \ e) = \text{eval} \ e :: \text{stack}.\)

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.

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

Simple automation

No 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.
Qed.

```
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.
```

Chipala automation

No automation
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.

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

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

...
nice story
does it scale?
- 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
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 certified 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.
• C verification framework based on higher-order separation logic in Coq
• Verified implementations of OpenSSL-HMAC and SHA-256
• working on additional crypto 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, …)
• 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!

Coq

Isabelle

Concrete Semantics
With Isabelle/HOL

(c,s) ⇒ t

Tobias Nipkow, Gerwin Klein

Certified Programming with Dependent Types
A Pragmatic Introduction to the Coq Proof Assistant

Adam Chlipala
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

Historical Cost of Computer Memory and Storage
Are we done?

Nope.
Lessons from CompCert

C language

CompCert
Compiler

PowerPC ISA

CertiKOS hypervisor kernel

OS client interface

Program Logic
Verifiable C System

C language

C language

C language

IBM’s CPU

PowerPC ISA

Transistors

Leroy

Shao

Appel

Sewell
Lessons from CompCert

OS client interface
- CertiKOS hypervisor kernel
- C language

Program Logic
- Verifiable C System
- C language

C language
- CompCert Compiler
- PowerPC ISA

PowerPC ISA
- IBM’s CPU
- Transistors

Authors:
- Shao
- Appel
- Sewell
- Leroy
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

Two-sided specifications
“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”
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

Core Spec

Vellvm verified LLVM

CertiCoq

Quick Chick

One whole

deep spec server
The DeepSpec Web Server

• 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
Goal: A “single QED” encompassing the whole stack

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
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
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”
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

Web Server

Written in C over POSIX API

Monadic semantics

Hoare axioms for Posix calls

Verifiable C

Web Server

Written in Gallina over POSIX API

Monadic semantics

bytestream-level Interaction

Tree with failures

“zipper”

Skeleton ITree with just failures

Networking semantics

network-level ITree

Posix-level ITree
An “Echo Server”

Server

connect

Tester

“ab”

“ab”
Acceptance Tester

Definition echo_test :=
  c <- open_conn;;
  bytes <- arbitrary;;
  send_bytes c bytes;;
  for_each bytes (fun b =>
    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...

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

- **leaves** correspond to final **results** labeled with \( X \),
- **internal nodes** node are either
  - **internal events** (labeled \( \text{Tau} \)), or
  - **observable events** (labeled \( \text{Vis} \), 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…
<table>
<thead>
<tr>
<th>Component</th>
<th>Approximate LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common: axioms for socket API</td>
<td>500</td>
</tr>
<tr>
<td>C code for echo server</td>
<td>140</td>
</tr>
<tr>
<td>Interaction tree for echo server</td>
<td>100</td>
</tr>
<tr>
<td>Hoare triples for functions in echo server</td>
<td>200</td>
</tr>
<tr>
<td>VST proofs of C-to-OS-level-spec</td>
<td>400</td>
</tr>
<tr>
<td>Coq proofs of OS-level-to-network-level</td>
<td>1000-2000 ?</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1500-2500ish</strong></td>
</tr>
<tr>
<td>C code for web server</td>
<td>2880</td>
</tr>
<tr>
<td>Interaction tree for web server</td>
<td>2000?</td>
</tr>
<tr>
<td>Hoare triples for web server</td>
<td>4000?</td>
</tr>
<tr>
<td>VST proofs of C-to-OS-level-spec</td>
<td>8000?</td>
</tr>
<tr>
<td>Coq proofs of OS-level-to-network-level</td>
<td>20-40k?</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>30-50k ??</strong></td>
</tr>
</tbody>
</table>
Challenge:
Exercising the HTTP specification from both sides
HTTP(S) + web service spec

Web service

HTTP-based application (running in browser)

User-level spec

Stock web servers
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!

Teaching materials
Summer schools
PhD and postdoc positions

Technical workshops
(next one @ PLDI 2018)

Thank you!
(any (more) questions?)

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