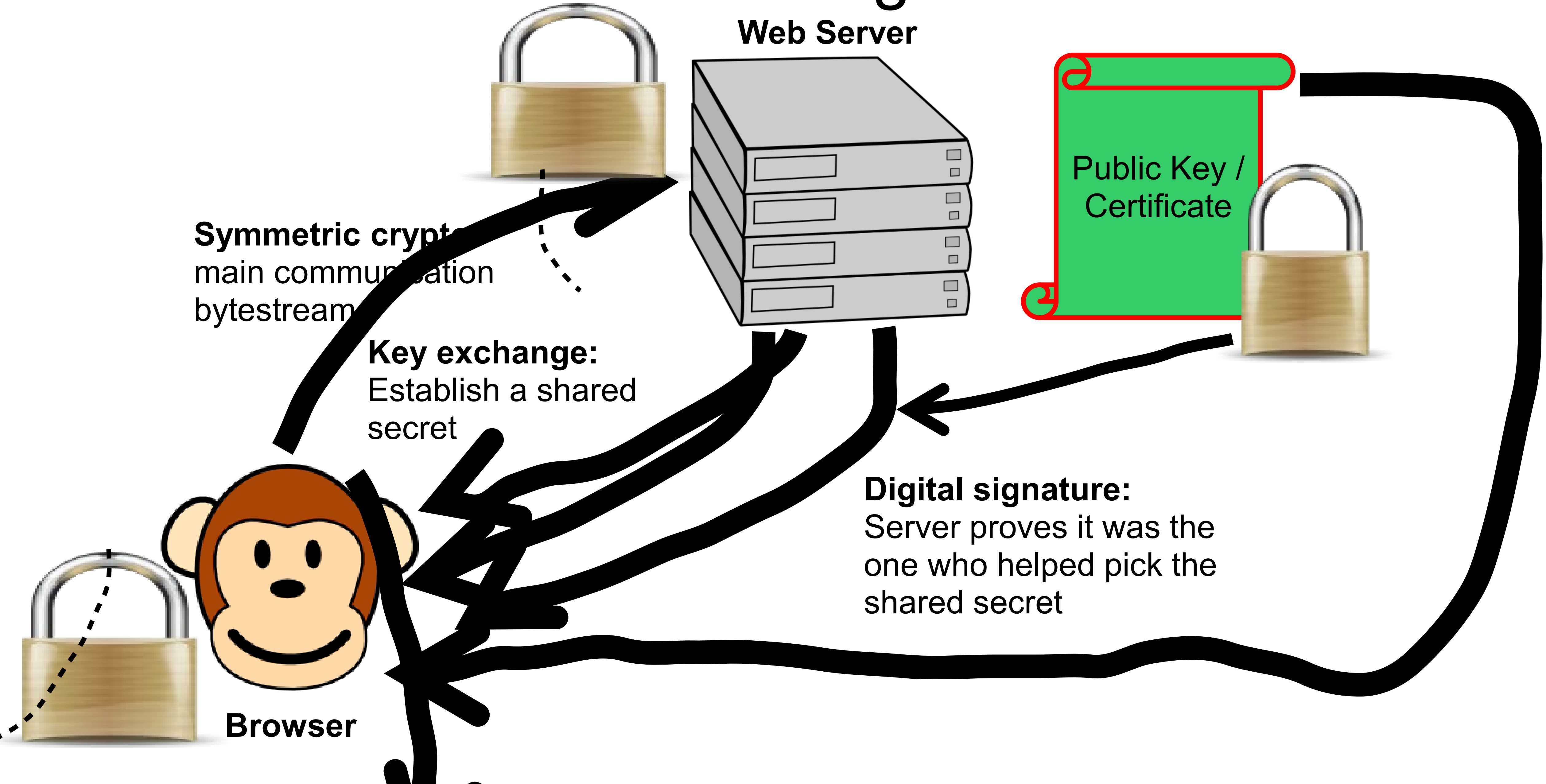
Correct-by-Construction Cryptography
Without Performance Compromises

Adam Chlipala, MIT CSAIL
NUS Computer Science Research Week
January 2022

Joint work with:
Joonwon Choi, Andres Erbsen, Jason Gross, Jade Philipoom, Robert Sloan, Clark Wood

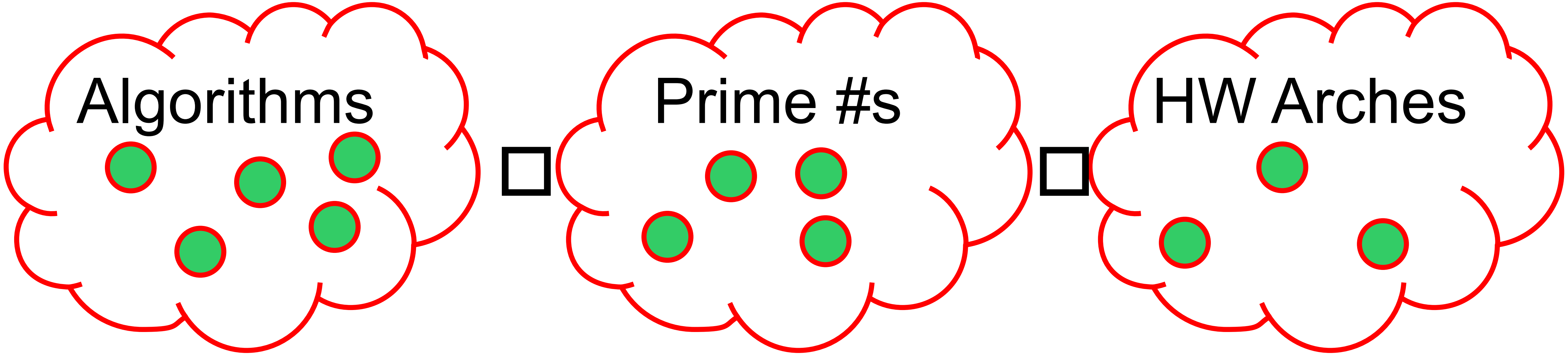
Web Browsing with SSL



About the First Two Stages (Public-Key Crypto)

- Public-key stages only run once per session, but, with many small HTTPS connections common in practice, their performance is still important.
- Balancing correctness and performance is also more challenging for the public-key algorithms.

But the experts know how to do all this, right?

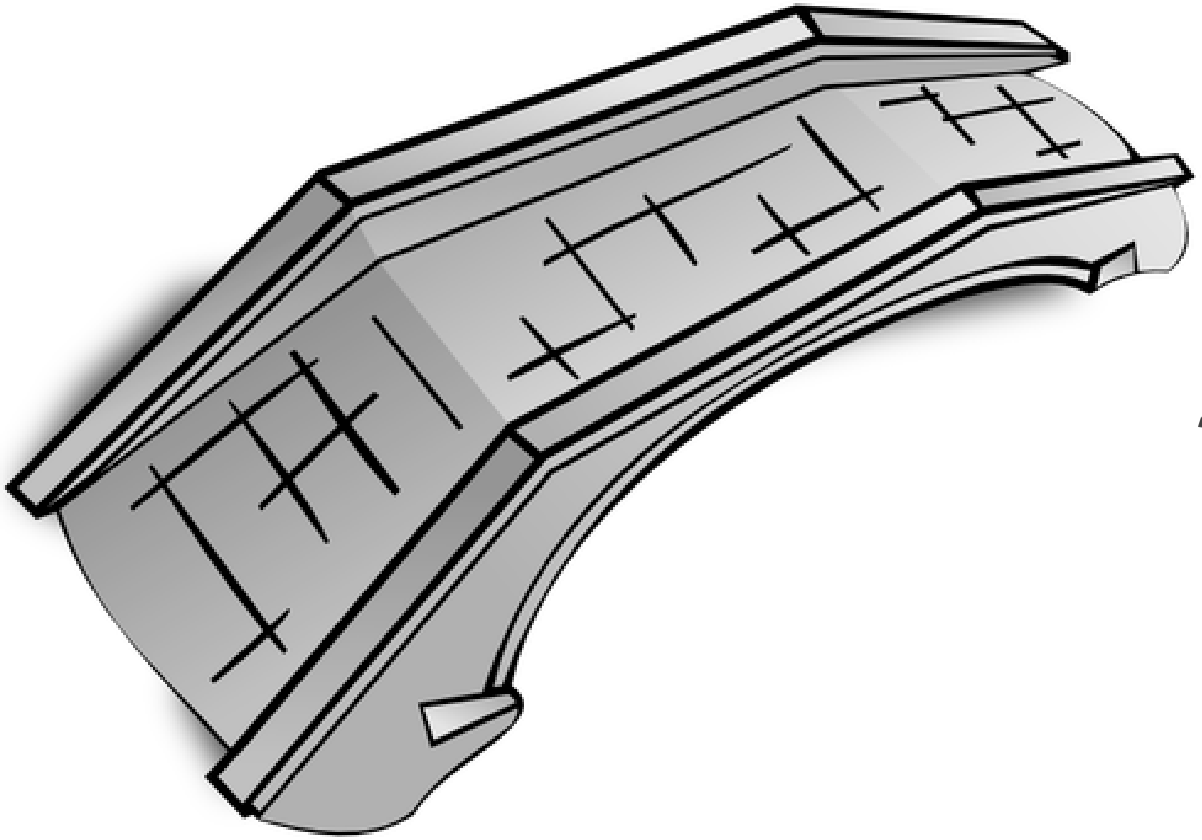


Labor-intensive adaptation, with each combination taking significant expert effort.

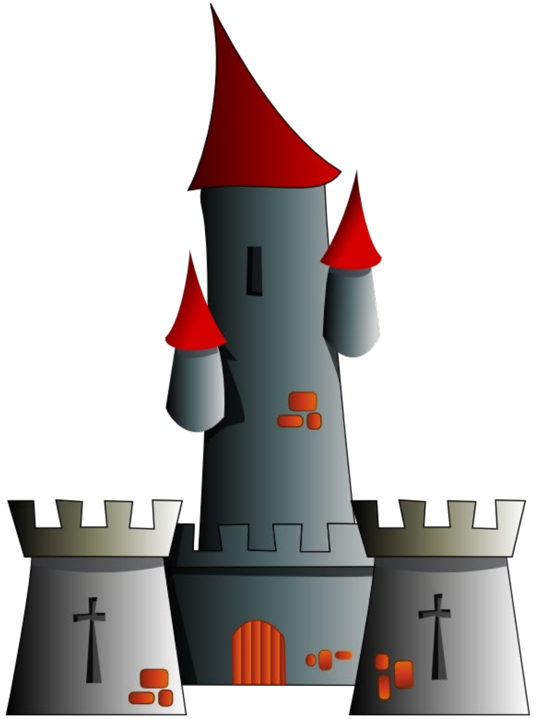
We introduced *Fiat Cryptography*.

- An automatic generator for this kind of code,
- with correctness proofs in the Coq theorem prover.
- Adopted for small but important parts of TLS implementations in both Chrome and Firefox, plus a number of blockchain systems, etc.

The Real World
(maintainers of OpenSSL,
etc., live here)



The Ivory Tower
(formal-verification
experts live here)



Outline

- Catching up: formal verification in the 21st century
- More specific project motivation
- Classic Fiat Cryptography
- Towards correct-by-construction cryptographic appliances

Catching up: formal verification in the 21st century

Debugging: The Secret Essence of Programming

"By June 1949 people had begun to realize that it was not so easy to get programs right as at one time appeared.

[…] the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs."

Maurice Wilkes, *Memoirs of a Computer Pioneer*, MIT Press, 1985, p. 145.

"EDSAC (9)". Licensed under CC BY 2.0 via Commons - https://commons.wikimedia.org/wiki/File:EDSAC_(9).jpg#/media/File:EDSAC_(9).jpg

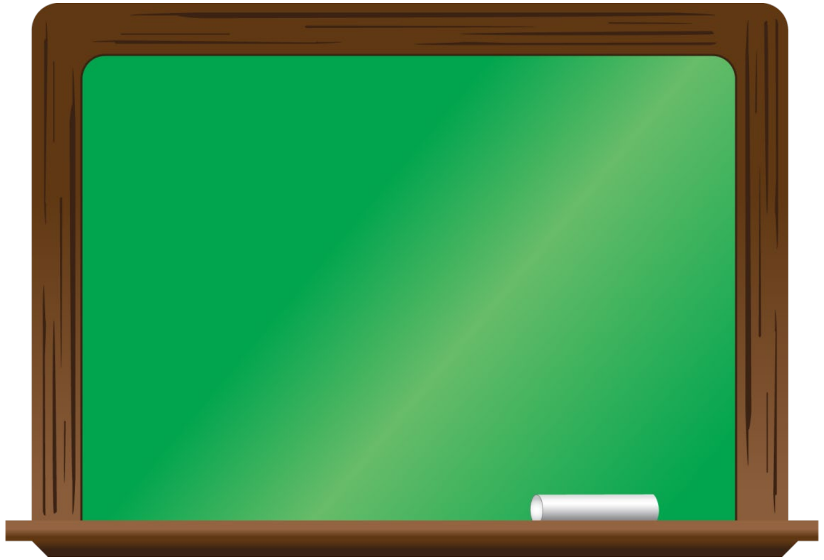
Crucial Substitutions

Debugging exploring concrete executions	→	**Proving** exploring symbolic arguments
Testing describing concrete scenarios	→	**Specifying** describing general requirements
Auditing code algorithms in detail	→	**Auditing specs** functionality without optimizations

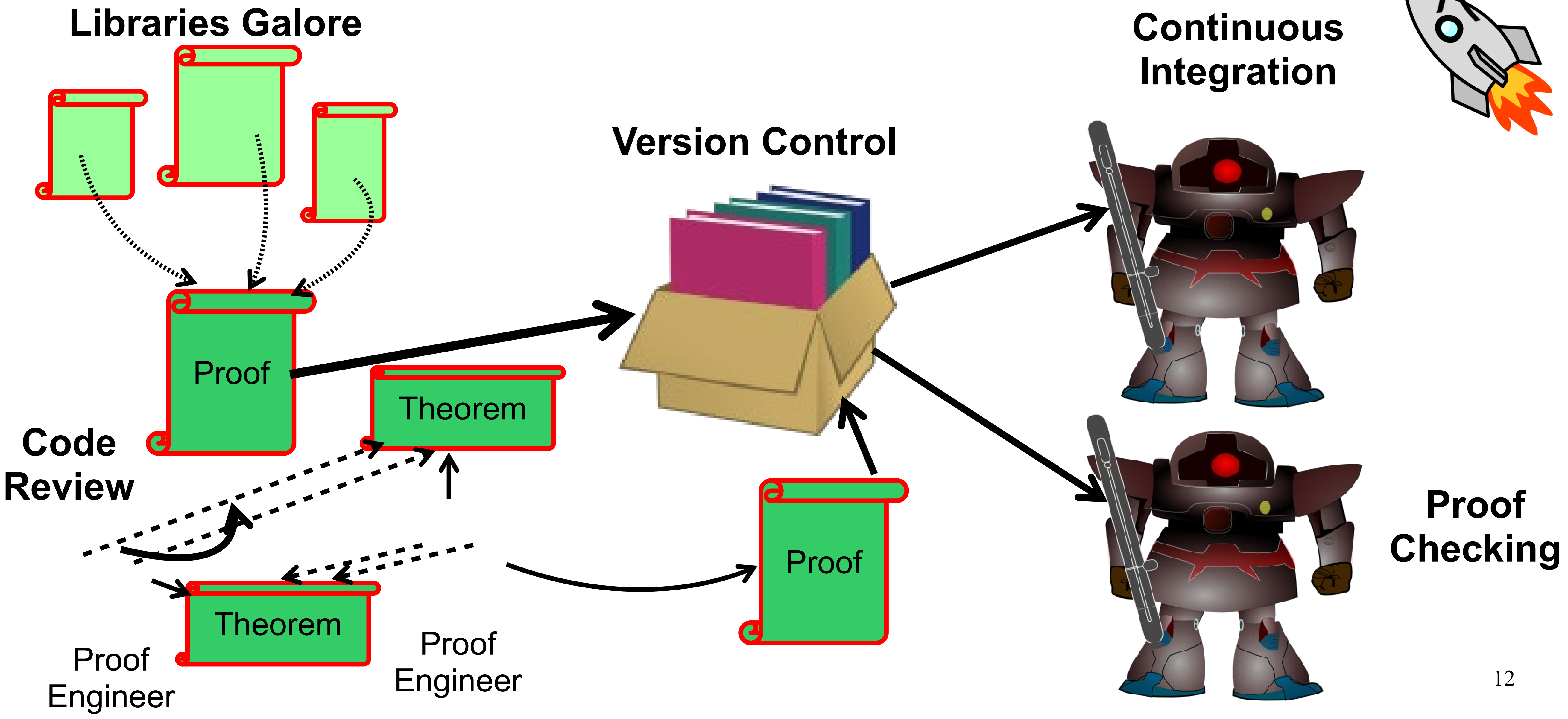
Q: Aren't These Proofs Too Boring for Mortals?

It is argued that formal verifications of programs, no matter how obtained, will not play the same key role in the development of computer science and software engineering as proofs do in mathematics. Furthermore the absence of continuity, the inevitability of change, and the complexity of specification of significantly many real programs make the formal verification process difficult to justify and manage.

– De Millo, Lipton, and Perlis,
"Social Processes and Proofs of
Theorems and Programs," CACM, 1979



The Proof Workflow of the Future



Proof Assistant

noun: a software package essentially providing an

integrated development environment (IDE)

for *stating and proving mathematical theorems*

where writing proofs takes human effort

but **checking proofs is automatic**

The Most Popular Proof Assistants

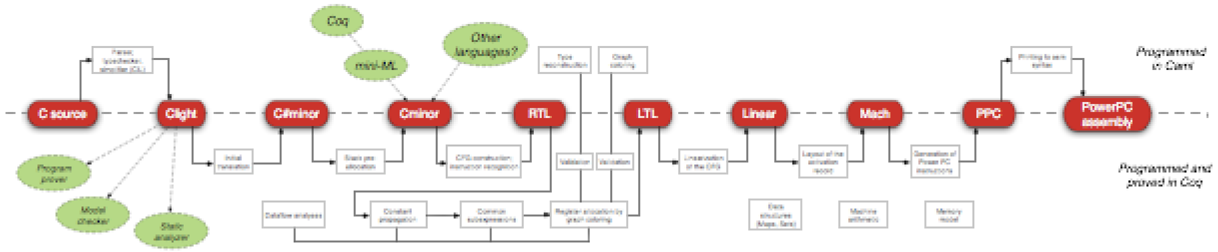
Isabelle/HOL

One well-known application:

Verified microkernel OS

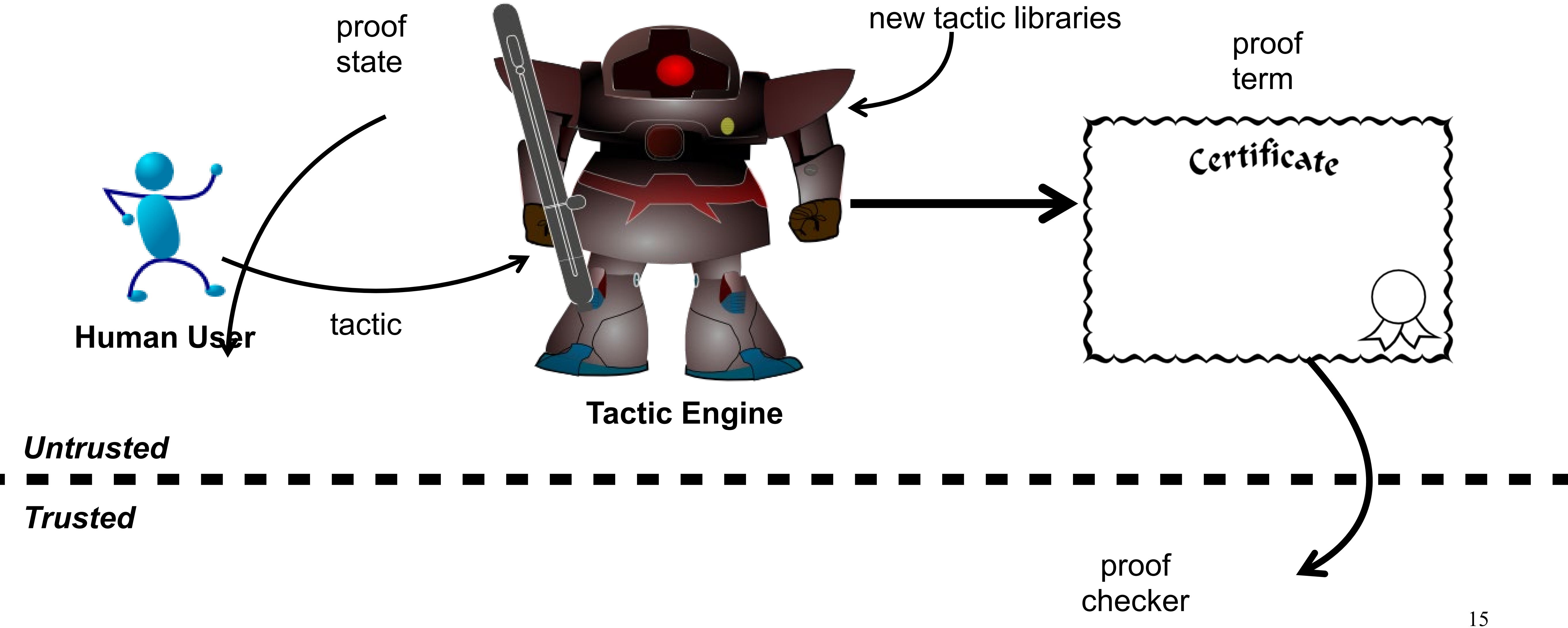
Coq

One well-known application:
CompCert



Verified C compiler

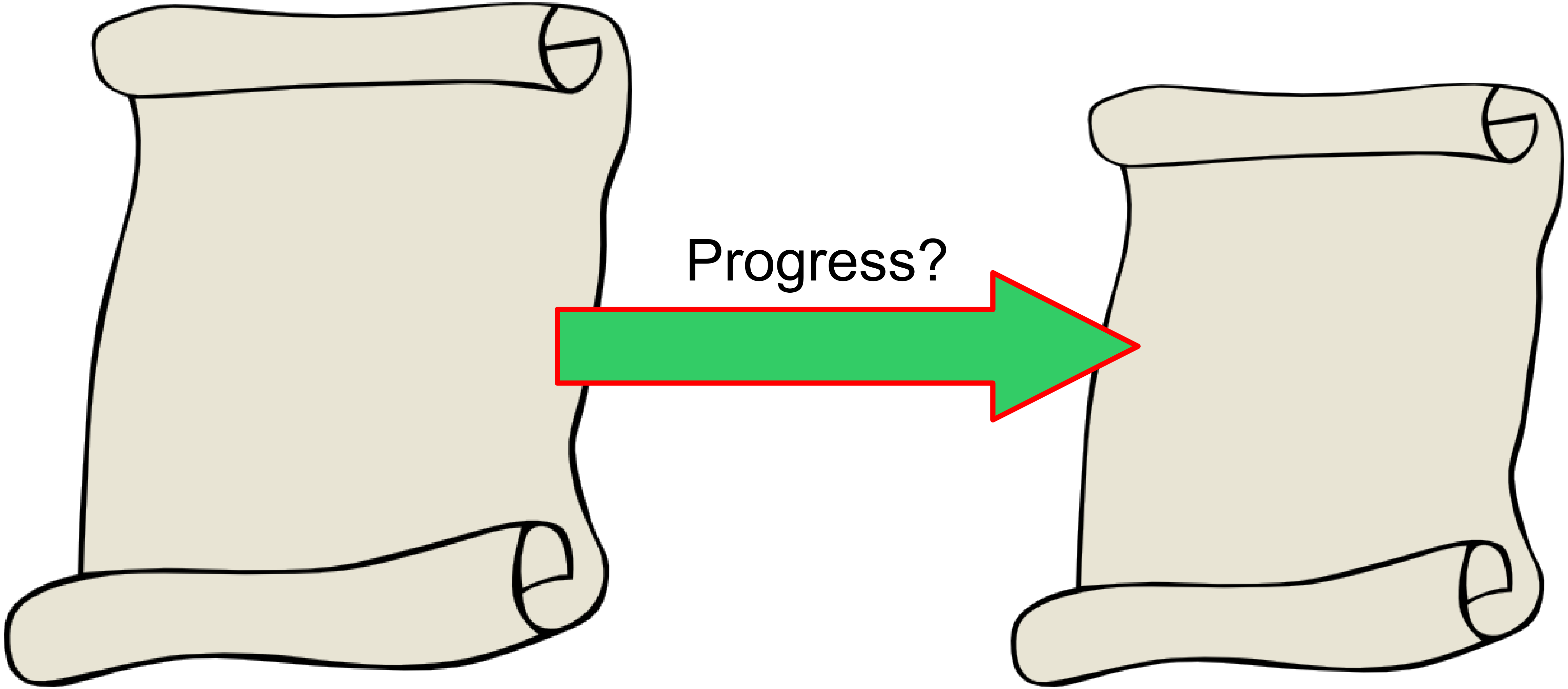
Why Should the [Machine|Human] Trust the [Human|Machine]?



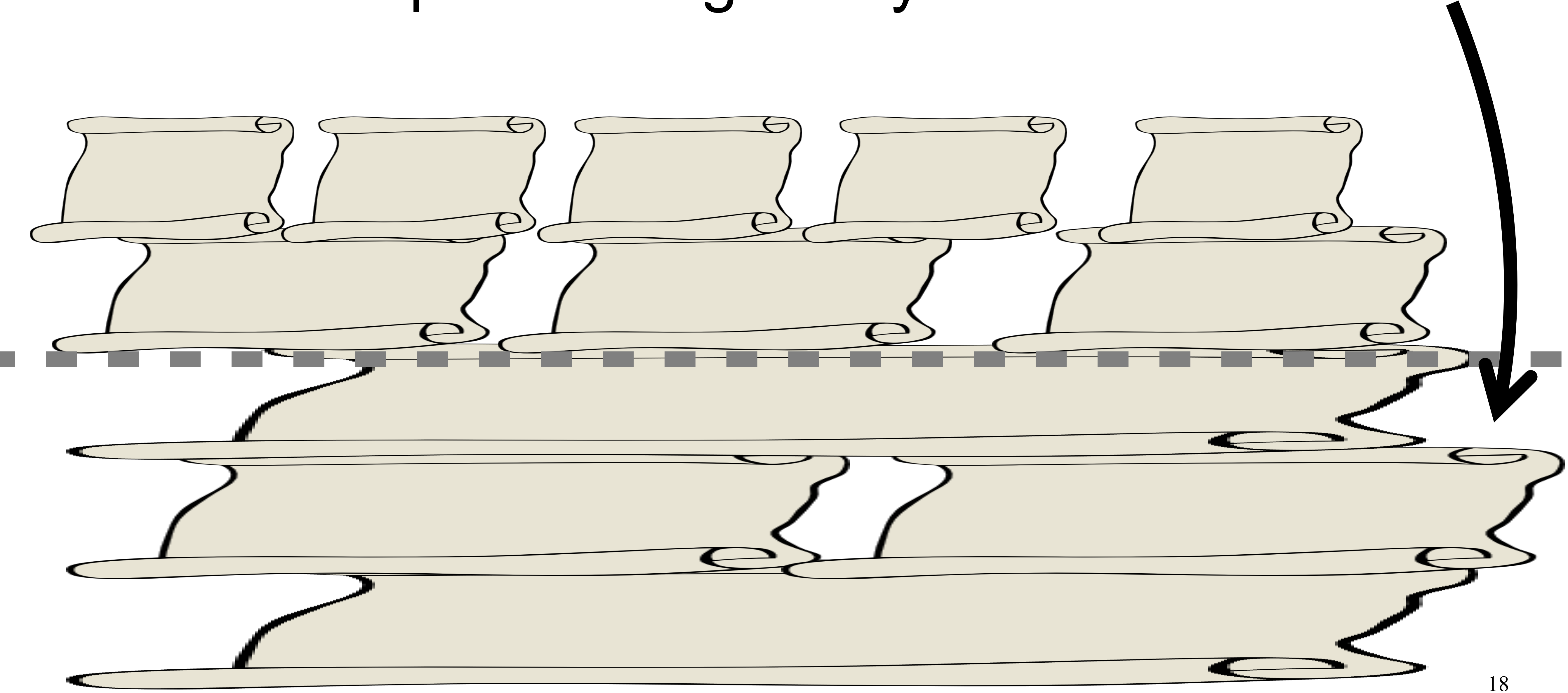
Demo

Some simple proofs in Coq

Q: Isn't It (About) As Hard to Get Specs Right?



A: Focus Spec-Writing on Systems Infrastructure



An Approximate Truth About Software

Spec

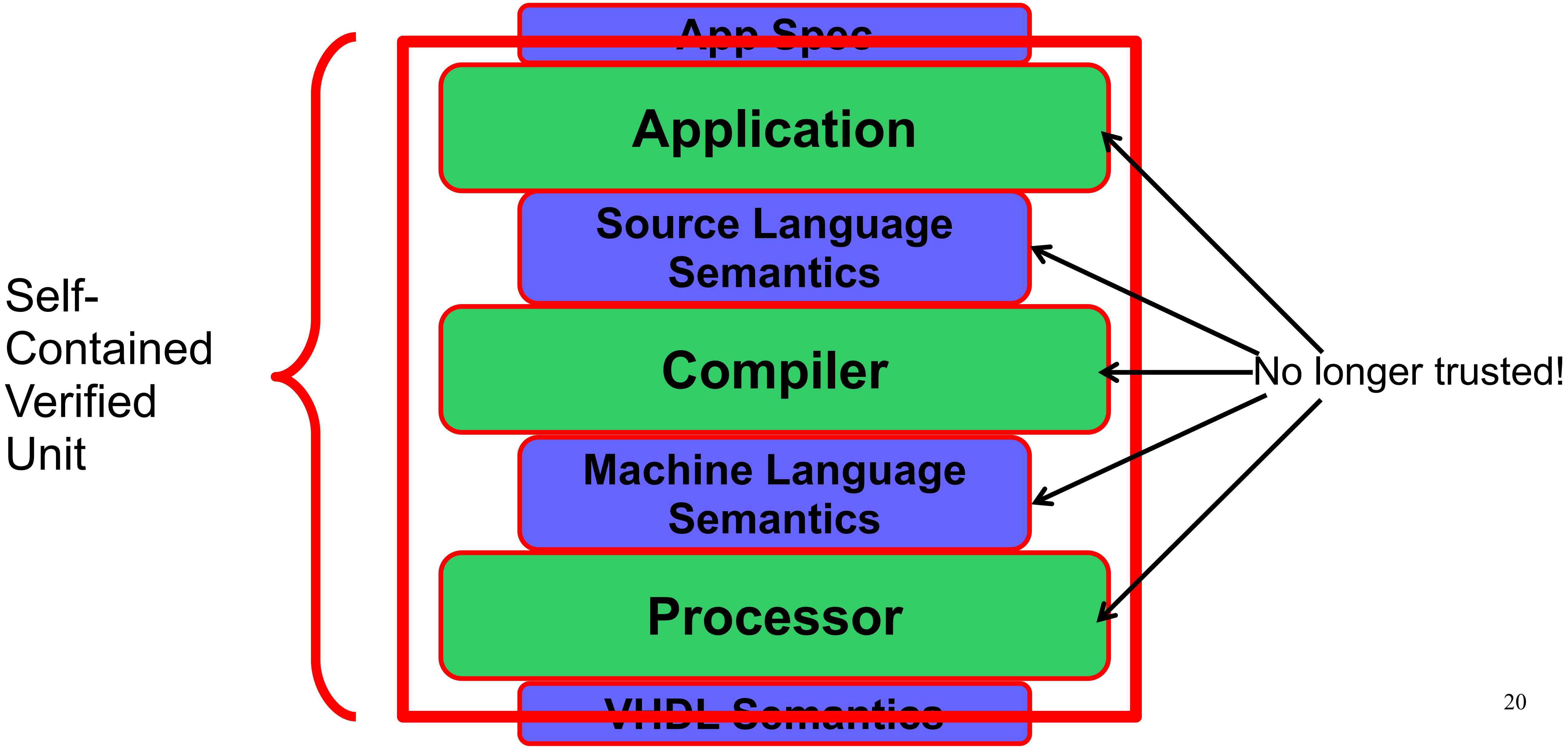
+

Optimizations

=

Implementation

Q: Aren't Those Specs *Still* Hard to Get Right?

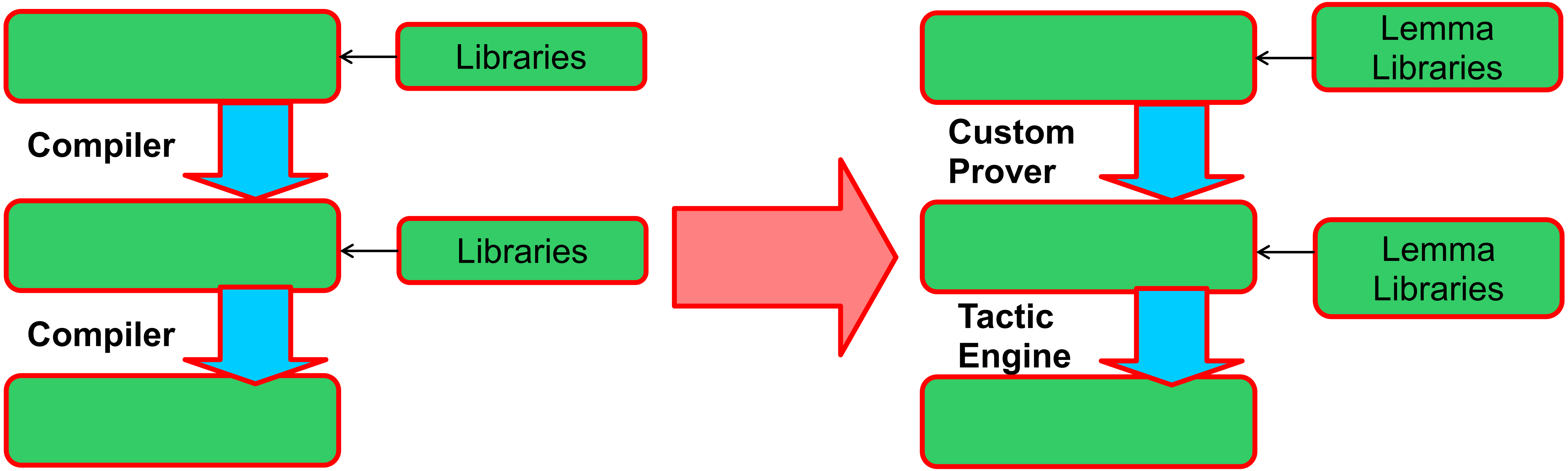


Old vs. New

Old	**New**
System-integration tests *and* unit tests, since combined state space grows **exponentially** as we compose pieces	System-integration theorems imply proper functioning of all components.
Careful code review of all components, since a corner-case bug in any of them can wreck the whole system	Careful code review only of **externally facing specs**

Q: Aren't the Proofs Huge and Unwieldy?

Well, aren't machine-code programs huge, too?



Motivation: correct-by-construction crypto

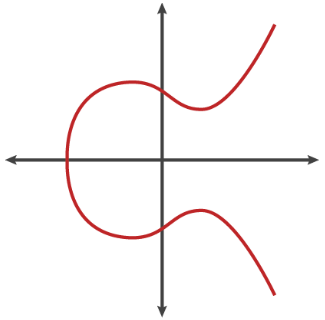
Correct-by-Construction Cryptography

Abstract security property

"Knowledge of the secret key is needed to produce a signature in polynomial time."

protocol verification

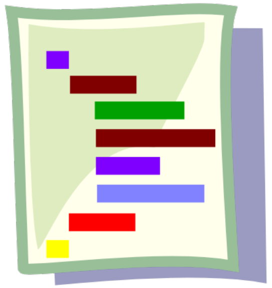
Mathematical algorithm



$y^2 = x^3 – x + 1$

implementation synthesis

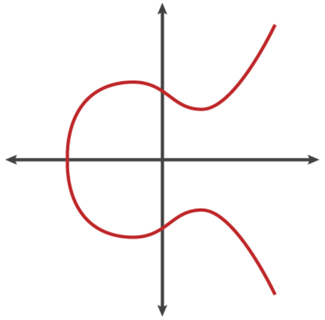
Low-level code



specialized assembly code

Correct-by-Construction Cryptography

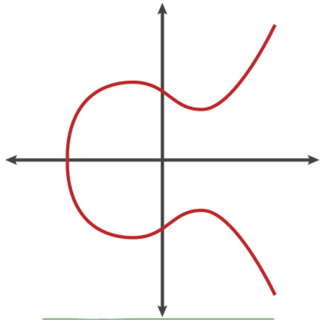
Mathematical algorithm



point = (x, y)

classic verification of functional programs

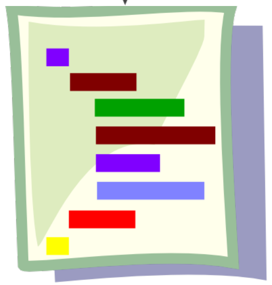
Optimized point format



point = (x, y, z, t)

classic verification of functional programs

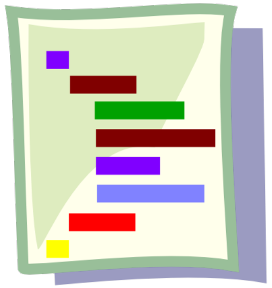
High-level modular arithmetic



$x = x_0, x_1, \ldots, x_n$
(mathematical integers)

compile-time code specialization

Low-level code



specialized low-level code
(assumes fixed set of integer sizes)

compiler verification

Generated Code

Squaring a number (64-bit)

```
λ '(x7, x8, x6, x4, x2)%core,
uint64_t x9 = x2 * 0x2;
uint64_t x10 = x4 * 0x2;
uint64_t x11 = x6 * 0x2 * 0x13;
uint64_t x12 = x7 * 0x13;
uint64_t x13 = x12 * 0x2;
uint128_t x14 = (uint128_t) x2 * x2 + (uint128_t) x13 * x4 + (uint128_t) x11 * x8;
uint128_t x15 = (uint128_t) x9 * x4 + (uint128_t) x13 * x6 + (uint128_t) x8 * (x8 * 0x13);
uint128_t x16 = (uint128_t) x9 * x6 + (uint128_t) x4 * x4 + (uint128_t) x13 * x8;
uint128_t x17 = (uint128_t) x9 * x8 + (uint128_t) x10 * x6 + (uint128_t) x7 * x12;
uint128_t x18 = (uint128_t) x9 * x7 + (uint128_t) x10 * x8 + (uint128_t) x6 * x6;
uint64_t x19 = (uint64_t) (x14 >> 0x33);
uint64_t x20 = (uint64_t) x14 & 0x7ffffffffffff;
uint128_t x21 = x19 + x15;
uint64_t x22 = (uint64_t) (x21 >> 0x33);
uint64_t x23 = (uint64_t) x21 & 0x7ffffffffffff;
uint128_t x24 = x22 + x16;
uint64_t x25 = (uint64_t) (x24 >> 0x33);
uint64_t x26 = (uint64_t) x24 & 0x7ffffffffffff;
uint128_t x27 = x25 + x17;
uint64_t x28 = (uint64_t) (x27 >> 0x33);
uint64_t x29 = (uint64_t) x27 & 0x7ffffffffffff;
uint128_t x30 = x28 + x18;
uint64_t x31 = (uint64_t) (x30 >> 0x33);
uint64_t x32 = (uint64_t) x30 & 0x7ffffffffffff;
uint64_t x33 = x20 + 0x13 * x31;
uint64_t x34 = x33 >> 0x33;
uint64_t x35 = x33 & 0x7ffffffffffff;
uint64_t x36 = x34 + x23;
uint64_t x37 = x36 >> 0x33;
uint64_t x38 = x36 & 0x7ffffffffffff;
return (Return x32, Return x29, x37 + x26, Return x38, Return x35))
```

Squaring a number (32-bit)

```
λ '(x17, x18, x16, x14, x12, x10, x8, x6, x4, x2)%core,
uint64_t x19 = (uint64_t) x2 * x2;
uint64_t x20 = (uint64_t) (0x2 * x2) * x4;
uint64_t x21 = 0x2 * ((uint64_t) x4 * x4 + (uint64_t) x2 * x6);
uint64_t x22 = 0x2 * ((uint64_t) x4 * x6 + (uint64_t) x2 * x8);
uint64_t x23 = (uint64_t) x6 * x6 + (uint64_t) (0x4 * x4) * x8 + (uint64_t) (0x2 * x2) * x10;
uint64_t x24 = 0x2 * ((uint64_t) x6 * x8 + (uint64_t) x4 * x10 + (uint64_t) x2 * x12);
uint64_t x25 = 0x2 * ((uint64_t) x8 * x8 + (uint64_t) x6 * x10 + (uint64_t) x2 * x14 + (uint64_t) (0x2 * x4) * x12);
uint64_t x26 = 0x2 * ((uint64_t) x8 * x10 + (uint64_t) x6 * x12 + (uint64_t) x4 * x14 + (uint64_t) x2 * x16);
uint64_t x27 = (uint64_t) x10 * x10 + 0x2 * ((uint64_t) x6 * x14 + (uint64_t) x2 * x18 + 0x2 * ((uint64_t) x4 * x16 + (uint64_t) x8 * x12));
uint64_t x28 = 0x2 * ((uint64_t) x10 * x12 + (uint64_t) x8 * x14 + (uint64_t) x6 * x16 + (uint64_t) x4 * x18 + (uint64_t) x2 * x17);
uint64_t x29 = 0x2 * ((uint64_t) x12 * x12 + (uint64_t) x10 * x14 + (uint64_t) x6 * x18 + 0x2 * ((uint64_t) x8 * x16 + (uint64_t) x4 * x17));
uint64_t x30 = 0x2 * ((uint64_t) x12 * x14 + (uint64_t) x10 * x16 + (uint64_t) x8 * x18 + (uint64_t) x6 * x17);
uint64_t x31 = (uint64_t) x14 * x14 + 0x2 * ((uint64_t) x10 * x18 + 0x2 * ((uint64_t) x12 * x16 + (uint64_t) x8 * x17));
uint64_t x32 = 0x2 * ((uint64_t) x14 * x16 + (uint64_t) x12 * x18 + (uint64_t) x10 * x17);
uint64_t x33 = 0x2 * ((uint64_t) x16 * x16 + (uint64_t) x14 * x18 + (uint64_t) (0x2 * x12) * x17);
uint64_t x34 = 0x2 * ((uint64_t) x16 * x18 + (uint64_t) x14 * x17);
uint64_t x35 = (uint64_t) x18 * x18 + (uint64_t) (0x4 * x16) * x17;
uint64_t x36 = (uint64_t) (0x2 * x18) * x17;
uint64_t x37 = (uint64_t) (0x2 * x17) * x17;
uint64_t x38 = x27 + x37 << 0x4;
uint64_t x39 = x38 + x37 << 0x1;
uint64_t x40 = x39 + x37;
uint64_t x41 = x26 + x36 << 0x4;
uint64_t x42 = x41 + x36 << 0x1;
uint64_t x43 = x42 + x36;
uint64_t x44 = x25 + x35 << 0x4;
uint64_t x45 = x44 + x35 << 0x1;
uint64_t x46 = x45 + x35;
uint64_t x47 = x24 + x34 << 0x4;
uint64_t x48 = x47 + x34 << 0x1;
uint64_t x49 = x48 + x34;
uint64_t x50 = x23 + x33 << 0x4;
uint64_t x51 = x50 + x33 << 0x1;
uint64_t x52 = x51 + x33;
uint64_t x53 = x22 + x32 << 0x4;
uint64_t x54 = x53 + x32 << 0x1;
uint64_t x55 = x54 + x32;
uint64_t x56 = x21 + x31 << 0x4;
uint64_t x57 = x56 + x31 << 0x1;
uint64_t x58 = x57 + x31;
uint64_t x59 = x20 + x30 << 0x4;
uint64_t x60 = x59 + x30 << 0x1;
uint64_t x61 = x60 + x30;
uint64_t x62 = x19 + x29 << 0x4;
uint64_t x63 = x62 + x29 << 0x1;
uint64_t x64 = x63 + x29;
uint64_t x65 = x64 >> 0x1a;
uint32_t x66 = (uint32_t) x64 & 0x3ffffff;
uint64_t x67 = x65 + x61;
uint64_t x68 = x67 >> 0x19;
uint32_t x69 = (uint32_t) x67 & 0x1ffffff;
uint64_t x70 = x68 + x58;
uint64_t x71 = x70 >> 0x1a;
uint32_t x72 = (uint32_t) x70 & 0x3ffffff;
uint64_t x73 = x71 + x55;
uint64_t x74 = x73 >> 0x19;
uint32_t x75 = (uint32_t) x73 & 0x1ffffff;
uint64_t x76 = x74 + x52;
uint64_t x77 = x76 >> 0x1a;
uint32_t x78 = (uint32_t) x76 & 0x3ffffff;
uint64_t x79 = x77 + x49;
uint64_t x80 = x79 >> 0x19;
uint32_t x81 = (uint32_t) x79 & 0x1ffffff;
uint64_t x82 = x80 + x46;
uint32_t x83 = (uint32_t) (x82 >> 0x1a);
uint32_t x84 = (uint32_t) x82 & 0x3ffffff;
uint64_t x85 = x83 + x43;
uint32_t x86 = (uint32_t) (x85 >> 0x19);
uint32_t x87 = (uint32_t) x85 & 0x1ffffff;
uint64_t x88 = x86 + x40;
uint32_t x89 = (uint32_t) (x88 >> 0x1a);
uint32_t x90 = (uint32_t) x88 & 0x3ffffff;
uint64_t x91 = x89 + x28;
uint32_t x92 = (uint32_t) (x91 >> 0x19);
uint32_t x93 = (uint32_t) x91 & 0x1ffffff;
uint64_t x94 = x66 + (uint64_t) 0x13 * x92;
uint32_t x95 = (uint32_t) (x94 >> 0x1a);
uint32_t x96 = (uint32_t) x94 & 0x3ffffff;
uint32_t x97 = x95 + x69;
uint32_t x98 = x97 >> 0x19;
uint32_t x99 = x97 & 0x1ffffff;
return (Return x93, Return x90, Return x87, Return x84, Return x81, Return x78, Return x75, x98 + x72, Return x99, Return x96))
```

Surprising (?) Fact About Modular Arithmetic

Different prime moduli have dramatically different efficiency with best code on commodity processors.

$2^{255} - 19$ is a popular choice for relatively easy implementation.
General pattern: $2^k - c$, for $c \ll 2^k$. (Called *pseudo-Mersenne*.)
Example of a fast operation: *modular reduction*

$$
\begin{aligned}
t &= x + 2^k y \pmod{2^k - c} \quad \text{too big to fit below the modulus!} \\
&= x + (2^k - c + c)y \pmod{2^k - c} \\
&= x + \cancel{(2^k - c)y} + cy \pmod{2^k - c} \\
&= x + cy \pmod{2^k - c}
\end{aligned}
$$

Representing Numbers mod $2^{255} - 19$

t result of multiplying two numbers in the prime field, so **510 bits wide**

$= t_0\ t_1\ t_2\ t_3\ t_4\ t_5\ t_6\ t_7$ each "digit" fits in 64-bit register

$= (t_0 + 2^{64}\ t_1 + \ldots) + 2^{256}\ (t_4 + 2^{64}\ t_5 + \ldots)$

darn, that's 2^{256}, not 2^{255}, so we can't use that reduction trick!

However.... $51 \times 10 = 510$.

$t = (t_0 + 2^{51}\ t_1 + \ldots) + 2^{255}\ (t_5 + 2^{51}\ t_6 + \ldots)$

champion rep. on **64-bit processors**
(note: not using full bitwidth!)

Also.... $25.5 \times 2 = 51$.

$t = s_0 + 2^{25.5}\ s_1 + 2^{2 \times 25.5}\ s_2 + 2^{3 \times 25.5}\ s_3 + \ldots$

$t = s_0 + 2^{26}\ s_1 + 2^{51}\ s_2 + 2^{77}\ s_3 + \ldots$

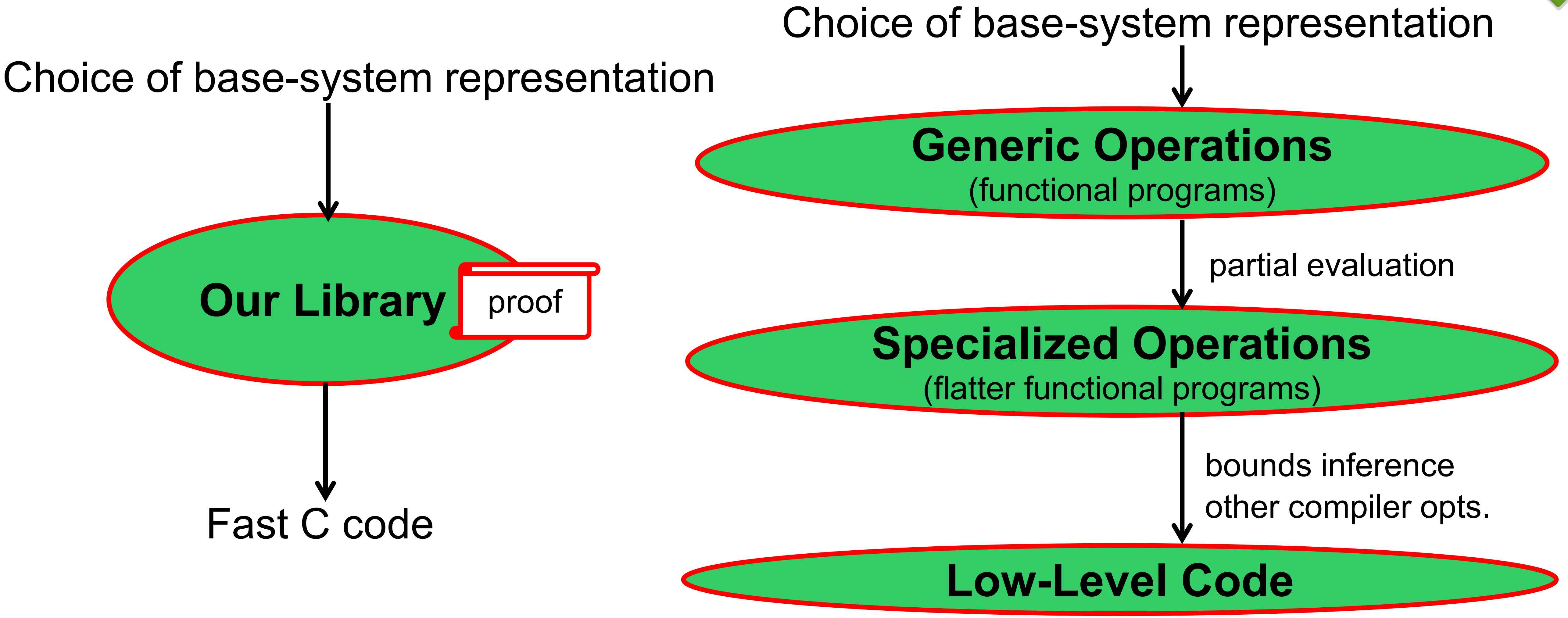
champion rep. on **32-bit processors**
(note: nonuniform bitwidths!)

Demo

Invoking Fiat Cryptography

The Fiat Cryptography approach

The Basic Idea



Example: Multiplication (for modulus $2^{127} - 1$)

$s = s_0 + 2^{43}\, s_1 + 2^{85}\, s_2$
$t = t_0 + 2^{43}\, t_1 + 2^{85}\, t_2$

$s \square t = u = u_0\ u_1\ u_2\ u_3\ u_4$

$$s \square t = 1 \square s_0t_0 + 2^{43} \square s_0t_1 + 2^{85} \square s_0t_2 + 2^{43} \square s_1t_0 + 2^{86} \square s_1t_1 + 2^{128} \square s_1t_2 + 2^{85} \square s_2t_0 + 2^{128} \square s_2t_1 + 2^{170} \square s_2t_2$$

$\boxed{u_0 = s_0t_0}$

$\boxed{u_1 = s_0t_1 + s_1t_0}$

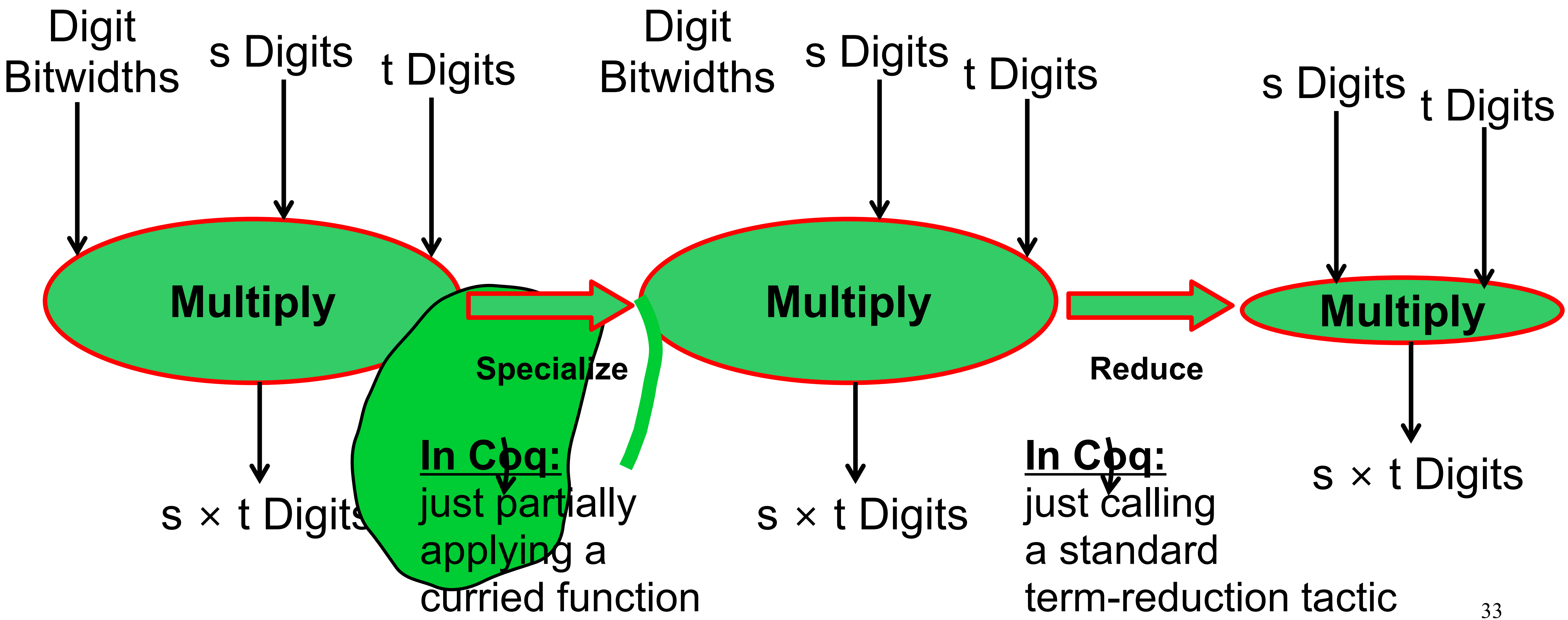
$\boxed{u_2 = s_0t_2 + 2s_1t_1 + s_2t_0}$

$\boxed{u_3 = 2s_1t_2 + 2s_2t_1}$

$\boxed{u_4 = s_2t_2}$

$$u = u_0 + 2^{43}\, u_1 + 2^{85}\, u_2 + 2^{127}\, (u_3 + 2^{43}\, u_4)$$
$$= (u_0 + u_3) + 2^{43}\, (u_1 + u_4) + 2^{85}\, u_2$$

Time for Some Partial Evaluation



An Example

```
Definition w (i:nat) : Z := 2^Qceiling((25+1/2)*i).

Example base_25_5_mul (f g:tuple Z 10) :
  { fg : tuple Z 10 |
   (eval w fg) mod (2^255-19)
   = (eval w f * eval w g) mod (2^255-19) }.
```

```
(f0*g9+f1*g8+f2*g7+f3*g6+f4*g5+f5*g4+f6*g3+f7*g2+f8*g1+f9*g0,
 f0*g8+2*f1*g7+f2*g6+2*f3*g5+f4*g4+2*f5*g3+f6*g2+2*f7*g1+f8*g0+38*f9*g9,
 f0*g7+f1*g6+f2*g5+f3*g4+f4*g3+f5*g2+f6*g1+f7*g0+19*f8*g9+19*f9*g8,
 f0*g6+2*f1*g5+f2*g4+2*f3*g3+f4*g2+2*f5*g1+f6*g0+38*f7*g9+19*f8*g8+38*f9*g7,
 f0*g5+f1*g4+f2*g3+f3*g2+f4*g1+f5*g0+19*f6*g9+19*f7*g8+19*f8*g7+19*f9*g6,
 f0*g4+2*f1*g3+f2*g2+2*f3*g1+f4*g0+38*f5*g9+19*f6*g8+38*f7*g7+19*f8*g6+38*f9*g5,
 f0*g3+f1*g2+f2*g1+f3*g0+19*f4*g9+19*f5*g8+19*f6*g7+19*f7*g6+19*f8*g5+19*f9*g4,
 f0*g2+2*f1*g1+f2*g0+38*f3*g9+19*f4*g8+38*f5*g7+19*f6*g6+38*f7*g5+19*f8*g4+38*f9*g3,
 f0*g1+f1*g0+19*f2*g9+19*f3*g8+19*f4*g7+19*f5*g6+19*f6*g5+19*f7*g4+19*f8*g3+19*f9*g2,
 f0*g0+38*f1*g9+19*f2*g8+38*f3*g7+19*f4*g6+38*f5*g5+19*f6*g4+38*f7*g3+19*f8*g2+38*f9*g1)
```

Compiling to Low-Level Code

$$1 \times (1 \times 2^{52} + (1 \times x + 0)) + (1 \times (1 \times (-y) + 0) + 0)$$

reify to syntax tree

constant-fold

$$(2^{52} + x) - y$$

flatten

let c = 2^{52} + x in
let d = c – y in
d

Assume: $0 \leq x, y \leq 2^{51} + 2^{48}$
Deduce: $2^{52} \leq c \leq 2^{52} + 2^{51} + 2^{48}$
Deduce: $2^{51} - 2^{48} \leq d \leq 2^{52} + 2^{51} + 2^{48}$

infer bounds

```
uint64_t c = 2^52 +
x;
uint64_t d = c - y;
```

Implementation and Experiments

- ~38 kloc in full library (including significant parts that belong in stdlib)
- Very little code needed to instantiate to new prime moduli.
- In fact, we wrote a Python script (under 3000 lines) to generate parameters automatically from prime numbers, written suggestively, e.g. $2^{256} - 2^{224} + 2^{192} + 2^{96} – 1$.
- This script is outside the TCB, since any successful compilation is guaranteed to implement correct arithmetic.

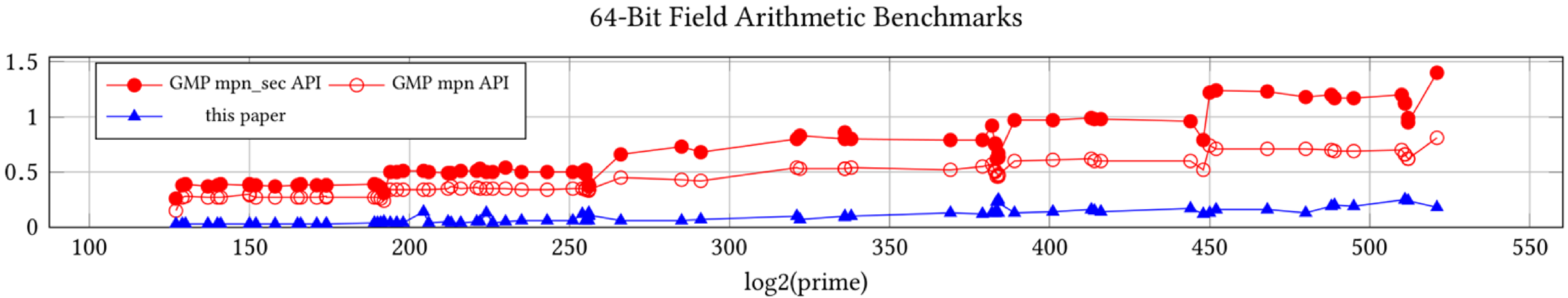
Q: Where do we get a lot of reasonable moduli?

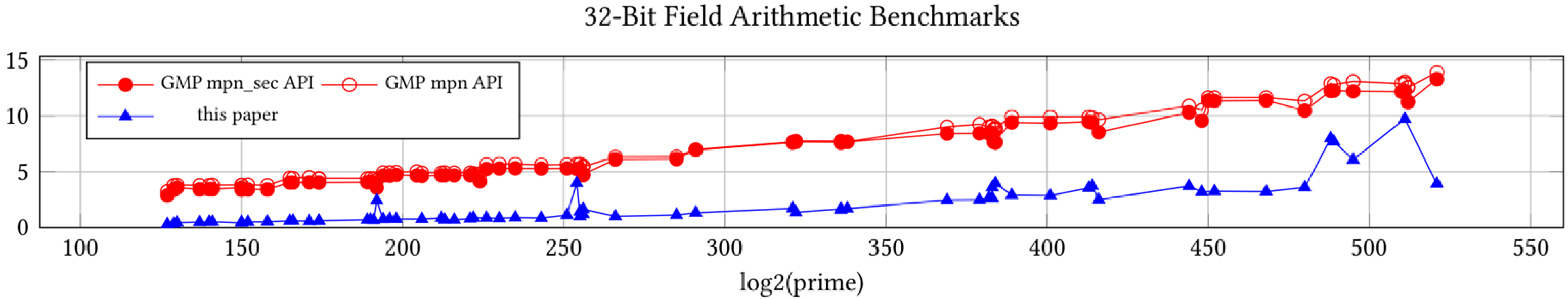
A: Scrape all prime numbers appearing in a popular mailing list.

We used the elliptic curves list at moderncrypto.org.
We found about 80 primes.

Only a few turned out to be terrible ideas posted by newbies.

Many-Primes Experiment

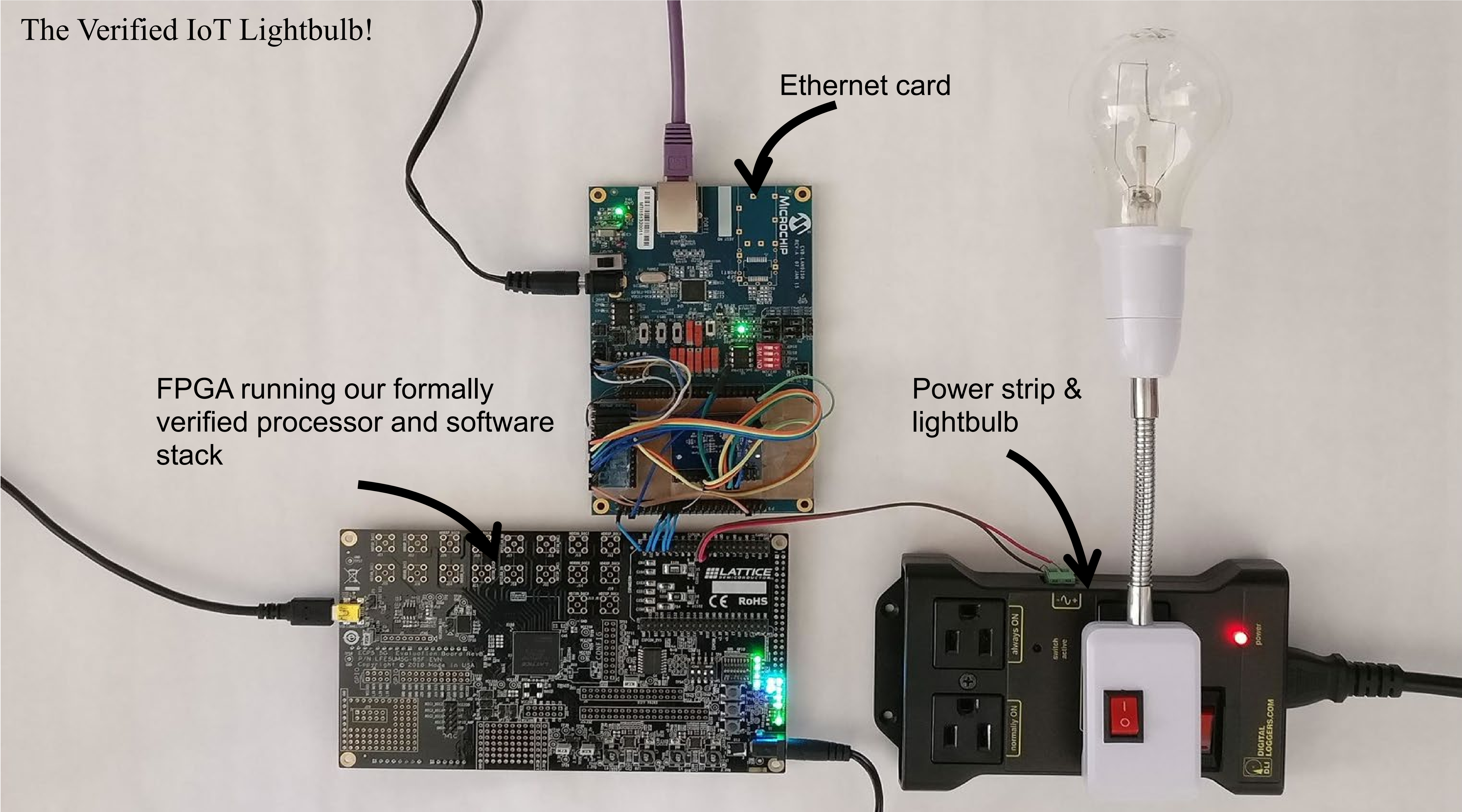




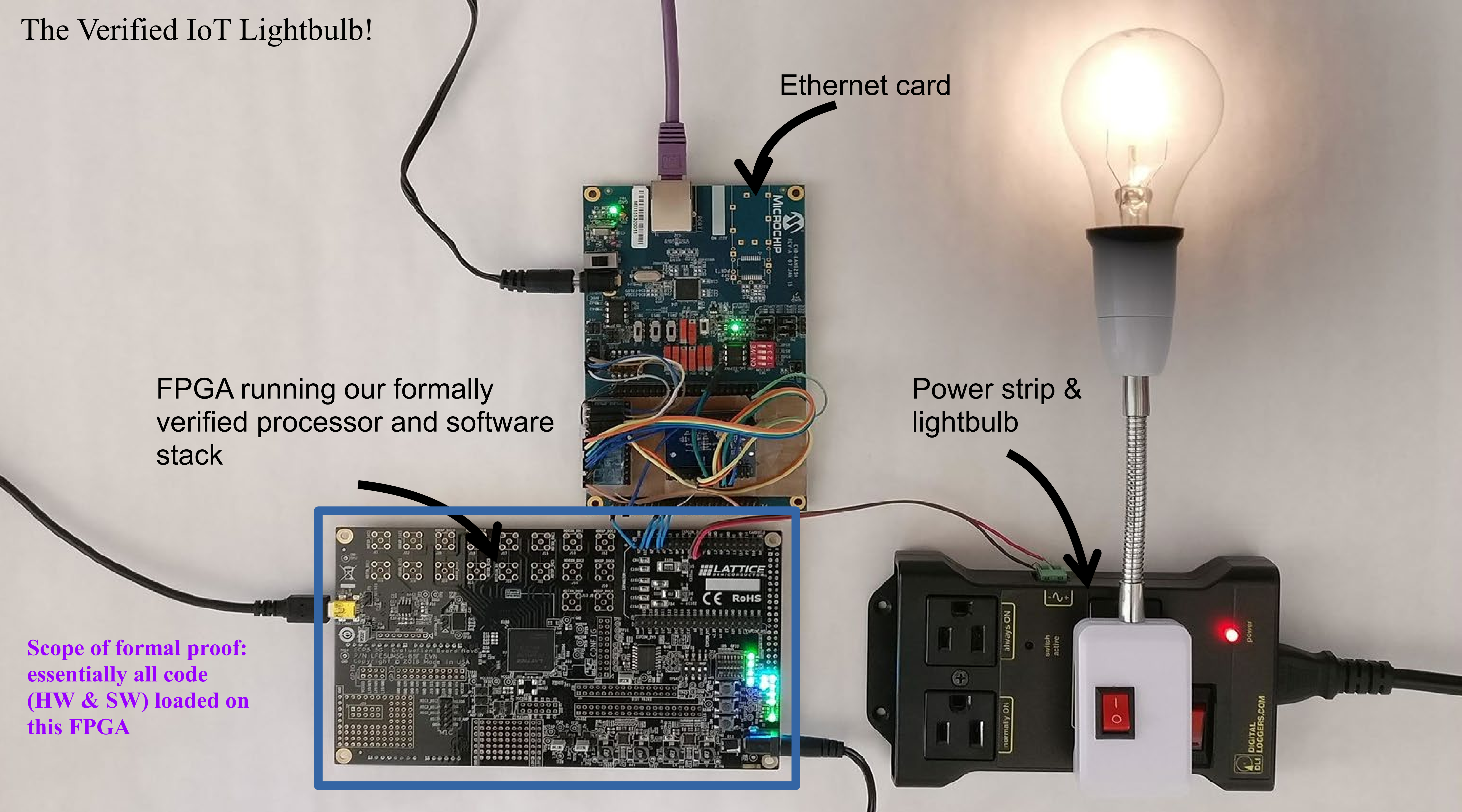
P256 Mixed Addition

Implementation	CPU cycles	μs at 2.6GHz
`OpenSSL` AMD64+ADX asm	544	.21
`OpenSSL` AMD64 asm	644	.25
this work, `icc`	1112	.43
this work, `gcc`	1808	.70
`OpenSSL` C	1968	.76

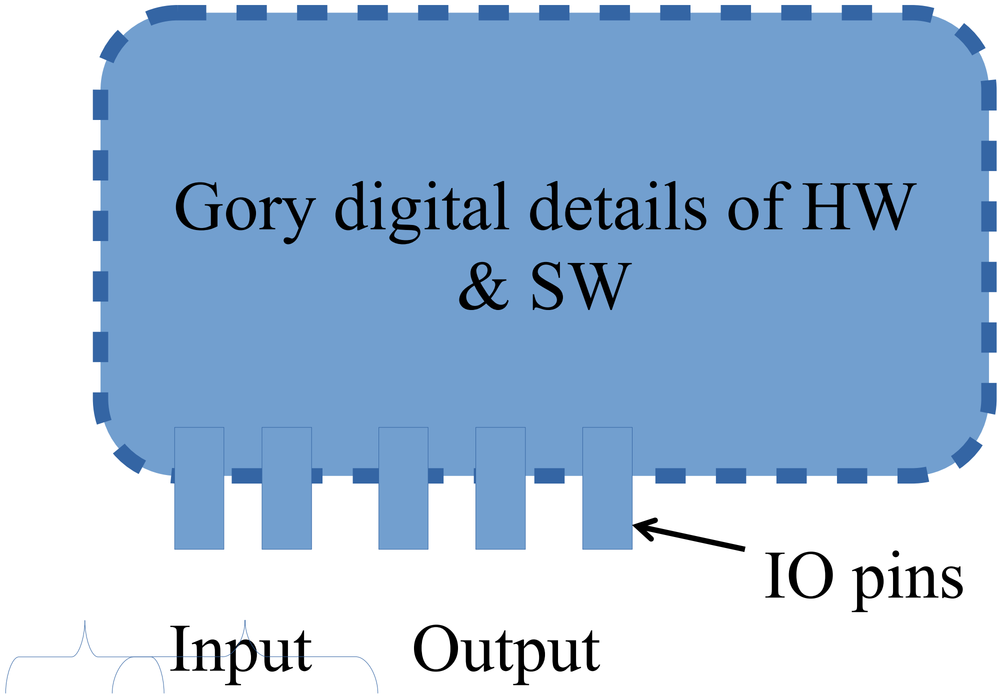
Towards correct-by-construction cryptographic appliances


The Verified IoT Lightbulb!
Ethernet card
FPGA running our formally verified processor and software stack
Power strip & lightbulb
MICROCHIP
LATTICE
CE RoHS
always ON
normally ON
switch active
power
DIGITAL LOGGERS.COM

The Verified IoT Lightbulb!



Specification?



<u>Consider all **traces** the system could generate:</u>
00100, 11000, 00100, ...

Recording pin values each cycle

Output pins: we as spec-writers may mandate what they are allowed to be!

Input pins: the environment may choose any values each cycle.

"Output pin controlling lightbulb is only on if the last valid Ethernet packet said so."

Key Layers of End-to-End Proof

Controller Spec (Trace Predicate)

Controller SW

Programming Language Semantics

Verified Compiler

ISA Family Semantics

Verified Hardware

RTL Semantics

Disappearing Specs

Controller Spec (Trace Predicate)

Must get this spec right.

Everything this box hides is no longer trusted!

System as a Proved Black Box

Must get this one right, too.

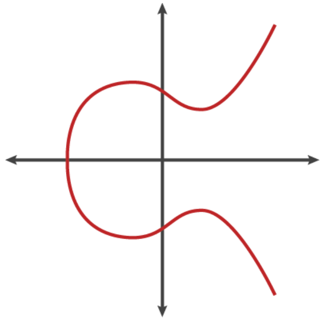
RTL Semantics

Expanding Scope

Abstract security property

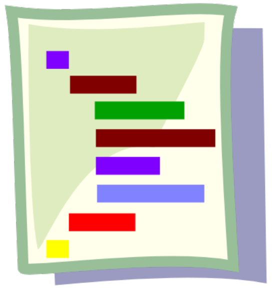
"Knowledge of the secret key is needed to produce a signature in polynomial time."

Mathematical algorithm



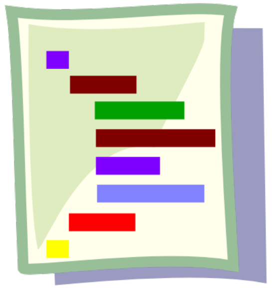
$$y^2 = x^3 - x + 1$$

High-level modular arithmetic

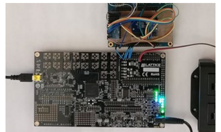


$$x = x_0, x_1, \ldots, x_n$$

Low-level code



specialized assembly code



Protocol verification, perhaps following past work by Appel & others, using our new higher-level notation for protocol programming

Synthesizing C code for more of a crypto library (beyond straightline code) with Rupicola, a proof-generating compiler

Genetic search for fast assembly code (collaboration with Prof. Yuval Yarom et al.), plus formally verified program-equivalence checker

Connect to verified HW & systems software https://github.com/mit-plv/fiat-crypto

https://github.com/mit-plv/bedrock2