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

Key exchange:
Establish a shared secret

Symmetric crypto:
main communication bytestream

Digital signature:
Server proves it was the one who helped pick the shared secret

Public Key / Certificate
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?

**Algorithms**

**Prime #s**

**HW Arches**

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.”

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.

The Proof Workflow of the Future

Libraries Galore

Version Control

Continuous Integration

Proof Checking
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

Proof Assistant
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]?

Human User → tactic → Tactic Engine

proof state

new tactic libraries

Trusted

Untrusted

Certificate

proof term

proof checker
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?

Self-Contained Verified Unit

No longer trusted!
<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>System-integration tests <em>and</em> unit tests, since combined state space grows <em>exponentially</em> as we compose pieces</td>
<td>System-integration theorems imply proper functioning of all components.</td>
</tr>
<tr>
<td>Careful code review of all components, since a corner-case bug in any of them can wreck the whole system</td>
<td>Careful code review only of <em>externally facing specs</em></td>
</tr>
</tbody>
</table>
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."

Mathematical algorithm

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

Low-level code

specialized assembly code

protocol verification

implementation synthesis
Correct-by-Construction Cryptography

Mathematical algorithm

point = (x, y)

High-level modular arithmetic

x = x_0, x_1, ..., x_n
(mathematical integers)

Optimized point format

point = (x, y, z, t)

Low-level code

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

Squaring a number (64-bit)

\[ \lambda \ (x7, x8, x6, x4, x2) \ |
\]
\[
\text{uint64_t x9} = x2 \ * 0x2;
\text{uint64_t x10} = x4 \ * 0x2;
\text{uint64_t x11} = x6 \ * 0x2 \ * 0x13;
\text{uint64_t x12} = x7 \ * 0x13;
\text{uint64_t x13} = x12 \ * 0x02;
\text{uint128_t x14} = (\text{uint128_t}) x12 \ * x2 + (\text{uint128_t}) x13 \ * x4 + (\text{uint128_t}) x11 \ * x8;
\text{uint128_t x15} = (\text{uint128_t}) x9 \ * x4 + (\text{uint128_t}) x13 \ * x6 + (\text{uint128_t}) x8 \ * (x8 \ * 0x13);
\text{uint128_t x16} = (\text{uint128_t}) x9 \ * x6 + (\text{uint128_t}) x4 \ * x4 + (\text{uint128_t}) x13 \ * x8;
\text{uint128_t x17} = (\text{uint128_t}) x9 \ * x8 + (\text{uint128_t}) x10 \ * x6 + (\text{uint128_t}) x7 \ * x12;
\text{uint128_t x18} = (\text{uint128_t}) x9 \ * x7 + (\text{uint128_t}) x10 \ * x10 \ * 0x13 + (\text{uint128_t}) x6 \ * x6;
\text{uint64_t x19} = (\text{uint128_t}) (x14 >> 0x33);
\text{uint64_t x20} = (\text{uint64_t}) x14 \ & \ 0x7fffffffffffffff;
\text{uint128_t x21} = x19 + x15;
\text{uint64_t x22} = (\text{uint64_t}) (x21 >> 0x33);
\text{uint64_t x23} = (\text{uint64_t}) x21 \ & \ 0x7fffffffffffffff;
\text{uint128_t x24} = x22 + x16;
\text{uint64_t x25} = (\text{uint64_t}) (x24 >> 0x33);
\text{uint64_t x26} = (\text{uint64_t}) x24 \ & \ 0x7fffffffffffffff;
\text{uint128_t x27} = x25 + x17;
\text{uint64_t x28} = (\text{uint64_t}) (x27 >> 0x33);
\text{uint64_t x29} = (\text{uint64_t}) x27 \ & \ 0x7fffffffffffffff;
\text{uint128_t x30} = x28 + x18;
\text{uint64_t x31} = (\text{uint64_t}) (x30 >> 0x33);
\text{uint64_t x32} = (\text{uint64_t}) x30 \ & \ 0x7fffffffffffffff;
\text{uint64_t x33} = x30 + 0x13 \ * 0x31;
\text{uint64_t x34} = x33 \ >> 0x33;
\text{uint64_t x35} = \text{x33} \ & \ 0x7fffffffffffffff;
\text{uint64_t x36} = x34 + x23;
\text{uint64_t x37} = x36 \ >> 0x33;
\text{uint64_t x38} = x36 \ & \ 0x7fffffffffffffff;
\text{return (Return x32, Return x29, x37 + x26, Return x38, Return x35))}

Squaring a number (32-bit)

\[ \lambda \ (x7, x8, x6, x4, x2) \ |
\]
\[
\text{uint32_t x9} = x2 \ * 0x2;
\text{uint32_t x10} = x4 \ * 0x2;
\text{uint32_t x11} = x6 \ * 0x2 \ * 0x13;
\text{uint32_t x12} = x7 \ * 0x13;
\text{uint32_t x13} = x12 \ * 0x02;
\text{uint32_t x14} = (\text{uint32_t}) x12 \ * x2 + (\text{uint32_t}) x13 \ * x4 + (\text{uint32_t}) x11 \ * x8;
\text{uint32_t x15} = (\text{uint32_t}) x9 \ * x4 + (\text{uint32_t}) x13 \ * x6 + (\text{uint32_t}) x8 \ * (x8 \ * 0x13);
\text{uint32_t x16} = (\text{uint32_t}) x9 \ * x6 + (\text{uint32_t}) x4 \ * x4 + (\text{uint32_t}) x13 \ * x8;
\text{uint32_t x17} = (\text{uint32_t}) x9 \ * x8 + (\text{uint32_t}) x10 \ * x6 + (\text{uint32_t}) x7 \ * x12;
\text{uint32_t x18} = (\text{uint32_t}) x9 \ * x7 + (\text{uint32_t}) x10 \ * x10 \ * 0x13 + (\text{uint32_t}) x6 \ * x6;
\text{uint32_t x19} = (\text{uint32_t}) (x14 >> 0x33);
\text{uint32_t x20} = (\text{uint32_t}) x14 \ & \ 0x3ffffff;
\text{uint32_t x21} = x19 + x15;
\text{uint32_t x22} = (\text{uint32_t}) (x21 >> 0x33);
\text{uint32_t x23} = (\text{uint32_t}) x21 \ & \ 0x3ffffff;
\text{uint32_t x24} = x22 + x16;
\text{uint32_t x25} = (\text{uint32_t}) (x24 >> 0x33);
\text{uint32_t x26} = (\text{uint32_t}) x24 \ & \ 0x3ffffff;
\text{uint32_t x27} = x25 + x17;
\text{uint32_t x28} = (\text{uint32_t}) (x27 >> 0x33);
\text{uint32_t x29} = (\text{uint32_t}) x27 \ & \ 0x3ffffff;
\text{uint32_t x30} = x28 + x18;
\text{uint32_t x31} = (\text{uint32_t}) (x30 >> 0x33);
\text{uint32_t x32} = (\text{uint32_t}) x30 \ & \ 0x3ffffff;
\text{uint32_t x33} = x30 + 0x13 \ * 0x31;
\text{uint32_t x34} = x33 \ >> 0x33;
\text{uint32_t x35} = \text{x33} \ & \ 0x3ffffff;
\text{uint32_t x36} = x34 + x23;
\text{uint32_t x37} = x36 \ >> 0x33;
\text{uint32_t x38} = x36 \ & \ 0x3ffffff;
\text{return (Return x32, Return x29, x37 + x26, Return x38, Return x35))}

26
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 $c << 2^k$. (Called *pseudo-Mersenne*.)

Example of a fast operation: *modular reduction*

\[
\begin{align*}
t &= x + 2^ky \pmod{2^k - c} \quad \text{too big to fit below the modulus!} \\
&= x + (2^k - c + c)y \pmod{2^k - c} \\
&= x + (2^k - c)y + cy \pmod{2^k - c} \\
&= x + cy \pmod{2^k - c}
\end{align*}
\]
Representing Numbers mod $2^{255} - 19$

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

\[ t = t_0 \ t_1 \ t_2 \ t_3 \ t_4 \ t_5 \ t_6 \ t_7 \]
\[ = (t_0 + 2^{64} t_1 + \ldots) + 2^{256} (t_4 + 2^{64} t_5 + \ldots) \]

each “digit” fits in 64-bit register

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

Choice of base-system representation

Our Library

proof

Generic Operations
(functional programs)

partial evaluation

Specialized Operations
(flatter functional programs)

bounds inference
other compiler opts.

Low-Level Code

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

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

\[ s \begin{array}{c} \uparrow \end{array} t = 1 \begin{array}{c} \uparrow \end{array} s_0 t_0 + 2^{43} \begin{array}{c} \uparrow \end{array} s_0 t_1 + 2^{85} \begin{array}{c} \uparrow \end{array} s_0 t_2 + 2^{43} \begin{array}{c} \uparrow \end{array} s_1 t_0 + 2^{86} \begin{array}{c} \uparrow \end{array} s_1 t_1 + 2^{128} \begin{array}{c} \uparrow \end{array} s_1 t_2 + 2^{85} \begin{array}{c} \uparrow \end{array} s_2 t_0 + 2^{128} \begin{array}{c} \uparrow \end{array} s_2 t_1 + 2^{170} \begin{array}{c} \uparrow \end{array} s_2 t_2 \]

\[ u_0 = s_0 t_0, \quad s_2 t_0 + 2^{128} s_1 t_1 + s_1 t_0 + 2^{170} s_2 t_2 \]
\[ u_2 = s_0 t_2 + 2 s_1 t_1 + s_2 t_0 \]
\[ u_3 = 2 s_1 t_2 + 2 s_2 t_1 \]
\[ u_4 = s_2 t_2 \]

\[ u = u_0 + 2^{43} u_1 + 2^{85} u_2 + 2^{127} (u_3 + 2^{43} u_4) \]
\[ = (u_0 + u_3) + 2^{43} (u_1 + u_4) + 2^{85} u_2 \]
Time for Some Partial Evaluation

Multiply

Digit Bitwidths

s Digits

t Digits

Digit Bitwidths

s Digits

t Digits

Multiply

s × t Digits

Specialize

In Coq:
just partially applying a curried function

Multiply

s × t Digits

Reduce

In Coq:
just calling a standard term-reduction tactic

s Digits

t Digits

s × t Digits
An Example

Definition \( w (i: \text{nat}) : \mathbb{Z} := 2^{\lceil 25 + 1/2 \rceil i} \).

Example base_25_5_mul \((f, g: \text{tuple} \ Z \ \text{10}) : \) : 
\[
\{ \ \text{fg} : \ \text{tuple} \ Z \ \text{10} \ |
\quad \text{(eval w fg) mod (2^{255-19})}
\quad = \ \text{(eval w f * eval w g) mod (2^{255-19})} \ \}
\]

\[
(f0*g9+f1*g8+f2*g7+f3*g6+f4*g5+f5*g4+f6*g3+f7*g2+f8*g1+f9*g0,
\quad f0*g8+2*f1*g7+f2*g6+2*f3*g5+f4*g4+2*f5*g3+f6*g2+2*f7*g1+f8*g0+38*f9*g9,
\quad f0*g7+f1*g6+f2*g5+f3*g4+f4*g3+f5*g2+f6*g1+f7*g0+19*f8*g9+19*f9*g8,
\quad f0*g6+2*f1*g5+f2*g4+2*f3*g3+f4*g2+2*f5*g1+f6*g0+38*f7*g9+19*f8*g8+38*f9*g7,
\quad f0*g5+f1*g4+f2*g3+f3*g2+f4*g1+f5*g0+19*f6*g9+19*f7*g8+19*f8*g7+19*f9*g6,
\quad f0*g4+2*f1*g3+f2*g2+2*f3*g1+f4*g0+38*f5*g9+19*f6*g8+38*f7*g7+19*f8*g6+38*f9*g5,
\quad f0*g3+3*f1*g2+f2*g1+f3*g0+19*f4*g9+19*f5*g8+19*f6*g7+19*f7*g6+19*f8*g5+19*f9*g4,
\quad f0*g2+2*f1*g1+f2*g0+38*f3*g9+19*f4*g8+38*f5*g7+19*f6*g6+38*f7*g5+19*f8*g4+38*f9*g3,
\quad f0*g1+f1*g0+19*f2*g9+19*f3*g8+19*f4*g7+19*f5*g6+19*f6*g5+19*f7*g4+19*f8*g3+19*f9*g2, 
\quad 34
\quad f0*g0+38*f1*g9+19*f2*g8+38*f3*g7+19*f4*g6+38*f5*g5+19*f6*g4+38*f7*g3+19*f8*g2+38*f9*g1)
\]
Compiling to Low-Level Code

1 × (1 × 2^{52} + (1 × x + 0)) + (1 × (1 × (-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

Deduce: 2^{52} ≤ c ≤ 2^{52} + 2^{51} + 2^{48}

Deduce: 2^{51} – 2^{48} ≤ d ≤ 2^{52} + 2^{51} + 2^{48}

Assume: 0 ≤ x, y ≤ 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

64-Bit Field Arithmetic Benchmarks

32-Bit Field Arithmetic Benchmarks
## P256 Mixed Addition

<table>
<thead>
<tr>
<th>Implementation</th>
<th>CPU cycles</th>
<th>μs at 2.6GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSL AMD64+ADX asm</td>
<td>544</td>
<td>.21</td>
</tr>
<tr>
<td>OpenSSL AMD64 asm</td>
<td>644</td>
<td>.25</td>
</tr>
<tr>
<td><em>this work</em>, icc</td>
<td>1112</td>
<td>.43</td>
</tr>
<tr>
<td><em>this work</em>, gcc</td>
<td>1808</td>
<td>.70</td>
</tr>
<tr>
<td>OpenSSL C</td>
<td>1968</td>
<td>.76</td>
</tr>
</tbody>
</table>
Towards correct-by-construction cryptographic appliances
The Verified IoT Lightbulb!

FPGA running our formally verified processor and software stack

Ethernet card

Power strip & lightbulb
The Verified IoT Lightbulb!

FPGA running our formally verified processor and software stack

Ethernet card

Power strip & lightbulb

Scope of formal proof: essentially all code (HW & SW) loaded on this FPGA
Specification?

Gory digital details of HW & SW

IO pins

Consider all traces the system could generate:
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.

System as a Proved Black Box

Must get this one right, too.

RTL Semantics

Everything this box hides is no longer trusted!

Must get this one right, too.
“Knowledge of the secret key is needed to produce a signature in polynomial time.”

\[ y^2 = x^3 - x + 1 \]

\[ x = x_0, x_1, \ldots, x_n \]

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