Simple High-Level Code For Cryptographic Arithmetic – With Proofs, Without Compromises

Andres Erbsen, Jade Philipoom, Jason Gross,

Robert Sloan, Adam Chlipala

MIT CSAIL

`github.com/mit-plv/fiat-crypto`

Finite Field Arithmetic

- Important for elliptic-curve cryptography
  - TLS, Signal, SSH...
- Performance-sensitive
- Hand-coded for each modulus, CPU word size
  - Widely implemented: P-256 and Curve25519
- Persistent concerns about correctness

In [OpenSSL multiplication modulo P-256] there are a number of comments saying "doesn't overflow". Unfortunately, they aren't correct.

In [OpenSSL multiplication modulo P-256] there are a number of comments saying "doesn't overflow". Unfortunately, they aren't correct.

Got math wrong :-(. [fix] attached.

[unclear if existing attacks can exploit this]

In [OpenSSL multiplication modulo P-256] there are a number of comments saying "doesn't overflow". Unfortunately, they aren't correct.

Got math wrong :-(. [fix] attached.

[unclear if existing attacks can exploit this]

[still wrong; counterexample]

In [OpenSSL multiplication modulo P-256] there are a number of comments saying "doesn't overflow". Unfortunately, they aren't correct.

Got math wrong :-(. [fix] attached.

[unclear if existing attacks can exploit this]

[still wrong; counterexample]

[...]

Attached. A little bit worse performance on some CPUs

In [OpenSSL multiplication modulo P-256] there are a number of comments saying "doesn't overflow". Unfortunately, they aren't correct.

Got math wrong :-(. [fix] attached.

[unclear if existing attacks can exploit this]

[still wrong; counterexample]

[...]

Attached. A little bit worse performance on some CPUs

It's good for ~6B random tests. [...] I think we can safely say that there aren't any low-hanging bugs left.

Our Library

- Reusable, parametric implementations
- Automatically specialized to parameter values
- One computer-checkable correctness proof

Our Library

- Reusable, parametric implementations
- Automatically specialized to parameter values
- One computer-checkable correctness proof
- Deployed to billions of users with BoringSSL

demo

push-button code generation
(Curve25519 for 32-bit CPUs)

```
fiat@ashryn ~/fiat-crypto-demo (git)-[master] % unsaturated_solinas
```

```
fiat@ashryn ~/fiat-crypto-demo (git)-[master] % unsaturated_solinas
 "curve25519"
```

```
fiat@ashryn ~/fiat-crypto-demo (git)-[master] % unsaturated_solinas
 "curve25519" 10
```

```
fiat@ashryn ~/fiat-crypto-demo (git)-[master] % unsaturated_solinas
 "curve25519" 10 '2^255' 1,19
```

```
fiat@ashryn ~/fiat-crypto-demo (git)-[master] % unsaturated_solinas
 "curve25519" 10 '2^255' 1,19 32
```

```
fiat@ashryn ~/fiat-crypto-demo (git)-[master] % unsaturated_solinas
 "curve25519" 10 '2^255' 1,19 32 carry_mul
```

```
/* Autogenerated */
/* curve description: curve25519 */
/* requested operations: carry_mul */
/* n = 10 (from "10") */
/* s = 0x8000000000000000000000000000000000000000000000000000000000000000 (from "2^255") */
/* c = [(1, 19)] (from "1,19") */
/* machine_wordsize = 32 (from "32") */

/* Computed values: */
/* carry_chain = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1] */

#include <stdint.h>


/*
 * The function fiat_curve25519_carry_mul multiplies two field elements and reduces the result.
 * Postconditions:
 *   eval out1 mod m = (eval arg1 * eval arg2) mod m
 *
 * Input Bounds:
 *   arg1: [[0x0 ~> 0xd333332], [0x0 ~> 0x6999999], [0x0 ~> 0xd333332], [0x0 ~> 0x6999999], [0x0 ~> 0xd333332],
 [0x0 ~> 0x6999999], [0x0 ~> 0xd333332], [0x0 ~> 0x6999999], [0x0 ~> 0xd333332], [0x0 ~> 0x6999999]]
 *   arg2: [[0x0 ~> 0xd333332], [0x0 ~> 0x6999999], [0x0 ~> 0xd333332], [0x0 ~> 0x6999999], [0x0 ~> 0xd333332],
 [0x0 ~> 0x6999999], [0x0 ~> 0xd333332], [0x0 ~> 0x6999999], [0x0 ~> 0xd333332], [0x0 ~> 0x6999999]]
 * Output Bounds:
 *   out1: [[0x0 ~> 0x4666666], [0x0 ~> 0x2333333], [0x0 ~> 0x4666666], [0x0 ~> 0x2333333], [0x0 ~> 0x4666666],
 [0x0 ~> 0x2333333], [0x0 ~> 0x4666666], [0x0 ~> 0x2333333], [0x0 ~> 0x4666666], [0x0 ~> 0x2333333]]
 */
static void fiat_curve25519_carry_mul(uint32_t out1[10], const uint32_t arg1[10], const uint32_t arg2[10]) {
  uint64_t x1 = ((uint64_t)(arg1[9]) * ((arg2[9]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x2 = ((uint64_t)(arg1[9]) * ((arg2[8]) * (uint32_t)UINT8_C(0x13)));
:
```

```
  uint64_t x3 = ((uint64_t)(arg1[9]) * ((arg2[7]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x4 = ((uint64_t)(arg1[9]) * ((arg2[6]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x5 = ((uint64_t)(arg1[9]) * ((arg2[5]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x6 = ((uint64_t)(arg1[9]) * ((arg2[4]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x7 = ((uint64_t)(arg1[9]) * ((arg2[3]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x8 = ((uint64_t)(arg1[9]) * ((arg2[2]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x9 = ((uint64_t)(arg1[9]) * ((arg2[1]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x10 = ((uint64_t)(arg1[8]) * ((arg2[9]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x11 = ((uint64_t)(arg1[8]) * ((arg2[8]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x12 = ((uint64_t)(arg1[8]) * ((arg2[7]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x13 = ((uint64_t)(arg1[8]) * ((arg2[6]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x14 = ((uint64_t)(arg1[8]) * ((arg2[5]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x15 = ((uint64_t)(arg1[8]) * ((arg2[4]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x16 = ((uint64_t)(arg1[8]) * ((arg2[3]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x17 = ((uint64_t)(arg1[8]) * ((arg2[2]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x18 = ((uint64_t)(arg1[7]) * ((arg2[9]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x19 = ((uint64_t)(arg1[7]) * ((arg2[8]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x20 = ((uint64_t)(arg1[7]) * ((arg2[7]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x21 = ((uint64_t)(arg1[7]) * ((arg2[6]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x22 = ((uint64_t)(arg1[7]) * ((arg2[5]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x23 = ((uint64_t)(arg1[7]) * ((arg2[4]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x24 = ((uint64_t)(arg1[7]) * ((arg2[3]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x25 = ((uint64_t)(arg1[6]) * ((arg2[9]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x26 = ((uint64_t)(arg1[6]) * ((arg2[8]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x27 = ((uint64_t)(arg1[6]) * ((arg2[7]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x28 = ((uint64_t)(arg1[6]) * ((arg2[6]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x29 = ((uint64_t)(arg1[6]) * ((arg2[5]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x30 = ((uint64_t)(arg1[6]) * ((arg2[4]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x31 = ((uint64_t)(arg1[5]) * ((arg2[9]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x32 = ((uint64_t)(arg1[5]) * ((arg2[8]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x33 = ((uint64_t)(arg1[5]) * ((arg2[7]) * ((uint32_t)0x2 * UINT8_C(0x13))));
:
```

```
  uint64_t x34 = ((uint64_t)(arg1[5]) * ((arg2[6]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x35 = ((uint64_t)(arg1[5]) * ((arg2[5]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x36 = ((uint64_t)(arg1[4]) * ((arg2[9]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x37 = ((uint64_t)(arg1[4]) * ((arg2[8]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x38 = ((uint64_t)(arg1[4]) * ((arg2[7]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x39 = ((uint64_t)(arg1[4]) * ((arg2[6]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x40 = ((uint64_t)(arg1[3]) * ((arg2[9]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x41 = ((uint64_t)(arg1[3]) * ((arg2[8]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x42 = ((uint64_t)(arg1[3]) * ((arg2[7]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x43 = ((uint64_t)(arg1[2]) * ((arg2[9]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x44 = ((uint64_t)(arg1[2]) * ((arg2[8]) * (uint32_t)UINT8_C(0x13)));
  uint64_t x45 = ((uint64_t)(arg1[1]) * ((arg2[9]) * ((uint32_t)0x2 * UINT8_C(0x13))));
  uint64_t x46 = ((uint64_t)(arg1[9]) * (arg2[0]));
  uint64_t x47 = ((uint64_t)(arg1[8]) * (arg2[1]));
  uint64_t x48 = ((uint64_t)(arg1[8]) * (arg2[0]));
  uint64_t x49 = ((uint64_t)(arg1[7]) * (arg2[2]));
  uint64_t x50 = ((uint64_t)(arg1[7]) * ((arg2[1]) * (uint32_t)0x2));
  uint64_t x51 = ((uint64_t)(arg1[7]) * (arg2[0]));
  uint64_t x52 = ((uint64_t)(arg1[6]) * (arg2[3]));
  uint64_t x53 = ((uint64_t)(arg1[6]) * (arg2[2]));
  uint64_t x54 = ((uint64_t)(arg1[6]) * (arg2[1]));
  uint64_t x55 = ((uint64_t)(arg1[6]) * (arg2[0]));
  uint64_t x56 = ((uint64_t)(arg1[5]) * (arg2[4]));
  uint64_t x57 = ((uint64_t)(arg1[5]) * ((arg2[3]) * (uint32_t)0x2));
  uint64_t x58 = ((uint64_t)(arg1[5]) * (arg2[2]));
  uint64_t x59 = ((uint64_t)(arg1[5]) * ((arg2[1]) * (uint32_t)0x2));
  uint64_t x60 = ((uint64_t)(arg1[5]) * (arg2[0]));
  uint64_t x61 = ((uint64_t)(arg1[4]) * (arg2[5]));
  uint64_t x62 = ((uint64_t)(arg1[4]) * (arg2[4]));
  uint64_t x63 = ((uint64_t)(arg1[4]) * (arg2[3]));
  uint64_t x64 = ((uint64_t)(arg1[4]) * (arg2[2]));
:
```

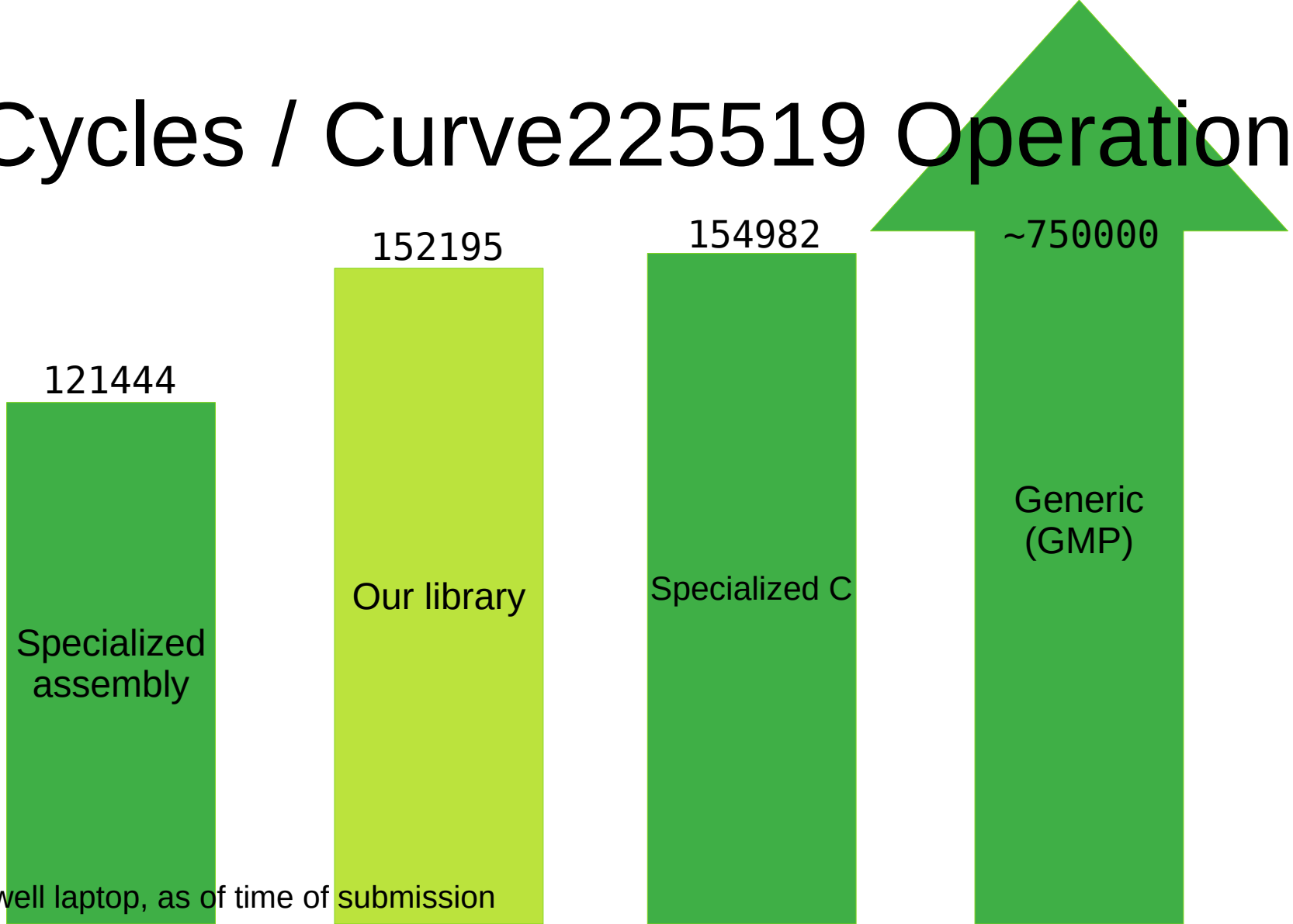
```
  uint64_t x65 = ((uint64_t)(arg1[4]) * (arg2[1]));
  uint64_t x66 = ((uint64_t)(arg1[4]) * (arg2[0]));
  uint64_t x67 = ((uint64_t)(arg1[3]) * (arg2[6]));
  uint64_t x68 = ((uint64_t)(arg1[3]) * ((arg2[5]) * (uint32_t)0x2));
  uint64_t x69 = ((uint64_t)(arg1[3]) * (arg2[4]));
  uint64_t x70 = ((uint64_t)(arg1[3]) * ((arg2[3]) * (uint32_t)0x2));
  uint64_t x71 = ((uint64_t)(arg1[3]) * (arg2[2]));
  uint64_t x72 = ((uint64_t)(arg1[3]) * ((arg2[1]) * (uint32_t)0x2));
  uint64_t x73 = ((uint64_t)(arg1[3]) * (arg2[0]));
  uint64_t x74 = ((uint64_t)(arg1[2]) * (arg2[7]));
  uint64_t x75 = ((uint64_t)(arg1[2]) * (arg2[6]));
  uint64_t x76 = ((uint64_t)(arg1[2]) * (arg2[5]));
  uint64_t x77 = ((uint64_t)(arg1[2]) * (arg2[4]));
  uint64_t x78 = ((uint64_t)(arg1[2]) * (arg2[3]));
  uint64_t x79 = ((uint64_t)(arg1[2]) * (arg2[2]));
  uint64_t x80 = ((uint64_t)(arg1[2]) * (arg2[1]));
  uint64_t x81 = ((uint64_t)(arg1[2]) * (arg2[0]));
  uint64_t x82 = ((uint64_t)(arg1[1]) * (arg2[8]));
  uint64_t x83 = ((uint64_t)(arg1[1]) * ((arg2[7]) * (uint32_t)0x2));
  uint64_t x84 = ((uint64_t)(arg1[1]) * (arg2[6]));
  uint64_t x85 = ((uint64_t)(arg1[1]) * ((arg2[5]) * (uint32_t)0x2));
  uint64_t x86 = ((uint64_t)(arg1[1]) * (arg2[4]));
  uint64_t x87 = ((uint64_t)(arg1[1]) * ((arg2[3]) * (uint32_t)0x2));
  uint64_t x88 = ((uint64_t)(arg1[1]) * (arg2[2]));
  uint64_t x89 = ((uint64_t)(arg1[1]) * ((arg2[1]) * (uint32_t)0x2));
  uint64_t x90 = ((uint64_t)(arg1[1]) * (arg2[0]));
  uint64_t x91 = ((uint64_t)(arg1[0]) * (arg2[9]));
  uint64_t x92 = ((uint64_t)(arg1[0]) * (arg2[8]));
  uint64_t x93 = ((uint64_t)(arg1[0]) * (arg2[7]));
  uint64_t x94 = ((uint64_t)(arg1[0]) * (arg2[6]));
  uint64_t x95 = ((uint64_t)(arg1[0]) * (arg2[5]));
:
```

```
  uint64_t x96 = ((uint64_t)(arg1[0]) * (arg2[4]));
  uint64_t x97 = ((uint64_t)(arg1[0]) * (arg2[3]));
  uint64_t x98 = ((uint64_t)(arg1[0]) * (arg2[2]));
  uint64_t x99 = ((uint64_t)(arg1[0]) * (arg2[1]));
  uint64_t x100 = ((uint64_t)(arg1[0]) * (arg2[0]));
  uint64_t x101 = (x100 + (x45 + (x44 + (x42 + (x39 + (x35 + (x30 + (x24 + (x17 + x9)))))))));
  uint64_t x102 = (x101 >> 26);
  uint32_t x103 = (uint32_t)(x101 & UINT32_C(0x3ffffff));
  uint64_t x104 = (x91 + (x82 + (x74 + (x67 + (x61 + (x56 + (x52 + (x49 + (x47 + x46)))))))));
  uint64_t x105 = (x92 + (x83 + (x75 + (x68 + (x62 + (x57 + (x53 + (x50 + (x48 + x1)))))))));
  uint64_t x106 = (x93 + (x84 + (x76 + (x69 + (x63 + (x58 + (x54 + (x51 + (x10 + x2)))))))));
  uint64_t x107 = (x94 + (x85 + (x77 + (x70 + (x64 + (x59 + (x55 + (x18 + (x11 + x3)))))))));
  uint64_t x108 = (x95 + (x86 + (x78 + (x71 + (x65 + (x60 + (x25 + (x19 + (x12 + x4)))))))));
  uint64_t x109 = (x96 + (x87 + (x79 + (x72 + (x66 + (x31 + (x26 + (x20 + (x13 + x5)))))))));
  uint64_t x110 = (x97 + (x88 + (x80 + (x73 + (x36 + (x32 + (x27 + (x21 + (x14 + x6)))))))));
  uint64_t x111 = (x98 + (x89 + (x81 + (x40 + (x37 + (x33 + (x28 + (x22 + (x15 + x7)))))))));
  uint64_t x112 = (x99 + (x90 + (x43 + (x41 + (x38 + (x34 + (x29 + (x23 + (x16 + x8)))))))));
  uint64_t x113 = (x102 + x112);
  uint64_t x114 = (x113 >> 25);
  uint32_t x115 = (uint32_t)(x113 & UINT32_C(0x1ffffff));
  uint64_t x116 = (x114 + x111);
  uint64_t x117 = (x116 >> 26);
  uint32_t x118 = (uint32_t)(x116 & UINT32_C(0x3ffffff));
  uint64_t x119 = (x117 + x110);
  uint64_t x120 = (x119 >> 25);
  uint32_t x121 = (uint32_t)(x119 & UINT32_C(0x1ffffff));
  uint64_t x122 = (x120 + x109);
  uint64_t x123 = (x122 >> 26);
  uint32_t x124 = (uint32_t)(x122 & UINT32_C(0x3ffffff));
  uint64_t x125 = (x123 + x108);
  uint64_t x126 = (x125 >> 25);
:
```

```
  uint64_t x129 = (x128 >> 26);
  uint32_t x130 = (uint32_t)(x128 & UINT32_C(0x3ffffff));
  uint64_t x131 = (x129 + x106);
  uint64_t x132 = (x131 >> 25);
  uint32_t x133 = (uint32_t)(x131 & UINT32_C(0x1ffffff));
  uint64_t x134 = (x132 + x105);
  uint64_t x135 = (x134 >> 26);
  uint32_t x136 = (uint32_t)(x134 & UINT32_C(0x3ffffff));
  uint64_t x137 = (x135 + x104);
  uint64_t x138 = (x137 >> 25);
  uint32_t x139 = (uint32_t)(x137 & UINT32_C(0x1ffffff));
  uint64_t x140 = (x138 * (uint64_t)UINT8_C(0x13));
  uint64_t x141 = (x103 + x140);
  uint32_t x142 = (uint32_t)(x141 >> 26);
  uint32_t x143 = (uint32_t)(x141 & UINT32_C(0x3ffffff));
  uint32_t x144 = (x142 + x115);
  uint32_t x145 = (x144 >> 25);
  uint32_t x146 = (x144 & UINT32_C(0x1ffffff));
  uint32_t x147 = (x145 + x118);
  out1[0] = x143;
  out1[1] = x146;
  out1[2] = x147;
  out1[3] = x121;
  out1[4] = x124;
  out1[5] = x127;
  out1[6] = x130;
  out1[7] = x133;
  out1[8] = x136;
  out1[9] = x139;
}

(END)
```

Cycles / Curve225519 Operation



on a Broadwell laptop, as of time of submission

Modulus-Specific Representations

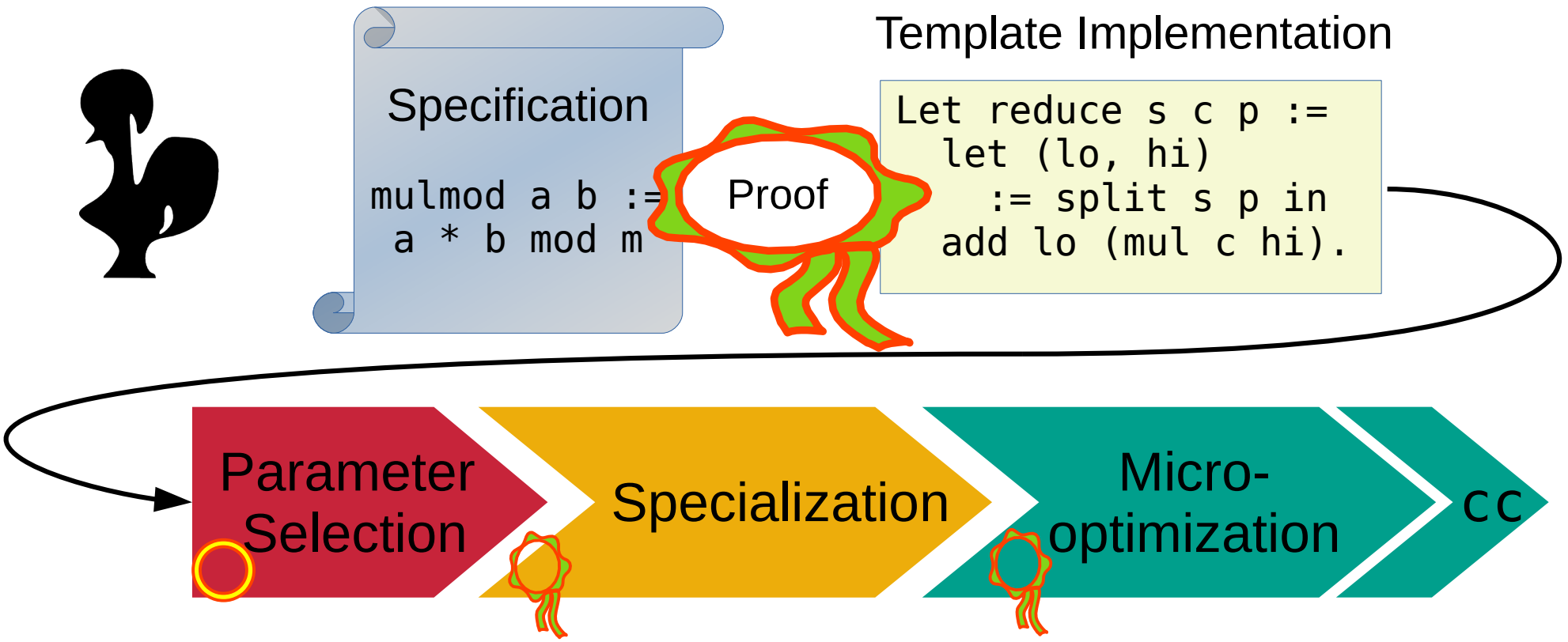
- Important driver of specialized implementation
- Break one field element into multiple digits

Modulus-Specific Representations

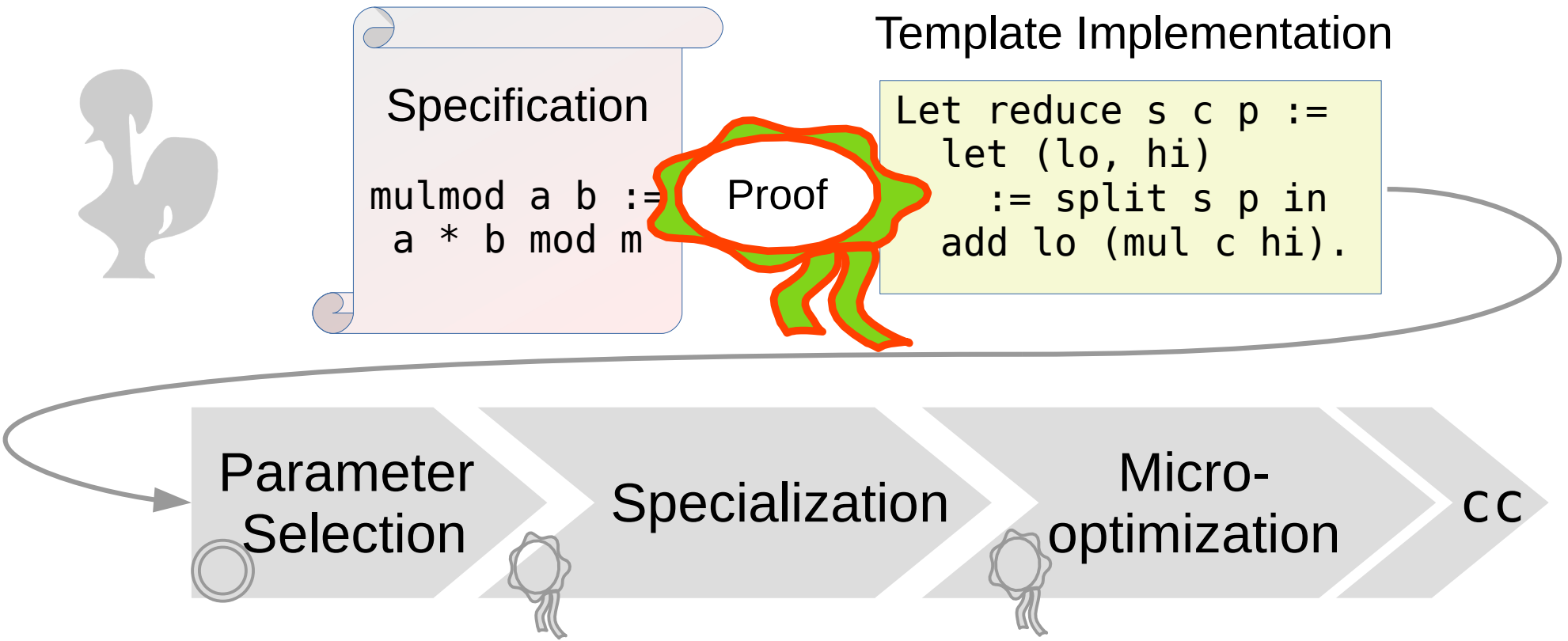
- Important driver of specialized implementation
- Break one field element into multiple digits
  - mod $2^{255}-19$: $x = 2^{256}\cdot x_4 + 2^{196}\cdot x_3 + 2^{128}\cdot x_2 + 2^{64}\cdot x_1 + x_0$
  - mod $2^{127}-1$: $x = 2^{127}\cdot x_3 + 2^{85}\cdot x_2 + 2^{43}\cdot x_1 + x_0$

    x_2: 42 bits; x_1: 42; x_0: 43
- Key challenge: generalizing algorithms across representations

Our Algorithm-Centric Workflow



Focus of This Talk



Compile-time Associational Representation

- $876 = 8 \cdot 10^2 + 7 \cdot 10 + 6 \cdot 1$

- `Let example := [(10^2, 8); (10, 7); (1,6)].`
- `Let eval ls := sum (map (fun '(a,x)=> a*x) ls).`

Compile-time Associational Representation

- $876 = 8 \cdot 10^2 + 7 \cdot 10 + 6 \cdot 1$
- `Let example := [(10^2, 8); (10, 7); (1,6)].`
- `Let eval ls := sum (map (fun '(a,x)=> a*x) ls).`
- $876 = 4 \cdot 200 + 5 \cdot 10 + 1 \cdot 10 + 16 \cdot 1$
- Later: conversion to standard representation

Example: Schoolbook Multiplication

```
Definition mul (p q : list (Z*Z)) : list (Z*Z) :=
  concat (map (fun '(a, x) =>
    map (fun '(b, y) =>
      (a*b, x*y))
  q) p).
```

a = [(100,3); (10,2); (1,1)]
b = [(10,7); (1,6)]

	3	2	1	
	18	12	6	6
21	14	7		7

ab = [(100, 18); (10, 12); (1, 6); (1000,21);(100, 14);(10,7)]

```
Lemma eval_map_mul a x q: eval (map (fun '(b, y)=>(a*b, x*y)) q)=a*x*eval q.
Proof. induction q; push; nsatz. Qed.
Hint Rewrite eval_map_mul : push.
Lemma eval_mul : forall p q, eval (mul p q) = eval p * eval q.
Proof. intros; induction p; cbv [mul]; push; nsatz. Qed.
```

But Are These the Implementations We're Looking For?

- Ahead-of-time specialization for performance!
- List lengths, digit weights are compile-time
- Evaluate, partially (grab a coffee while trying this at home):
  - `cbv -[blacklist] in (mul [(1,x);..] ..)`

Example Arithmetic Code

```
Definition mul (p q : list (Z*Z)) : list (Z*Z) :=
  concat (map (fun '(a, x) =>
    map (fun '(b, y) =>
      (a*b, x*y))
  q) p).
```

Annotated run-time operation

```
Lemma eval_map_mul a x q: eval (map (fun '(b, y)=>(a*b, x*y)) q)=a*x*eval q.
Proof. induction q; push; nsatz. Qed.
Hint Rewrite eval_map_mul : push.
Lemma eval_mul : forall p q, eval (mul p q) = eval p * eval q.
Proof. intros; induction p; cbv [mul]; push; nsatz. Qed.
```

Partial Evaluation Example

```
Eval cbv -[runtime_mul] in
    fun a0 a1 a2 b0 b1 b2 =>
      mul [(1, a0); (10, a1); (100, a2)]
          [(1, b0); (10, b1); (100, b2)].

= fun a0 a1 a2 b0 b1 b2 =>
   [  (1, a0*b0);   (10, a0*b1);   (100, a0*b2);
     (10, a1*b0);  (100, a1*b1);  (1000, a1*b2);
    (100, a2*b0); (1000, a2*b1); (10000, a2*b2)]
```

- Almost there; need to deduplicate the output list!

```
fun a0 a1 a2 b0 b1 b2 =>
   [(1, a0*b0); (10, a0*b1 + a1*b0); (100, a0*b2+a1*b1+a2*b0);
         (1000, a1*b2 + a2*b1); (10000, a2*b2)]
```

Deduplication to Positional Repr.

- Run-time representation: fixed-length array
  → Assign each term to the correct slot

Deduplication to Positional Repr.

- Run-time representation: fixed-length array
  - → Assign each term to the correct slot
- With slots for $[1, 10, 100]$, where does $(500, x)$ go?
  - Disallow? But proofs
  - Useful to handle for mixed-radix representations

Deduplication to Positional Repr.

- Run-time representation: fixed-length array
  - → Assign each term to the correct slot
- With slots for $[1, 10, 100]$, where does $(500, x)$ go?
  - Disallow? But proofs
  - Useful to handle for mixed-radix representations
- Verdict: to place $(500,x)$, add $5 \cdot x$ to the 100s

Three Tricks for Modular Reduction

- Pseudo-Mersenne – $m = 2^{nt} - c$ (c small)
- Solinas – $m = 2^{nt} - c$ (c sparse)
- Mixed-radix – $m = 2^{n(t/l)} - c$
  - Curve25519 on 32-bit, 2004

Three Tricks for Modular Reduction

- Pseudo-Mersenne – $m = 2^{nt} - c$ (c small)
- Solinas – $m = 2^{nt} - c$ (c sparse)
- Mixed-radix – $m = 2^{n(t/l)} - c$
  - Curve25519 on 32-bit, 2004
- One natural implementation will yield all 3!
- Key commonality: weight 2^k, $2^k \bmod m = c$

Our Modular Multiplication

- Associational representation for inputs **and c**
- Multiply
- Replace each $(2^k \cdot b, x)$ with `mul c [(b, x)]`
- Convert to positional with the desired slots

Our Modular Multiplication

- Associational representation for inputs **and c**
- Multiply
- Replace each $(2^k \cdot b, x)$ with `mul c [(b, x)]`
- Convert to positional with the desired slots
  - Some $(c \cdot b, x)$ become $(b, c \cdot x)$
- Always correct, fast for clever choices of c, k

Our Modular Multiplication

```
Eval cbv -[runtime_mul runtime_add] in
    (mulmod (n:=10) w (2 ^ 255) [(1, 19)]
       (f9, f8, f7, f6, f5, f4, f3, f2, f1, f0)
       (g9, g8, g7, g6, g5, g4, g3, g2, g1, g0)).
ring_simplify_subterms.
```

$(19 \cdot 2^{51}, f8 \cdot g9) = (2^{51}, 19 \cdot f8 \cdot g9)$

```
(* ?fg =
 (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)
*)
```

Our Modular Multiplication

```
Eval cbv -[runtime_mul runtime_add] in
    (mulmod (n:=10) w (2 ^ 255) [(1, 19)]
       (f9, f8, f7, f6, f5, f4, f3, f2, f1, f0)
       (g9, g8, g7, g6, g5, g4, g3, g2, g1, g0)).
ring_simplify_subterms.
```

$(2^{26}, f1) \cdot (2^{26}, g1)$
$= (2^{52}, f1 \cdot g1)$
$= (2^{51}, 2 \cdot f1 \cdot g2)$

```
(* ?fg =
 (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)
*)
```

Range Analysis

```
0 ≤ f0, f2, f4, f6, f8    ≤  1.25*2^26  (uint32_t)
0 ≤   f1, f3, f5, f7, f9  ≤  1.25*2^25  (uint32_t)
0 ≤ g0, g2, g4, g6, g8    ≤  1.25*2^26  (uint32_t)
0 ≤   g1, g3, g5, g7, g9  ≤  1.25*2^25  (uint32_t)
```

```
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
```

≤2^52

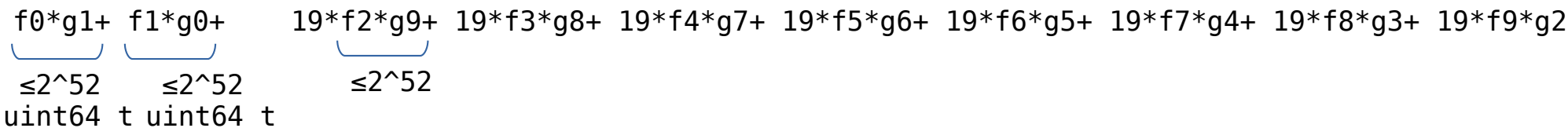
Range Analysis

```
0 ≤ f0, f2, f4, f6, f8    ≤  1.25*2^26  (uint32_t)
0 ≤   f1, f3, f5, f7, f9  ≤  1.25*2^25  (uint32_t)
0 ≤ g0, g2, g4, g6, g8    ≤  1.25*2^26  (uint32_t)
0 ≤   g1, g3, g5, g7, g9  ≤  1.25*2^25  (uint32_t)
```

```
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
 ≤2^52
uint64_t
```

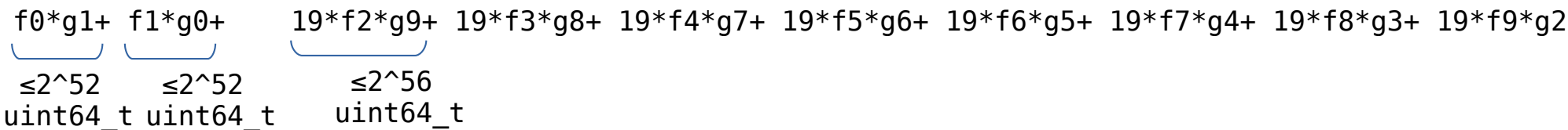
Range Analysis

```
 0 ≤ f0, f2, f4, f6, f8    ≤  1.25*2^26  (uint32_t)
 0 ≤   f1, f3, f5, f7, f9  ≤  1.25*2^25  (uint32_t)
 0 ≤ g0, g2, g4, g6, g8    ≤  1.25*2^26  (uint32_t)
 0 ≤   g1, g3, g5, g7, g9  ≤  1.25*2^25  (uint32_t)
```



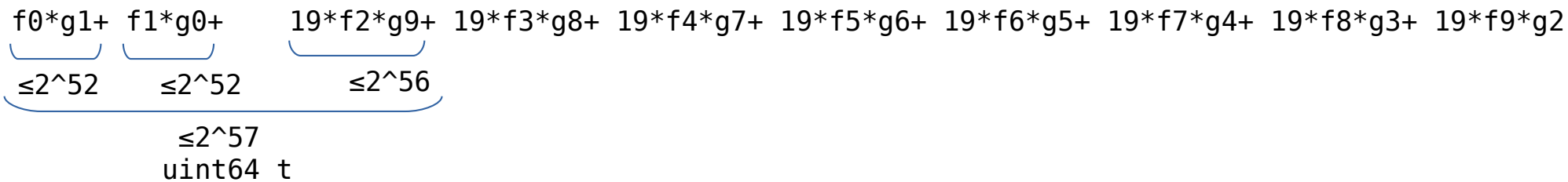
Range Analysis

```
0 ≤ f0, f2, f4, f6, f8    ≤  1.25*2^26  (uint32_t)
0 ≤   f1, f3, f5, f7, f9  ≤  1.25*2^25  (uint32_t)
0 ≤ g0, g2, g4, g6, g8    ≤  1.25*2^26  (uint32_t)
0 ≤   g1, g3, g5, g7, g9  ≤  1.25*2^25  (uint32_t)
```



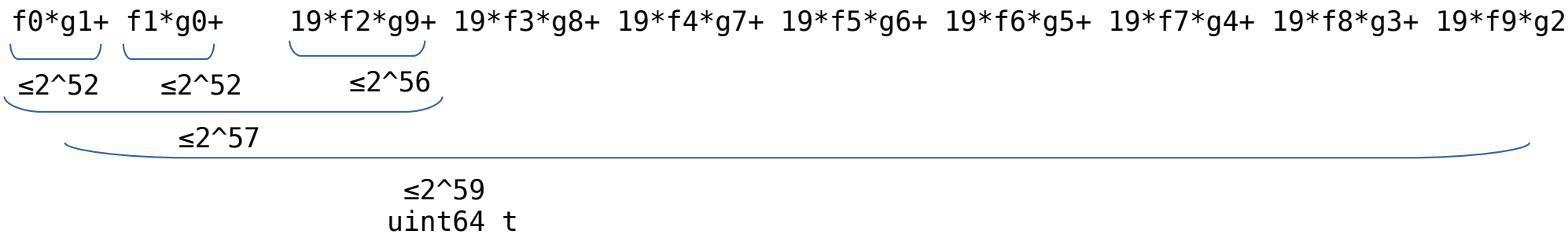
Range Analysis

```
0 ≤ f0, f2, f4, f6, f8    ≤  1.25*2^26  (uint32_t)
0 ≤   f1, f3, f5, f7, f9  ≤  1.25*2^25  (uint32_t)
0 ≤ g0, g2, g4, g6, g8    ≤  1.25*2^26  (uint32_t)
0 ≤   g1, g3, g5, g7, g9  ≤  1.25*2^25  (uint32_t)
```



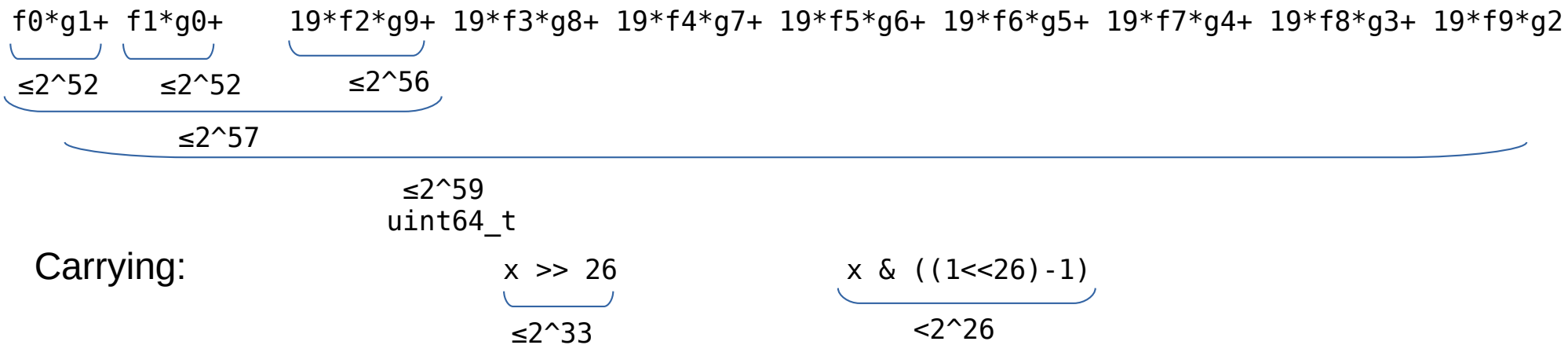
Range Analysis

```
0 ≤ f0, f2, f4, f6, f8    ≤  1.25*2^26  (uint32_t)
0 ≤   f1, f3, f5, f7, f9  ≤  1.25*2^25  (uint32_t)
0 ≤ g0, g2, g4, g6, g8    ≤  1.25*2^26  (uint32_t)
0 ≤   g1, g3, g5, g7, g9  ≤  1.25*2^25  (uint32_t)
```



Range Analysis

```
0 ≤ f0, f2, f4, f6, f8    ≤  1.25*2^26  (uint32_t)
0 ≤   f1, f3, f5, f7, f9  ≤  1.25*2^25  (uint32_t)
0 ≤ g0, g2, g4, g6, g8    ≤  1.25*2^26  (uint32_t)
0 ≤   g1, g3, g5, g7, g9  ≤  1.25*2^25  (uint32_t)
```



Reflections: Timeline

- Many months: experimentation with other reprs.
- One evening: associational repr, code, proofs
- Many months: engineering Coq partial reduction
- A couple of months: range analysis compiler, proof

- Several days: repr. design for add-with-carry operations
- Several days: figuring out Montgomery reduction proof
- One evening: proving Montgomery red. after refactor

Reflections: Was It Worth It?

- Relatively easy proofs, no technical surprises
- Many primes with one implementation
- We think our implementations are instructive

- Waiting for Coq to run out of memory (or not) → :(
- Performance is limited by C compiler quality
  - Translation validation for human-compiled variants?

thanks

`github.com/mit-plv/fiat-crypto`