Fiat Cryptography

Static Analysis section, **Lecture 27**

Pavel Panchekha

CS 6110, U of Utah

16 April 2020

Final Presentation

Final project presentations **due tonight**

- **Record and upload** the presentations

All class presentations **posted tomorrow**

- Watch **all other** presentations
- Write **one question** for each other presentation
- **Answer all questions** for your presentation

Final Project

Submit code by Tuesday (deadline changed)

- We'll **run quicksort** on test cases
- We'll **verify programs**, and try to verify false things
- Include a **README** locating each project component

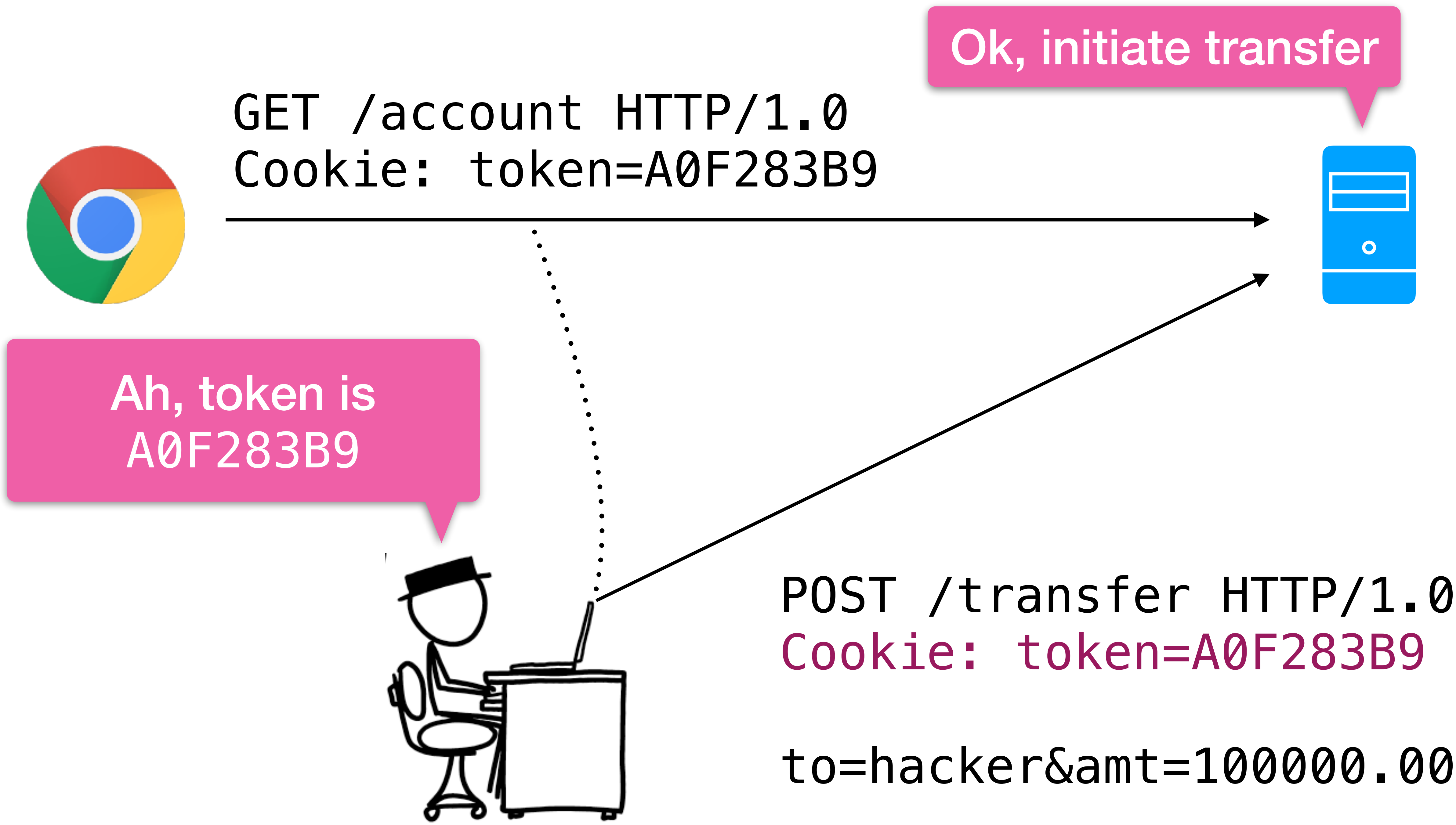
For group projects, **upload whatever is relevant**

- Diffs, evaluation scripts, evaluation data
- Group projects will be graded by **reading code**

Internet Cryptography

Why it must be fast and also secure

Encrypted Connections



TLS and HTTPS

Idea: **encrypt** network traffic, no eavesdropping

```
GET /account HTTP/1.0
Cookie: token=A0F283B9
```

Encryption Key Known

```
+2Q801GeTw/vdK7y2pu/aeBe/
3wcsEqr10AU22RfZgNHRyDDoBEm
wEnXYLl4QlKU
```

Encryption Key Unknown

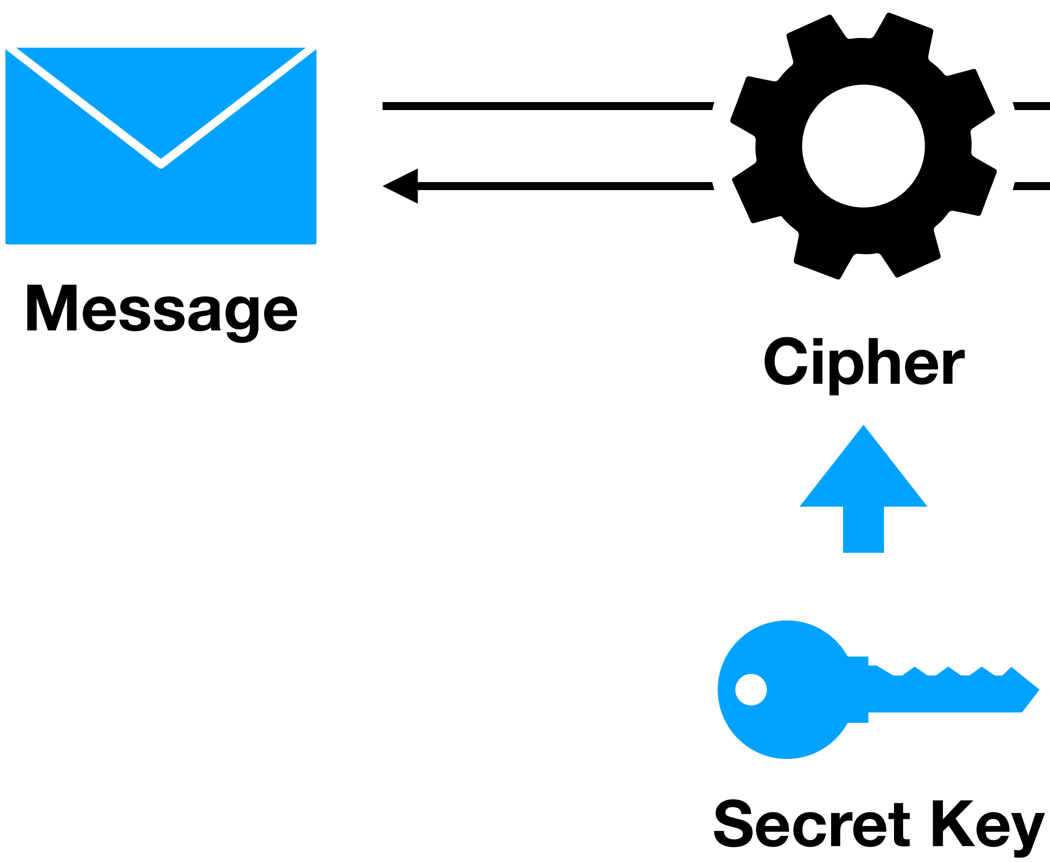
Transport Layer Security: encrypted sockets

HTTPS: **HTTP over TLS** (over TCP over IP)

Later versions more modular, more secure, more ciphers

How Encryption Works

Encrypt and decrypt messages using a **cipher**



Key Exchange

Public base
x

Private functions
$f, g, h, \ldots$

Lots of them!

Commutative
$f(g(x)) = g(f(x))$

Diffie-Hellman Key Exchange

Ralph Merkle

Martin Hellman

Whitfield Diffie

Cryptography

Public base
x

Private functions
$f, g, h, \ldots$

Commutative
$f(g(x)) = g(f(x))$

Need a **large space of commutative functions**

Mathematical structure called a **group**

Group must be **decided in advance** but can be public

Option 1: integer exponentiation, modulo large primes

Option 2: multiplication on elliptic curves modulo large primes

Elliptic Cur

Arithmetic modulo p

Ma

Curve P-256:
$y^2 = x^3 - 3x + A$ (with specific constants A, B)
$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1, x = B$

tative!

Curve X25519:
$y^2 = x^3 + 48662x^2 + x$ (different format)
$p = 2^{255} - 19, x = 9$

Standardized choice curve, p, and base x

Image from @YassineMrabet on Wikipedia

Implementing This

Need **fast arithmetic modulo** $p > 2^{64}$

- Key exchange a **latency bottleneck** for establishing connection
- Servers have **many connections**, limited CPU cycles
- Implementation must be **constant-time** for any input

Represent numbers as **vectors of machine integers**

$$x = x_0 + 2^{64}x_1 + 2^{128}x_2 + 2^{192}x_3$$

Implementing This

$$x = x_0 + 2^{64}x_1 + 2^{128}x_2 + 2^{192}x_3$$

$$\times \quad y = y_0 + 2^{64}y_1 + 2^{128}y_2 + 2^{192}y_3$$

$$2^{255} = 19 \pmod{2^{255} - 19}$$

1	2^{64}	2^{128}	2^{192}	2^{256}
x_0y_0	x_1y_0	x_0y_2	x_0y_3	x_3y_1
	x_0y_1	x_1y_1	x_1y_2	x_2y_2
		x_0y_2	x_2y_1	x_1y_3
			x_3y_0	

Implementing This

$$
\begin{array}{rl}
 & x = x_0 + 2^{64}x_1 + 2^{128}x_2 + 2^{192}x_3 \\
\times & y = y_0 + 2^{64}y_1 + 2^{128}y_2 + 2^{192}y_3
\end{array}
$$

1	2^{64}	2^{128}	2^{192}
x_0y_0	x_1y_0	x_0y_2	x_0y_3
	x_0y_1	x_1y_1	x_1y_2
		x_0y_2	x_2y_1
			x_3y_0
$38\,x_3y_1$	$38\,x_3y_2$	$38\,x_3y_3$	
$38\,x_2y_2$	$38\,x_2y_3$		
$38\,x_1y_3$			

$$xy = c_0 + 2^{64}c_1 + 2^{128}c_2 + 2^{192}c_3$$

Lots of other tricks!

Implementations

There are **dozens of curves** in common use

- Each involves **multiple non-trivial algorithms**
- Specialized to **many architectures** (word size, instructions, …)

K **algorithms** × N **curves** × M **machines**

Each combination **written by an expert cryptographer**

- Way too expensive, slow, fragile, rigid

Real Bugs

Even **experts make mistakes**:

```
/* XXX: Can it really happen that r<0?, See
HAC, Alg 14.42, Step 3. If so: Handle it here!*/
```

—The “ed25519” reference implementation

Partial audits have revealed a bug in this software (`r1 += 0 + carry` should be `r2 += 0 + carry` in `amd64-64-24k`)

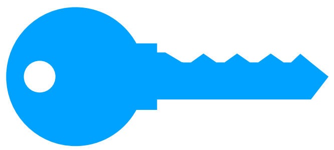
—TweetNACL: A crypto library in 100 tweets

Both by one of the **most famous cryptographers** alive

Correct by Construction

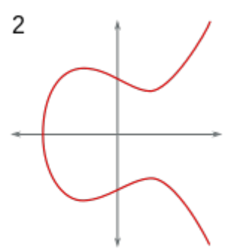
The vision for Fiat Cryptography

Crypto Pipeline



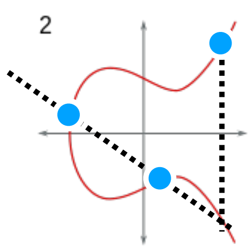
Goal of private key exchange

Decades of math



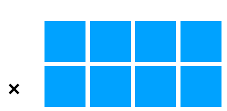
Mathematical principles of elliptic curves

Proof in Coq



Algorithm for multiplication on a curve

Proof in Coq



High-level code for multiplication modulo p

Partial Evaluation

`crypto.s`

Assembly or C code for implementation

Advantages

```
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))
```

32-bit square in X25519

Verify the **code generator**, not the code

- High-level properties **proven once**
- Low-level properties enforced by **code generator**
- Prove algorithms, curves, architectures **separately**

Lower **maintenance**, more agile

High-level Algorithm

Algorithms for **lists of arbitrary-size integers**:

$$(x_0, x_1, x_2, x_3) \rightsquigarrow [(1, x_0), (2^{32}, x_1), \ldots]$$

```
Definition mulmod {n} (a b:tuple Z n) : tuple Z n
  := let a_a := to_associational a in
     let b_a := to_associational b in
     let ab_a := Associational.mul a_a b_a in
     let abm_a := Associational.reduce s c ab_a in
     from_associational n abm_a.
```

Uses data structures, ignores overflow, etc…

Partial Execution

List handling **statically known** for a given curve:

```
Eval mulmod (a0, a1, a2, a3) (b0, b1, b2, b3)
```

```
let (c0, c1, x0) = mul2c(a0, b0, 0) in
let (c2, c3, x1) = mul2c(a0, b1, c0 + x) in
…
```

Leverages **mix of computation and proof** in Coq

Size Inference

Replace arbitrary-size with machine integers

`let (`$\texttt{c0}^{32}$`, `$\texttt{c1}^{32}$`, `$\texttt{x0}^{1}$`) = mul2c(a0, b0, 0) in`
`let (`$\texttt{c2}^{32}$`, `$\texttt{c3}^{32}$`, `$\texttt{x1}^{1}$`) = mul2c(a0, b1, c0 + x) in`
…

Abstract interpretation for integer bounds

- Implement **and verify** transfer functions for each operator
- Find **smallest machine size** to fit given variable

Hints for Analysis

Sometimes code has **hard-to-analyze tricks**

$$ab + cb + cd$$

2 adds (fast), 3 multiplies (slow)

$$(a + c)(b + d) - ad$$

3 adds (fast), 2 multiplies (slow)

Bound analysis **easier** for the first than second

- **Analyze using the first**, implement using the second
- Prove the two expressions are equal

Results

Use of Fiat Cryptography in the wild

Code

~38k lines (including boring theorems about numbers)

- Each new prime requires **minimal code**
- Automatically generate C code for **multiple architectures**

Enabled **novel experiments** with new primes

- **Scrape suggested primes** from crypto mailing list
- Generate and time each suggestion

Impossible without automated approach

BoringSSL

Fiat cryptography in **BoringSSL, used by Chrome**

Verification code used in **50% of all connections**

- Fiat crypto generated **faster 32-bit code** than existed
- Allowed experimenting with **new optimizations**

Numeric Results

Competitive with other **C, assembly implementations**

Implementation	CPU cycles	μs at 2.6GHz
amd64-64 asm	145008	56
amd64-51 asm	154248	59
sandy2x asm	154688	59
donna-c64 C	160352	62
this work, tweaked C	168364	65
this work, generated C	182580	70
OpenSSL C	348072	134
ref10 C	356716	137
ref C	6044504	2325

Ongoing Work

Speeding up code generation with custom reductions:

> There is currently a known issue where `fesub.c` for `p256` does not manage to complete the build (specialization) within a week on Coq 8.7.0
>
> —BoringSSL Fiat README file

More backends, including verified compilers:

> […] backend to our Bedrock systems-programming language in Coq, which, unlike the original C backend, has a proof of soundness.
>
> —Adam Chlipala, in an email

Next class:

Conclusion

To do:

- ☐ Course feedback
- ☐ Final presentations
- ☐ Final projects