Correct-by-Construction Cryptographic Arithmetic in Coq

Adam Chlipala, MIT CSAIL
Lambda Days
February 2021

Joint work with:
Andres Erbsen, Jade Philipoom, Jason Gross, and Robert Sloan
Web Browsing with SSL

Key exchange:
Establish a shared secret

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

Symmetric crypto:
main communication bytestream

Symmetric Key

Web Server

Public Key / Certificate

Private Key

Browser

Symmetric Key
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.
  - Primarily: **big-integer modular arithmetic**
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.
Correct-by-Construction Cryptography

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

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

specificial assembly code

Abstract security property

Mathematical algorithm

Low-level code

protocol verification

implementation synthesis
Correct-by-Construction Cryptography

Mathematical algorithm

Optimized point format

High-level modular arithmetic

Low-level code

point = (x, y)

point = (x, y, z, t)

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

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

classic verification of functional programs

compile-time code specialization

compiler verification
Squaring a number (64-bit)

\[
\lambda (x_7, x_8, x_6, x_4, x_2) \to (x_7, x_8, x_6, x_4, x_2) \cdot x_2
\]

uint64_t x38 = x36 & 0x7ffffffffffff;
uint64_t x37 = x36 >> 0x33;
uint64_t x36 = x34 + x23;
uint64_t x35 = x33 & 0x7ffffffffffff;
uint64_t x34 = x33 >> 0x33;
uint64_t x33 = x20 + 0x13 * x31;
uint64_t x32 = (uint64_t) x30 & 0x7ffffffffffff;
uint64_t x31 = (uint64_t) (x30 >> 0x33);
uint128_t x30 = x28 + x18;
uint64_t x29 = (uint64_t) x27 & 0x7ffffffffffff;
uint64_t x28 = (uint64_t) (x27 >> 0x33);
uint128_t x27 = x25 + x17;
uint64_t x26 = (uint64_t) x24 & 0x7ffffffffffff;
uint64_t x24 = (uint64_t) (x24 >> 0x33);
uint64_t x23 = (uint64_t) x21 & 0x3ffffff;
uint64_t x21 = (uint64_t) (x21 >> 0x1a);
uint64_t x20 = x14 + 0x13 * x29;
uint64_t x19 = x12 * 0x2;
uint64_t x18 = x7 * 0x13;
uint64_t x17 = x6 * 0x2 * 0x13;
uint64_t x16 = x4 * 0x2;
uint64_t x15 = x2 * 0x2;
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

\[
t = x + 2^k y \pmod{2^k - c}
\]

\[
t = x + (2^k - c + c)y \pmod{2^k - c}
\]

\[
t = x + (2^k - c)y + cy \pmod{2^k - c}
\]

\[
t = x + cy \pmod{2^k - c}
\]

too big to fit below the modulus!
Representing Numbers mod $2^{255} - 19$

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

$t = t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7$

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

each “digit” fits in 64-bit register

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

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

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

champion rep. on **64-bit processors**

(note: not using full bitwidth!)

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

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

champion rep. on **32-bit processors**

(note: nonuniform bitwidths!)

$t = s_0 + 2^{26} s_1 + 2^{51} s_2 + 2^{77} s_3 + ...$
The Basic Idea

Choice of base-system representation

Our Library
 proof

Fast C code

Choice of base-system representation

Generic Operations
(functional programs)
 partial evaluation

Specialized Operations
(flatter functional programs)
 bounds inference
 other compiler opts.

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

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

\[
s \times t = u = u_0 u_1 u_2 u_3 u_4
\]

\[
s \times t = 1 \times s_0 t_0 + 2^{43} \times s_0 t_1 + 2^{85} \times s_0 t_2
\]
\[
u_0 = s_0 t_0 + 2^{43} \times s_0 t_1 + 2^{86} \times s_0 t_1 + 2^{128} \times s_1 t_2
\]
\[
u_1 = s_0 t_1 + s_1 t_1 + 2^{85} \times s_1 t_2 + 2^{128} \times s_2 t_1 + 2^{170} \times s_2 t_2
\]
\[
u_2 = s_0 t_2 + 2s_1 t_1 + s_2 t_2
\]
\[
u_3 = 2s_1 t_2 + 2s_1 t_1
\]
\[
u_4 = s_2 t_2
\]

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

In Coq: just partially applying a curried function

In Coq: just calling a standard term-reduction tactic
An Example

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

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

\[
(f0 \cdot g9 + f1 \cdot g8 + f2 \cdot g7 + f3 \cdot g6 + f4 \cdot g5 + f5 \cdot g4 + f6 \cdot g3 + f7 \cdot g2 + f8 \cdot g1 + f9 \cdot g0, \\
(f0 \cdot g8 + 2 \cdot f1 \cdot g7 + f2 \cdot g6 + 2 \cdot f3 \cdot g5 + f4 \cdot g4 + 2 \cdot f5 \cdot g3 + f6 \cdot g2 + 2 \cdot f7 \cdot g1 + f8 \cdot g0 + 38 \cdot f9 \cdot g9, \\
f0 \cdot g7 + f1 \cdot g6 + f2 \cdot g5 + f3 \cdot g4 + f4 \cdot g3 + f5 \cdot g2 + f6 \cdot g1 + f7 \cdot g0 + 19 \cdot f8 \cdot g9 + 19 \cdot f9 \cdot g8, \\
f0 \cdot g6 + 2 \cdot f1 \cdot g5 + f2 \cdot g4 + 2 \cdot f3 \cdot g3 + f4 \cdot g2 + 2 \cdot f5 \cdot g1 + f6 \cdot g0 + 38 \cdot f7 \cdot g9 + 19 \cdot f8 \cdot g8 + 38 \cdot f9 \cdot g7, \\
f0 \cdot g5 + f1 \cdot g4 + f2 \cdot g3 + f3 \cdot g2 + f4 \cdot g1 + f5 \cdot g0 + 19 \cdot f6 \cdot g9 + 19 \cdot f7 \cdot g8 + 19 \cdot f8 \cdot g7 + 19 \cdot f9 \cdot g6, \\
f0 \cdot g4 + 2 \cdot f1 \cdot g3 + f2 \cdot g2 + 2 \cdot f3 \cdot g1 + f4 \cdot g0 + 38 \cdot f5 \cdot g9 + 19 \cdot f6 \cdot g8 + 38 \cdot f7 \cdot g7 + 19 \cdot f8 \cdot g6 + 38 \cdot f9 \cdot g5, \\
f0 \cdot g3 + f1 \cdot g2 + f2 \cdot g1 + f3 \cdot g0 + 19 \cdot f4 \cdot g9 + 19 \cdot f5 \cdot g8 + 19 \cdot f6 \cdot g7 + 19 \cdot f7 \cdot g6 + 19 \cdot f8 \cdot g5 + 19 \cdot f9 \cdot g4, \\
f0 \cdot g2 + 2 \cdot f1 \cdot g1 + f2 \cdot g0 + 38 \cdot f3 \cdot g9 + 19 \cdot f4 \cdot g8 + 38 \cdot f5 \cdot g7 + 19 \cdot f6 \cdot g6 + 38 \cdot f7 \cdot g5 + 19 \cdot f8 \cdot g4 + 38 \cdot f9 \cdot g3, \\
f0 \cdot g1 + f1 \cdot g0 + 19 \cdot f2 \cdot g9 + 19 \cdot f3 \cdot g8 + 19 \cdot f4 \cdot g7 + 19 \cdot f5 \cdot g6 + 19 \cdot f6 \cdot g5 + 19 \cdot f7 \cdot g4 + 19 \cdot f8 \cdot g3 + 19 \cdot f9 \cdot g2, \\
f0 \cdot g0 + 38 \cdot f1 \cdot g9 + 19 \cdot f2 \cdot g8 + 38 \cdot f3 \cdot g7 + 19 \cdot f4 \cdot g6 + 38 \cdot f5 \cdot g5 + 19 \cdot f6 \cdot g4 + 38 \cdot f7 \cdot g3 + 19 \cdot f8 \cdot g2 + 38 \cdot f9 \cdot g1)
\]
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**
- **Flatten**
  - `let c = 2^{52} + x in`
  - `let d = c - y in`
- **Infer bounds**

Assume: \(0 \leq x, y \leq 2^{51} + 2^{48}\)

Deduce: \(2^{52} \leq c \leq 2^{52} + 2^{51} + 2^{48}\)

Deduce: \(2^{51} - 2^{48} \leq d \leq 2^{52} + 2^{51} + 2^{48}\)

```c
uint64_t c = 2^{52} + x;
uint64_t d = c - y;
return d
```
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>
Next Steps

• Close the performance & trust gap with assembly by extending verified pipeline.
  – Maybe verify an *equivalence checker* between our code and code handwritten by experts.

• Extend code generation to apply to higher-level crypto code, like curve arithmetic, not just field arithmetic.
  – Requires handling loops, function calls, etc.

• Goal (medium-term): a complete TLS implementation derived in a correct-by-construction way!
https://github.com/mit-plv/fiat-crypto