Performance Engineering of Proof-Based Software Systems at Scale

Jason Gross

Ph.D. Defense

MIT CSAIL

Takeaways

- Opportunity: Automate Verification to Enable Innovation
- Big Problem: Asymptotic Performance
- My Contribution: Reflective Partial Evaluation
- Important Next Steps

Fiat Crypto

- Joint work with Andres Erbsen, Jade Philipoom, Adam Chlipala, et al
- Used in *majority* of secure connections from web browsers

ive.mozilla.org/foundation/identity-guidelines/firefox
HTTPS image modified from image by Sean MacEntee, CC BY 2.0, via Wikimedia Commons
Chrome logo from https://www.logo.wine/logo/Google_Chrome ©2018 Google LLC All rights reserved. Chrome is a trademark of Google LLC.
Go Logo By The Go Authors - https://blog.golang.org/go-brand, Public Domain, https://commons.wikimedia.org/w/index.php?curid=82371649
File:Logo of WireGuard.svg. (2020, April 21). Wikimedia Commons, the free media repository. Retrieved 23:20, November 28, 2020 from Wikimedia Commons
Libra logo from By Libra Association - https://libra.org/, Public Domain, https://commons.wikimedia.org/w/index.php?curid=79808006

Innovation with Cryptography

"Better! Faster! Cheaper!"

- Hedging against more powerful attackers
- More mathematical security
- Reduce costs (server & user)

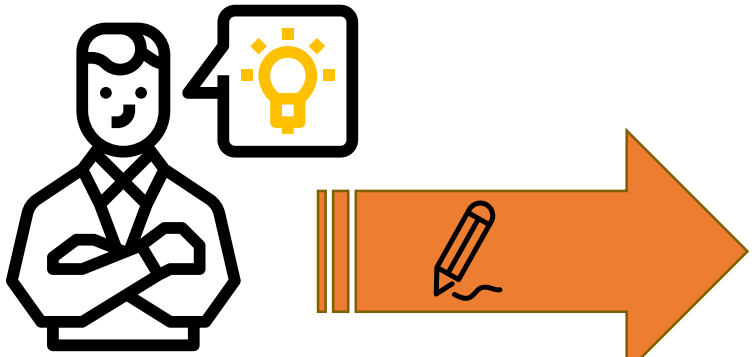
Mathematical Specification:
$(a \cdot b) \bmod p$

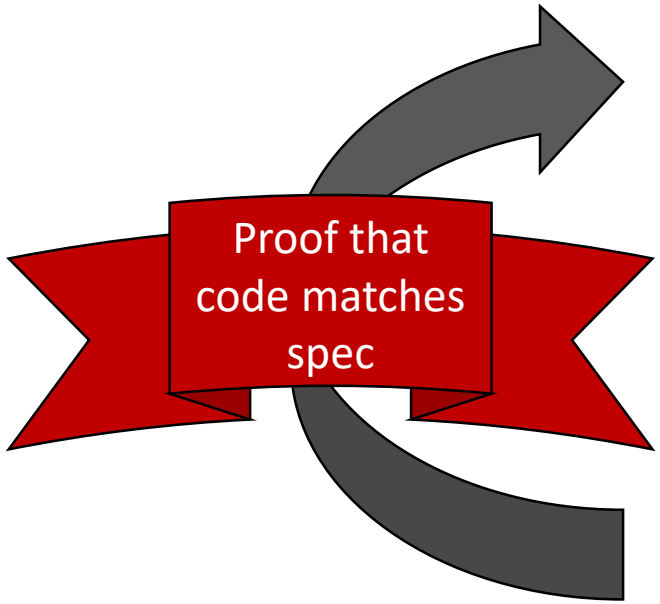
```
static inline void force_inline
fmul(felem output, const felem in2, const felem in) {
uint128_t t[5];
limb r0,r1,r2,r3,r4,s0,s1,s2,s3,s4,c;
r0 = in[0];
r1 = in[1];
r2 = in[2];
r3 = in[3];
r4 = in[4];
s0 = in2[0];
s1 = in2[1];
s2 = in2[2];
s3 = in2[3];
s4 = in2[4];
t[0] = ((uint128_t) r0) * s0;
t[1] = ((uint128_t) r0) * s1 + ((uint128_t) r1) * s0;
t[2] = ((uint128_t) r0) * s2 + ((uint128_t) r2) * s0 + ((uint128_t) r1) * s1;
t[3] = ((uint128_t) r0) * s3 + ((uint128_t) r3) * s0 + ((uint128_t) r1) * s2 + ((uint128_t) r2) * s1;
t[4] = ((uint128_t) r0) * s4 + ((uint128_t) r4) * s0 + ((uint128_t) r3) * s1 + ((uint128_t) r1) * s3 + ((uint128_t) r2) * s2;
r4 *= 19;
r1 *= 19;
r2 *= 19;
r3 *= 19;
t[0] += ((uint128_t) r4) * s1 + ((uint128_t) r1) * s4 + ((uint128_t) r2) * s3 + ((uint128_t) r3) * s2;
t[1] += ((uint128_t) r4) * s2 + ((uint128_t) r2) * s4 + ((uint128_t) r3) * s3;
t[2] += ((uint128_t) r4) * s3 + ((uint128_t) r3) * s4;
t[3] += ((uint128_t) r4) * s4;
r0 = (limb)t[0] & 0x7ffffffffffff; c = (limb)(t[0] >> 51);
t[1] += c; r1 = (limb)t[1] & 0x7ffffffffffff; c = (limb)(t[1] >> 51);
t[2] += c; r2 = (limb)t[2] & 0x7ffffffffffff; c = (limb)(t[2] >> 51);
t[3] += c; r3 = (limb)t[3] & 0x7ffffffffffff; c = (limb)(t[3] >> 51);
t[4] += c; r4 = (limb)t[4] & 0x7ffffffffffff; c = (limb)(t[4] >> 51);
r0 += c * 19; c = r0 >> 51; r0 = r0 & 0x7ffffffffffff;
r1 += c; c = r1 >> 51; r1 = r1 & 0x7ffffffffffff;
r2 += c;
output[0] = r0;
output[1] = r1;
output[2] = r2;
output[3] = r3;
output[4] = r4;
}
```

"Don't touch it; it works!"

- Lots of room for error
- Enormous cost of error
- Hard to find errors

The Promise of Verification



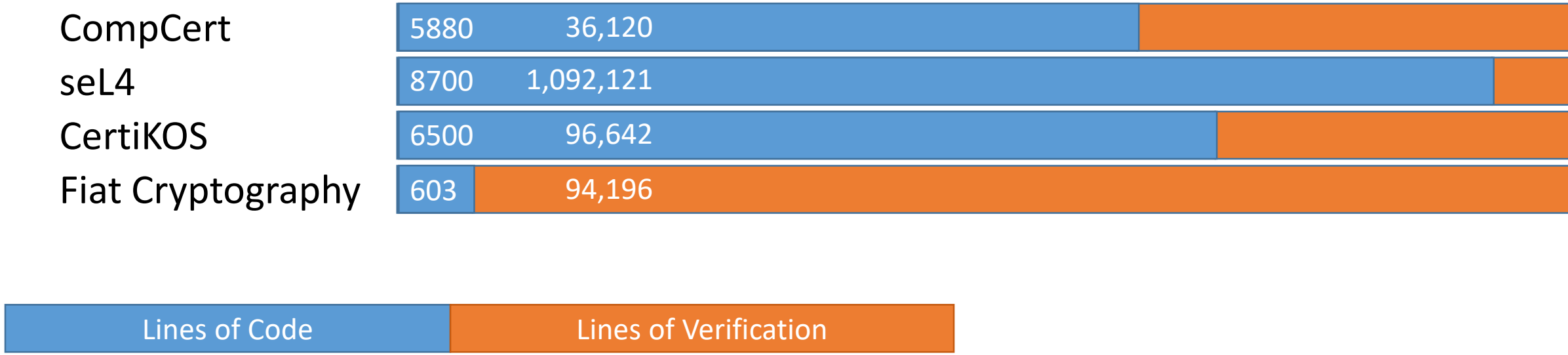


Mathematical Specification:
$(a \cdot b) \bmod p$

```
static inline void force_inline
fmul(felem output, const felem in2, const felem in) {
uint128_t t[5];
limb r0,r1,r2,r3,r4,s0,s1,s2,s3,s4,c;
r0 = in[0];
r1 = in[1];
r2 = in[2];
r3 = in[3];
r4 = in[4];
s0 = in2[0];
s1 = in2[1];
s2 = in2[2];
s3 = in2[3];
s4 = in2[4];
t[0] = ((uint128_t) r0) * s0;
t[1] = ((uint128_t) r0) * s1 + ((uint128_t) r1) * s0;
t[2] = ((uint128_t) r0) * s2 + ((uint128_t) r2) * s0 + ((uint128_t) r1) * s1;
t[3] = ((uint128_t) r0) * s3 + ((uint128_t) r3) * s0 + ((uint128_t) r1) * s2 + ((uint128_t) r2) * s1;
t[4] = ((uint128_t) r0) * s4 + ((uint128_t) r4) * s0 + ((uint128_t) r3) * s1 + ((uint128_t) r1) * s3 + ((uint128_t) r2) * s2;
r4 *= 19;
r1 *= 19;
r2 *= 19;
r3 *= 19;
t[0] += ((uint128_t) r4) * s1 + ((uint128_t) r1) * s4 + ((uint128_t) r2) * s3 + ((uint128_t) r3) * s2;
t[1] += ((uint128_t) r4) * s2 + ((uint128_t) r2) * s4 + ((uint128_t) r3) * s3;
t[2] += ((uint128_t) r4) * s3 + ((uint128_t) r3) * s4;
t[3] += ((uint128_t) r4) * s4;
r0 = (limb)t[0] & 0x7ffffffffffff; c = (limb)(t[0] >> 51);
t[1] += c; r1 = (limb)t[1] & 0x7ffffffffffff; c = (limb)(t[1] >> 51);
t[2] += c; r2 = (limb)t[2] & 0x7ffffffffffff; c = (limb)(t[2] >> 51);
t[3] += c; r3 = (limb)t[3] & 0x7ffffffffffff; c = (limb)(t[3] >> 51);
t[4] += c; r4 = (limb)t[4] & 0x7ffffffffffff; c = (limb)(t[4] >> 51);
r0 += c * 19; c = r0 >> 51; r0 = r0 & 0x7ffffffffffff;
r1 += c; c = r1 >> 51; r1 = r1 & 0x7ffffffffffff;
r2 += c;
output[0] = r0;
output[1] = r1;
output[2] = r2;
output[3] = r3;
output[4] = r4;
}
```

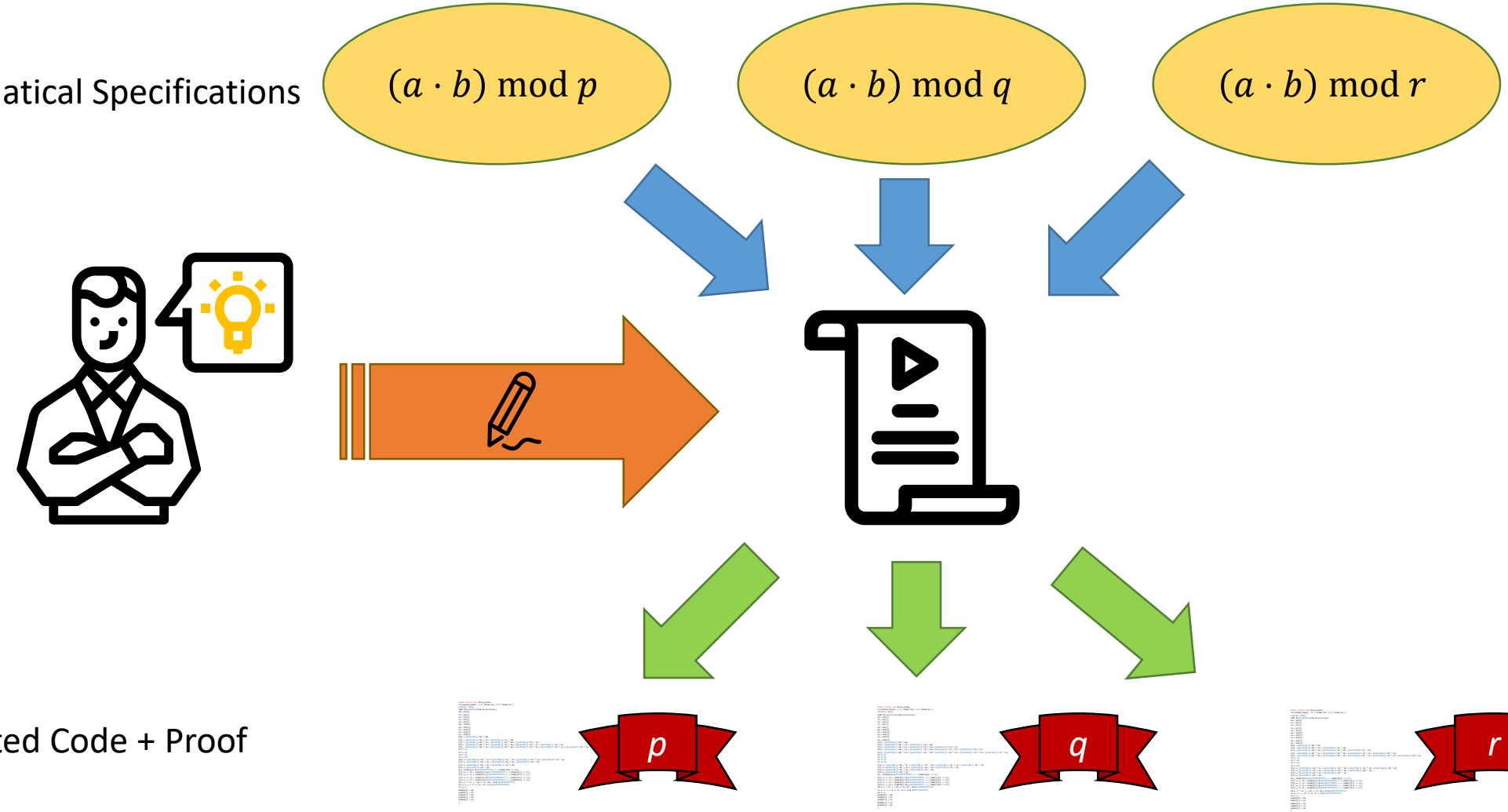
The Overhead of Verification

- 10x—100x overhead



Automating Verification

Mathematical Specifications

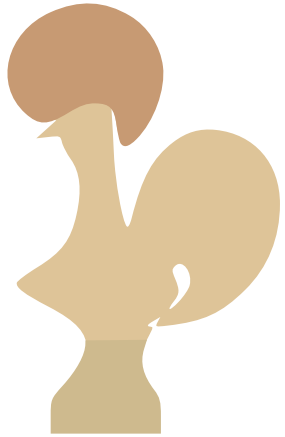


Generated Code + Proof

Image modified from Thinking by ArmOkay from the Noun Project; script by Berkah Icon from the Noun Project; write by royyanandrian from the Noun Project

Our script is run and checked by…

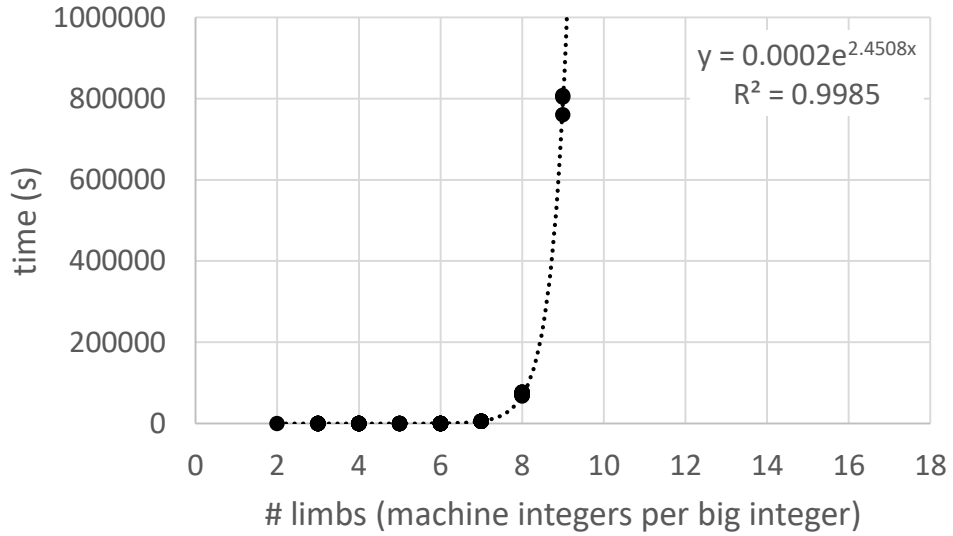
- Dependently typed, interactive, tactic-driven proof assistants
- Dependently typed proof assistants are expressive
- Interactivity allows easy insertion of human ingenuity
- Tactics allow automation



Coq logo from https://calebstanford.com/2019/01/15/coq-vector-image/

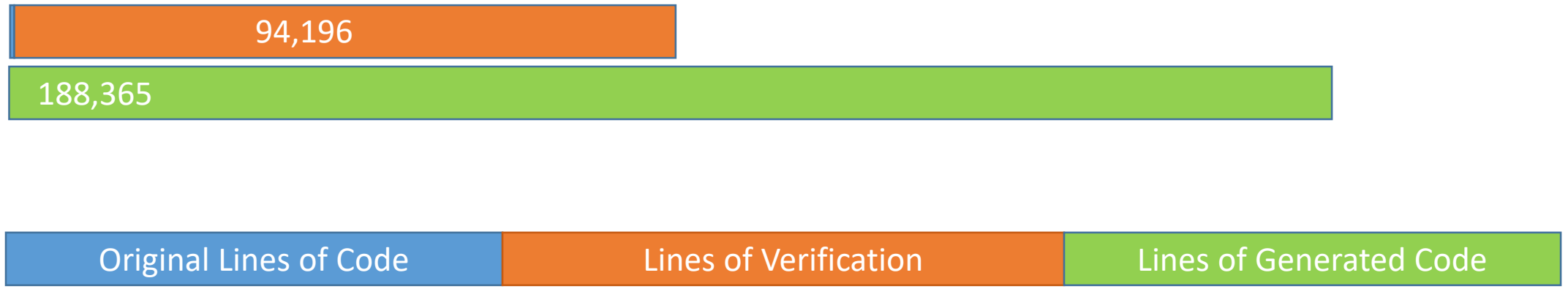
The Big Problem in Automating Verification

- **Asymptotic performance**
- We can automate verification of toy examples in the proof engine
- BUT this automation takes way too long on real examples
- My work has been fixing this performance problem

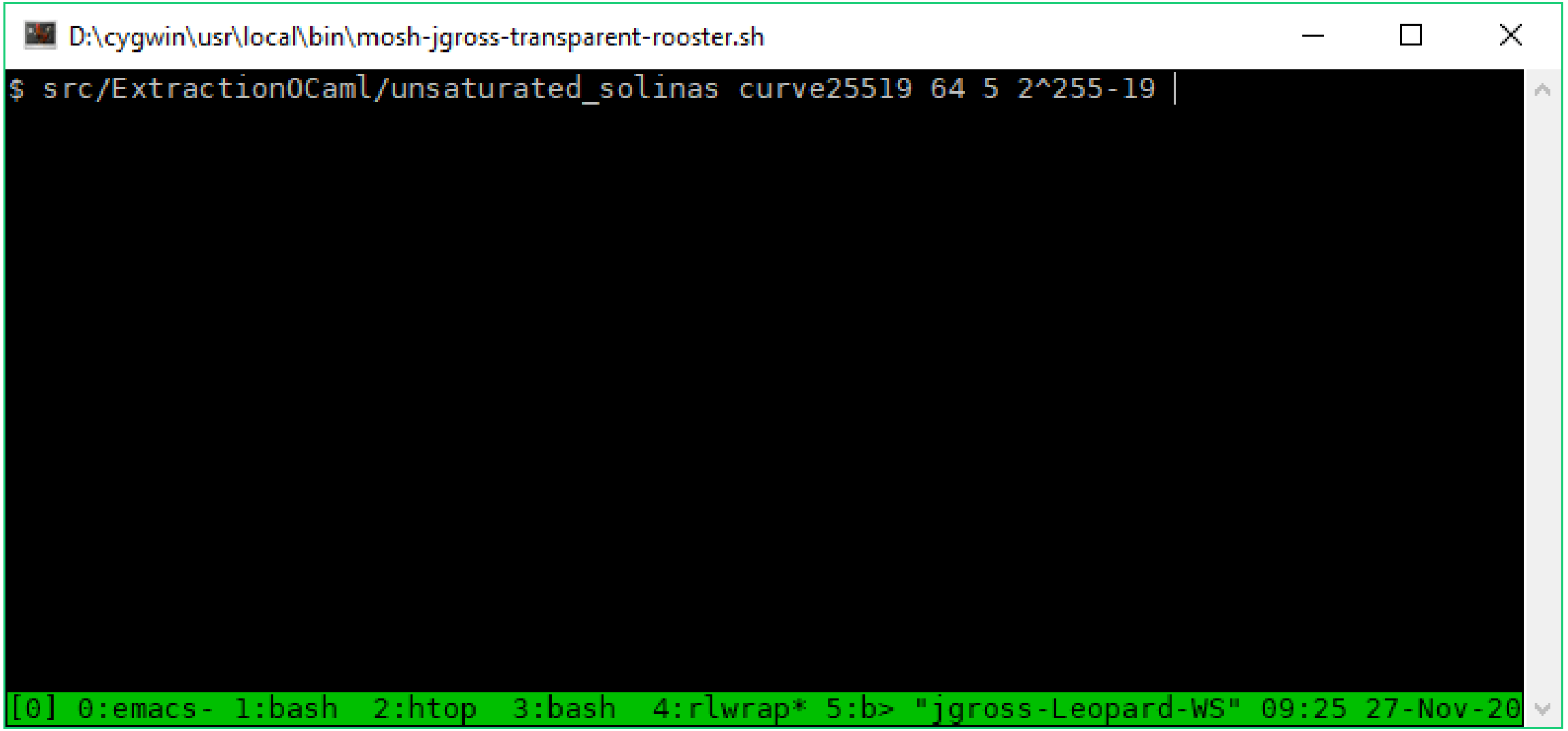


The Potential of Automating Verification

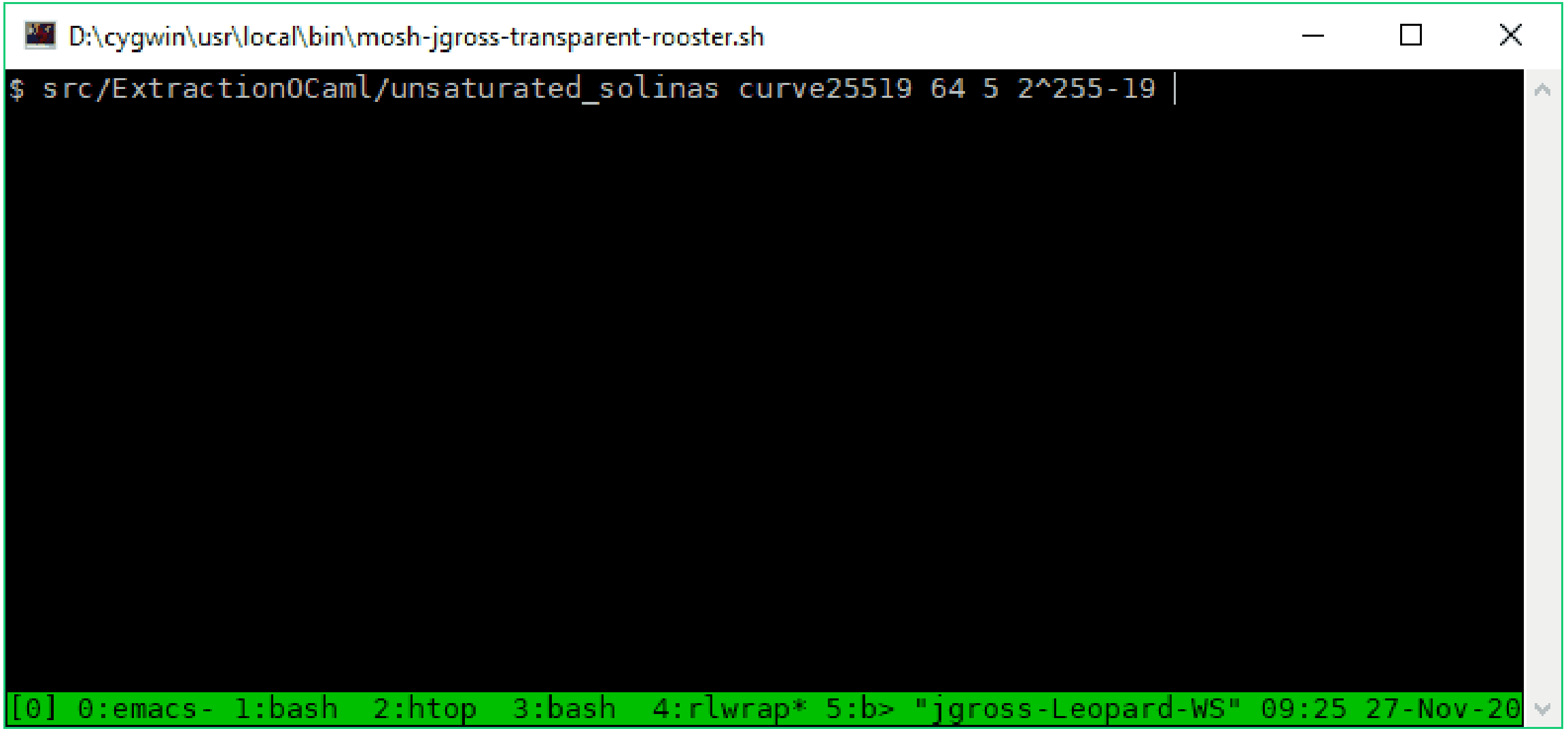
Fiat Cryptography:



It’s really easy to use!



It’s really easy to use!



Requirements

1. Code we generate must be fast and constant time

   Justification: server load, security

2. Easy to add and prove new algorithm, prime, architecture, …

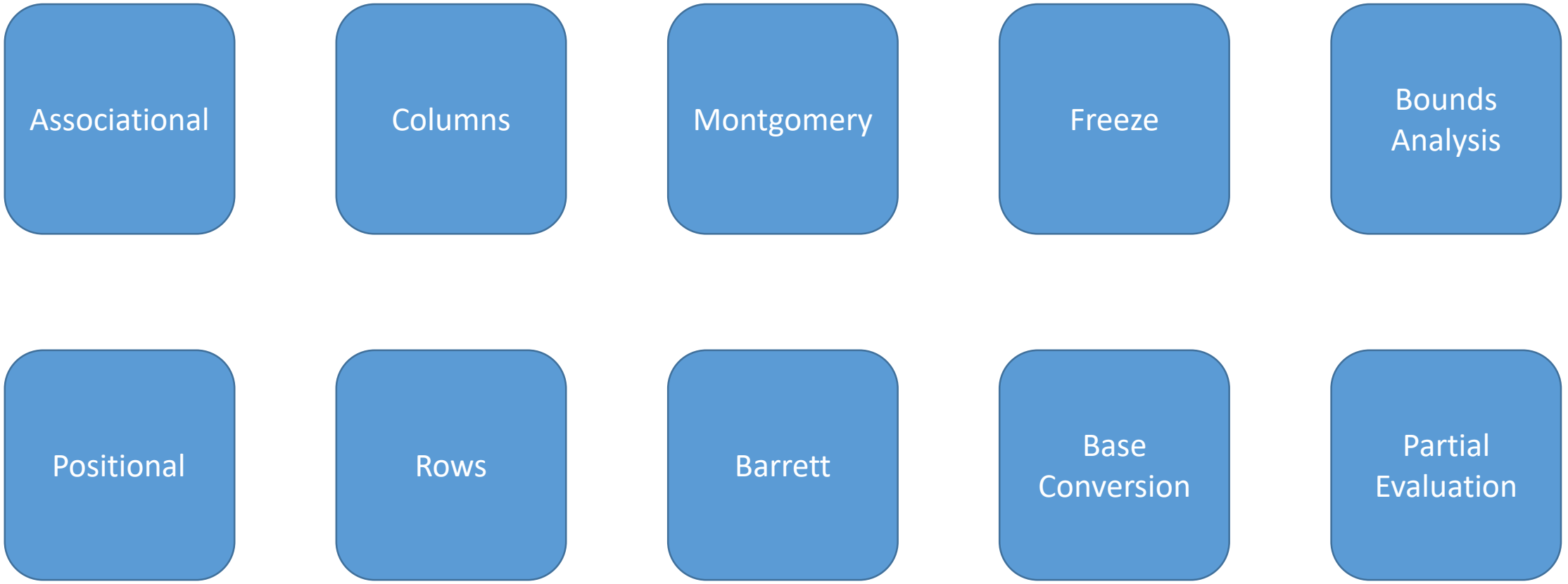
   Justification: scalability of human effort, edit-compile-debug loops

3. Verification should not run forever

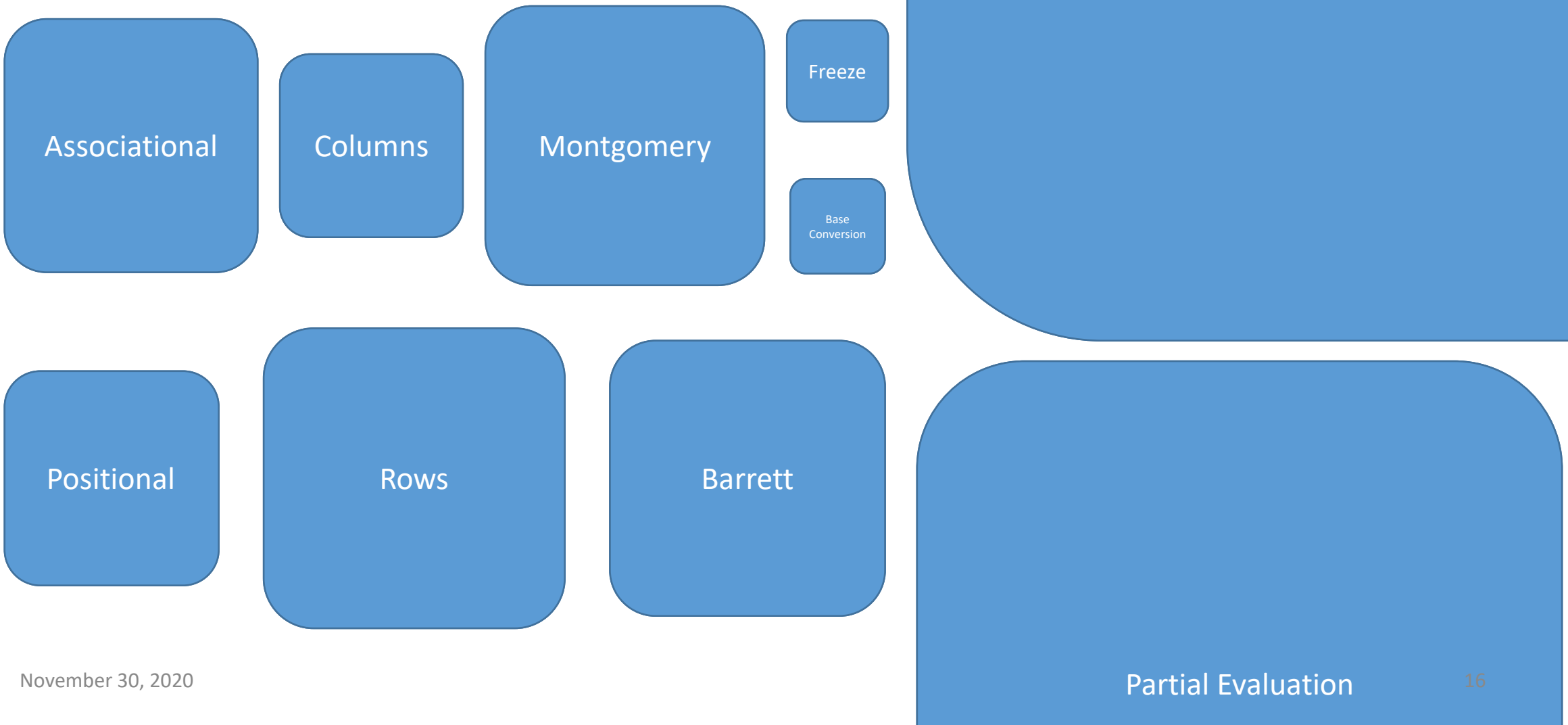
   Justification: usability

Where was the asymptotic performance issue?

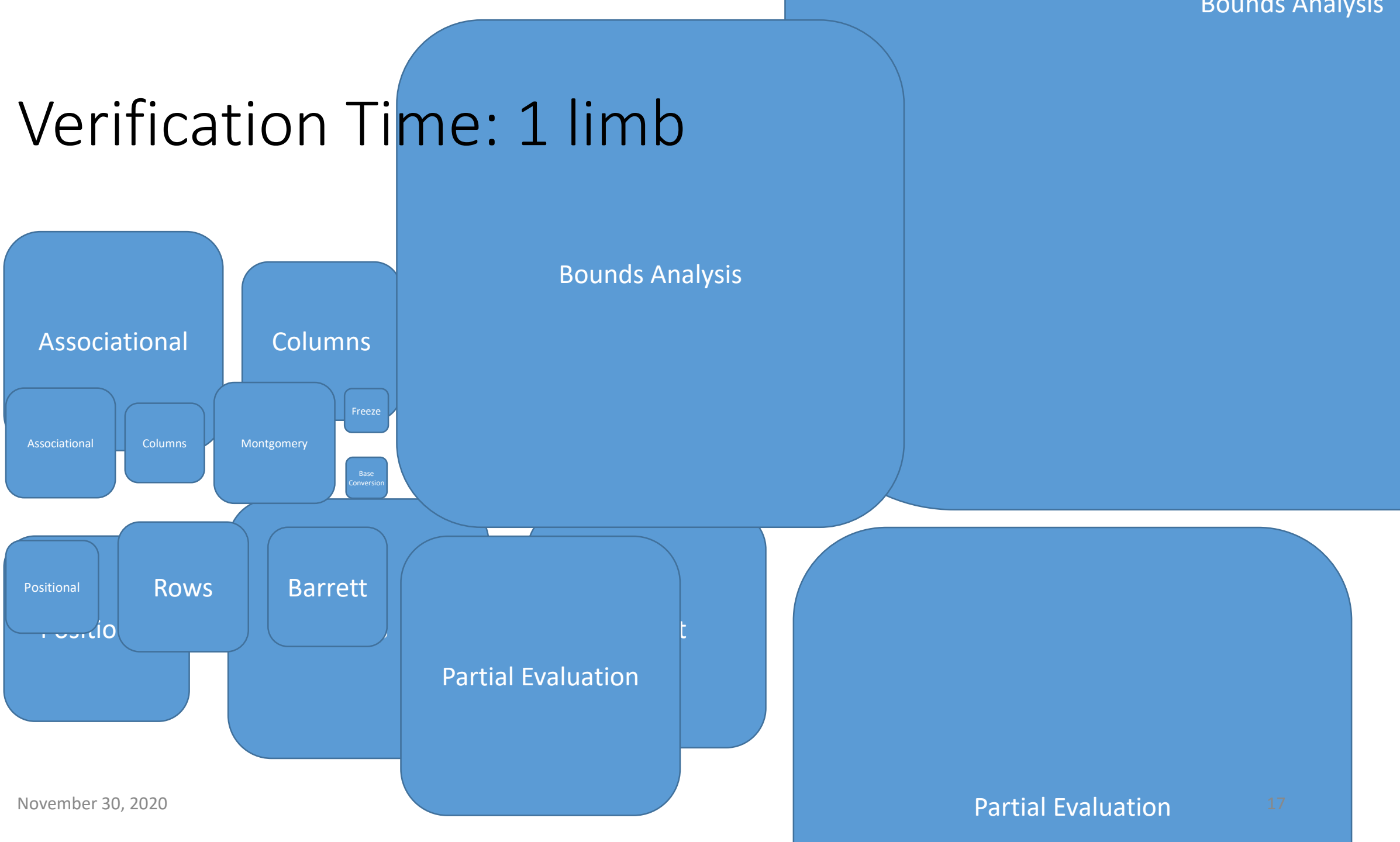
Fiat Cryptography Pieces



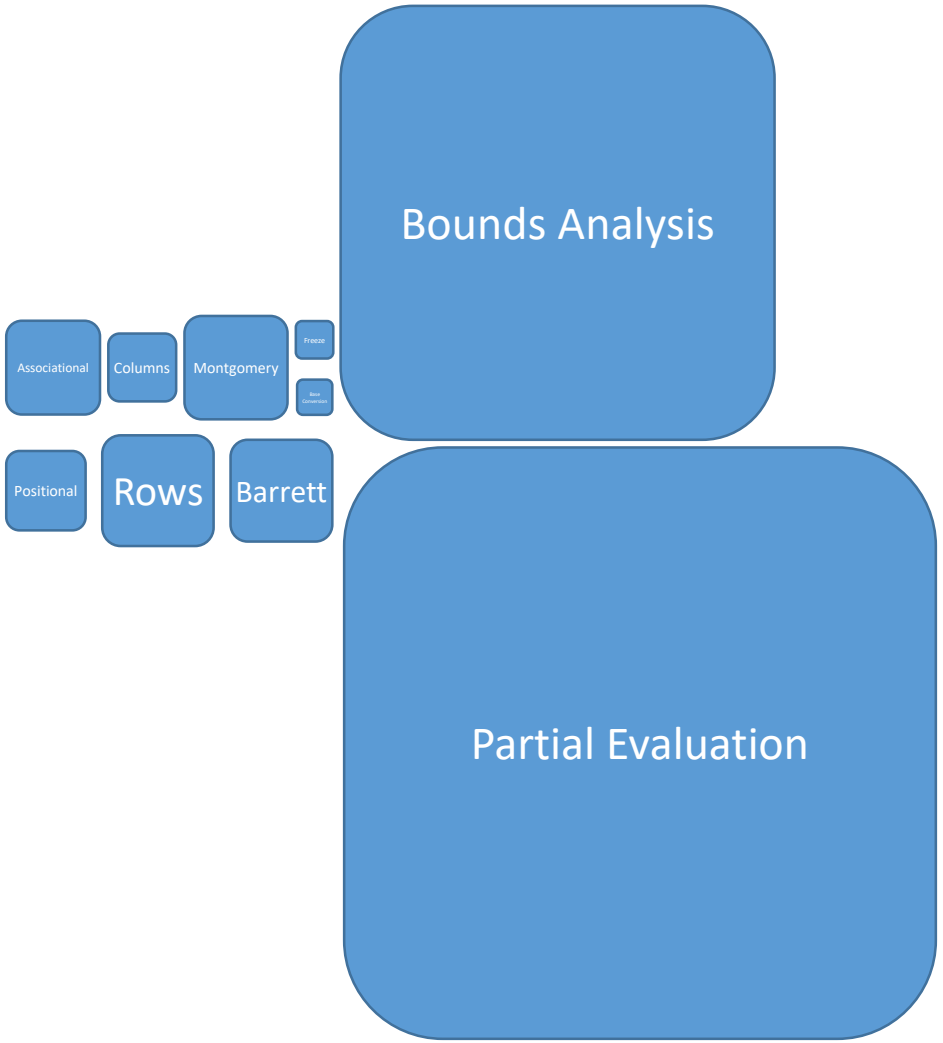
Verification Time: 1 limb



Verification Time: 1 limb



Verification Time: 2 limbs

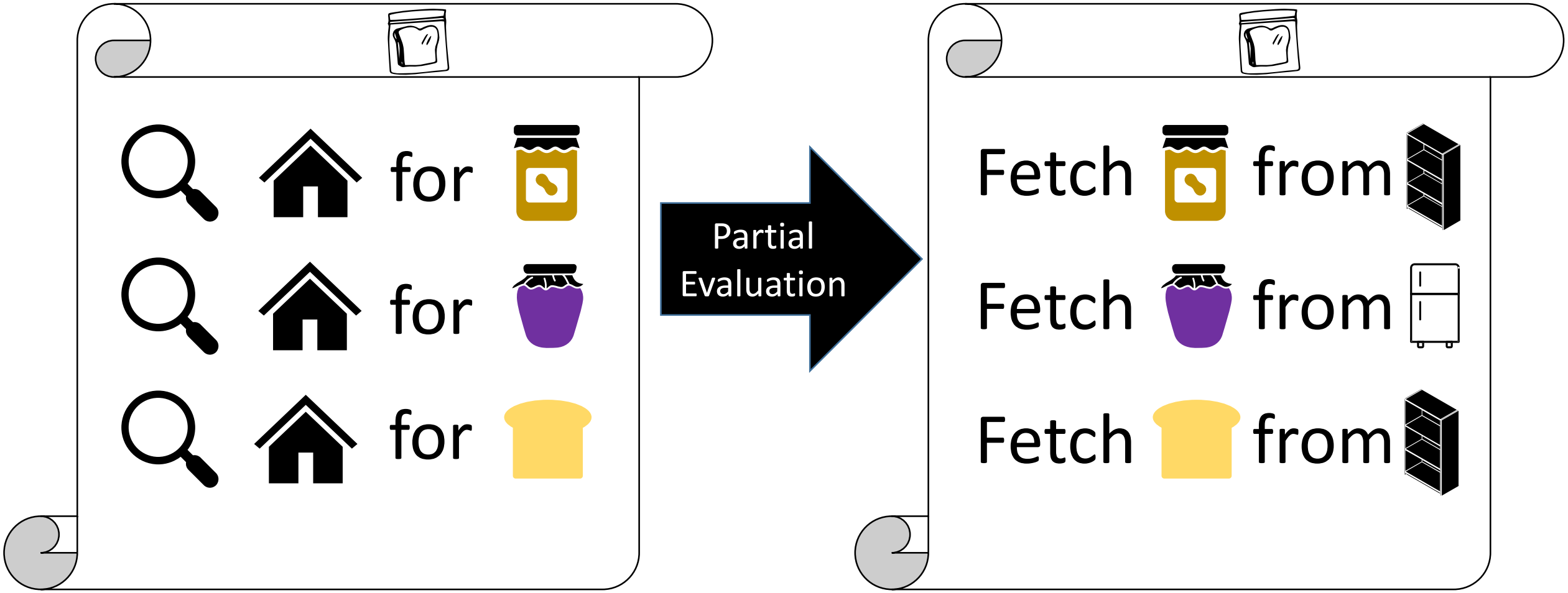


Verification Time: 3 limbs

Bounds Analysis

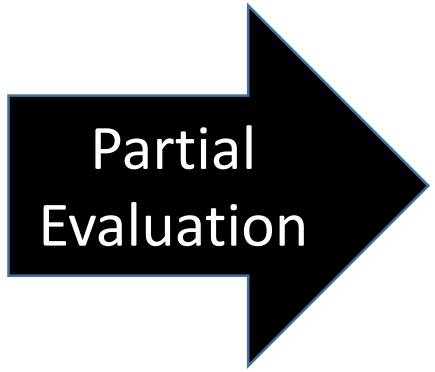
Partial Evaluation

What is Partial Evaluation?



What is Partial Evaluation?

$$x + 2 + y - x + 6$$



$$y + 8$$

Partial Evaluation in Fiat Cryptography

Template Code:

```
Definition mul (p q:list (Z*Z)):list (Z*Z) :=
  flat_map (fun '(w, t) =>
    map (fun '(w', t') =>
      (w * w', t * t'))
    q) p.
Fixpoint square (p:list (Z*Z)):list (Z*Z)
:= match p with
   | [] => []
   | (w, t) :: ts
     => let two_t := 2 * t in
        ((w * w, t * t)
          :: map (λ '(w', t'), (w * w', two_t * t')) ts)
        ++ square ts
   end.
Definition split (s:Z) (p:list(Z*Z)):list (Z*Z) * list (Z*Z)
:= let '(hi, lo) := partition (fun '(w, _) => w mod s =? 0) p in
   (lo, map (fun '(w, t) => (w / s, t)) hi).
Definition reduce (s:Z) (c:list (Z*Z)) (p:list (Z*Z)):list (Z*Z)
:= let '(lo, hi) := split s p in lo ++ mul c hi.
```

Partial Evaluation

Partial Evaluation

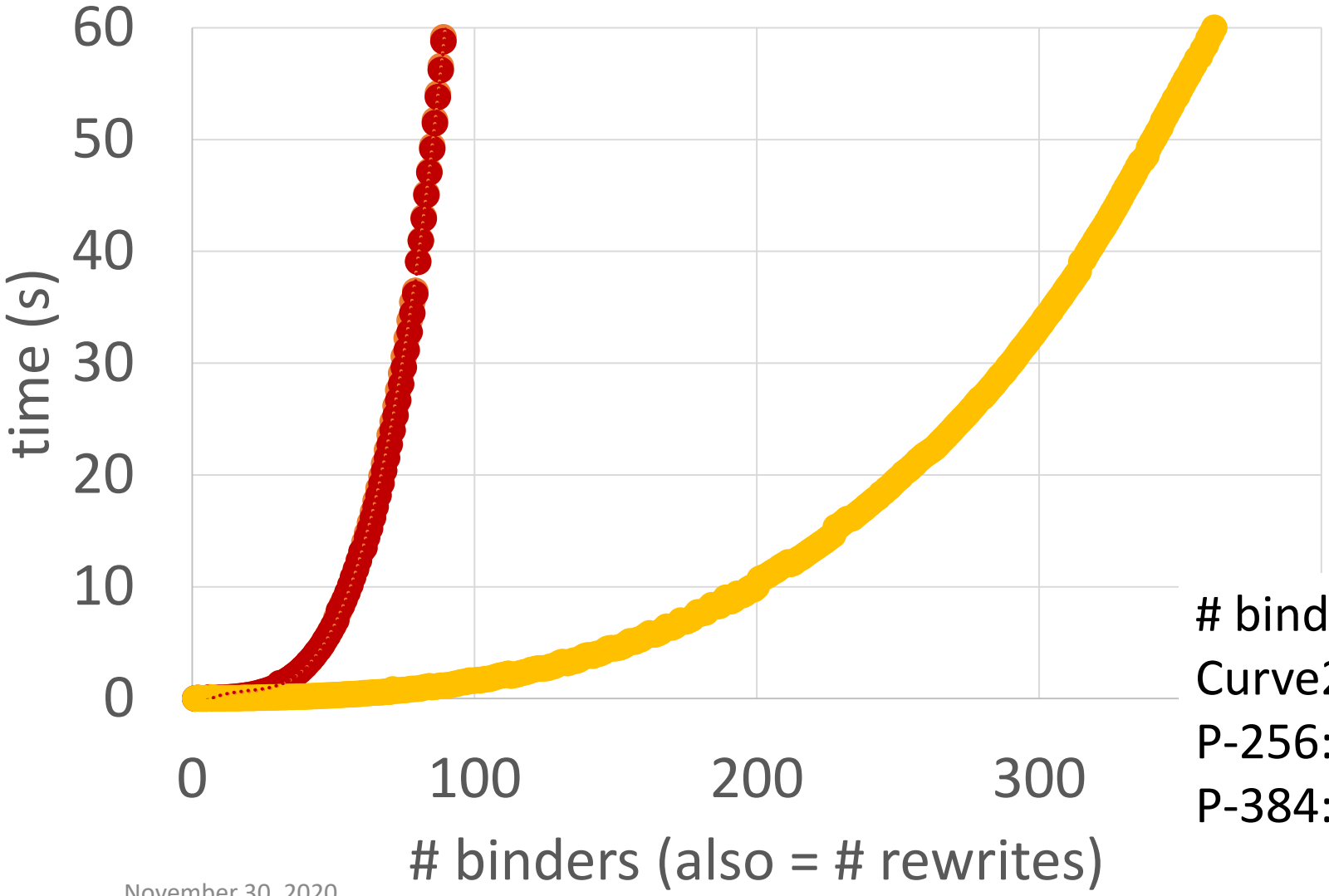
64-bit square

```
static void
fiat_25519_carry_square(uint64_t out1[5], const uint64_t arg1[5])
{
uint64_t x1;
uint64_t x2;
uint64_t x3;
uint64_t x4;
uint64_t x5;
uint64_t x6;
uint64_t x7;
uint64_t x8;
fiat_25519_uint128 x9;
fiat_25519_uint128 x10;
fiat_25519_uint128 x11;
fiat_25519_uint128 x12;
fiat_25519_uint128 x13;
fiat_25519_uint128 x14;
fiat_25519_uint128 x15;
fiat_25519_uint128 x16;
fiat_25519_uint128 x17;
fiat_25519_uint128 x18;
fiat_25519_uint128 x19;
fiat_25519_uint128 x20;
fiat_25519_uint128 x21;
fiat_25519_uint128 x22;
fiat_25519_uint128 x23;
fiat_25519_uint128 x24;
uint64_t x25;
uint64_t x26;
fiat_25519_uint128 x27;
fiat_25519_uint128 x28;
fiat_25519_uint128 x29;
fiat_25519_uint128 x30;
fiat_25519_uint128 x31;
uint64_t x32;
uint64_t x33;
fiat_25519_uint128 x34;
uint64_t x35;
uint64_t x36;
fiat_25519_uint128 x37;
uint64_t x38;
uint64_t x39;
fiat_25519_uint128 x40;
uint64_t x41;
uint64_t x42;
uint64_t x43;
uint64_t x44;
uint64_t x45;
uint64_t x46;
uint64_t x47;
fiat_25519_uint1 x48;
uint64_t x49;
uint64_t x50;
x1 = ((arg1[4]) *
UINT8_C(0x13));
x2 = (x1 * 0x2);
x3 = ((arg1[4]) * 0x2);
x4 = ((arg1[3]) *
UINT8_C(0x13));
x5 = (x4 * 0x2);
x6 = ((arg1[3]) * 0x2);
x7 = ((arg1[2]) * 0x2);
x8 = ((arg1[1]) * 0x2);
x9 =
((fiat_25519_uint128)(arg1[4]) *
x1);
x10 =
((fiat_25519_uint128)(arg1[3]) *
x2);
x11 =
((fiat_25519_uint128)(arg1[3]) *
x4);
x12 =
((fiat_25519_uint128)(arg1[2]) *
x2);
x13 =
((fiat_25519_uint128)(arg1[2]) *
x5);
x14 =
((fiat_25519_uint128)(arg1[2]) *
(arg1[2]));
x15 =
((fiat_25519_uint128)(arg1[1]) *
x2);
x16 =
((fiat_25519_uint128)(arg1[1]) *
x6);
x17 =
((fiat_25519_uint128)(arg1[1]) *
x7);
x18 =
((fiat_25519_uint128)(arg1[1]) *
(arg1[1]));
x19 =
((fiat_25519_uint128)(arg1[0]) *
x3);
x20 =
((fiat_25519_uint128)(arg1[0]) *
x6);
x21 =
((fiat_25519_uint128)(arg1[0]) *
x7);
x22 =
((fiat_25519_uint128)(arg1[0]) *
x8);
x23 =
((fiat_25519_uint128)(arg1[0]) *
(arg1[0]));
x24 = (x23 + (x15 + x13));
x25 = (uint64_t)(x24 >> 51);
x26 = (uint64_t)(x24 &
UINT64_C(0x7ffffffffffff));
x27 = (x19 + (x16 + x14));
x28 = (x20 + (x17 + x9));
x29 = (x21 + (x18 + x10));
x30 = (x22 + (x12 + x11));
x31 = (x25 + x30);
x32 = (uint64_t)(x31 >> 51);
x33 = (uint64_t)(x31 &
UINT64_C(0x7ffffffffffff));
x34 = (x32 + x29);
x35 = (uint64_t)(x34 >> 51);
x36 = (uint64_t)(x34 &
UINT64_C(0x7ffffffffffff));
x37 = (x35 + x28);
x38 = (uint64_t)(x37 >> 51);
x39 = (uint64_t)(x37 &
UINT64_C(0x7ffffffffffff));
x40 = (x38 + x27);
x41 = (uint64_t)(x40 >> 51);
x42 = (uint64_t)(x40 &
UINT64_C(0x7ffffffffffff));
x43 = (x41 * UINT8_C(0x13));
x44 = (x26 + x43);
x45 = (x44 >> 51);
x46 = (x44 &
UINT64_C(0x7ffffffffffff));
x47 = (x45 + x33);
x48 = (fiat_25519_uint1)(x47 >>
51);
x49 = (x47 &
UINT64_C(0x7ffffffffffff));
x50 = (x48 + x36);
out1[0] = x46;
out1[1] = x49;
out1[2] = x50;
out1[3] = x39;
out1[4] = x42;
}
```

32-bit square

```
static void fiat_25519_carry_square(uint32_t out1[10], const uint32_t arg1[10]) {
uint32_t x1;
uint32_t x2;
uint32_t x3;
uint32_t x4;
uint64_t x5;
uint32_t x6;
uint32_t x7;
uint32_t x8;
uint32_t x9;
uint32_t x10;
uint64_t x11;
uint32_t x12;
uint32_t x13;
uint32_t x14;
uint32_t x15;
uint32_t x16;
uint32_t x17;
uint32_t x18;
uint64_t x19;
uint32_t x20;
uint64_t x21;
uint64_t x22;
uint64_t x23;
uint64_t x24;
uint64_t x25;
uint64_t x26;
uint64_t x27;
uint64_t x28;
uint64_t x29;
uint64_t x30;
uint64_t x31;
uint64_t x32;
uint64_t x33;
uint64_t x34;
uint64_t x35;
uint64_t x36;
uint64_t x37;
uint64_t x38;
uint64_t x39;
uint64_t x40;
uint64_t x41;
uint64_t x42;
uint64_t x43;
uint64_t x44;
uint64_t x45;
uint64_t x46;
uint64_t x47;
uint64_t x48;
uint64_t x49;
uint64_t x50;
uint64_t x51;
uint64_t x52;
uint64_t x53;
uint64_t x54;
uint64_t x55;
uint64_t x56;
uint64_t x57;
uint64_t x58;
uint64_t x59;
uint64_t x60;
uint64_t x61;
uint64_t x62;
uint64_t x63;
uint64_t x64;
uint64_t x65;
uint64_t x66;
uint64_t x67;
uint64_t x68;
uint64_t x69;
uint64_t x70;
uint64_t x71;
uint64_t x72;
uint64_t x73;
uint64_t x74;
uint64_t x75;
uint32_t x76;
uint64_t x77;
uint64_t x78;
uint64_t x79;
uint64_t x80;
uint64_t x81;
uint64_t x82;
uint64_t x83;
uint64_t x84;
uint64_t x85;
uint64_t x86;
uint64_t x87;
uint32_t x88;
uint64_t x89;
uint64_t x90;
uint32_t x91;
uint64_t x92;
uint64_t x93;
uint32_t x94;
uint64_t x95;
uint64_t x96;
uint32_t x97;
uint64_t x98;
uint64_t x99;
uint32_t x100;
uint64_t x101;
uint64_t x102;
uint32_t x103;
uint64_t x104;
uint64_t x105;
uint32_t x106;
uint64_t x107;
uint64_t x108;
uint32_t x109;
uint64_t x110;
uint64_t x111;
uint32_t x112;
uint64_t x113;
uint64_t x114;
uint32_t x115;
uint32_t x116;
uint32_t x117;
fiat_25519_uint1 x118;
uint32_t x119;
uint32_t x120;
x1 = ((arg1[9]) * UINT8_C(0x13));
x2 = (x1 * 0x2);
x3 = ((arg1[9]) * 0x2);
x4 = ((arg1[8]) * UINT8_C(0x13));
x5 = ((uint64_t)x4 * 0x2);
x6 = ((arg1[8]) * 0x2);
x7 = ((arg1[7]) * UINT8_C(0x13));
x8 = (x7 * 0x2);
x9 = ((arg1[7]) * 0x2);
x10 = ((arg1[6]) * UINT8_C(0x13));
x11 = (x10 * 0x2);
x12 = ((arg1[6]) * 0x2);
x13 = ((arg1[5]) * UINT8_C(0x13));
x14 = ((arg1[5]) * 0x2);
x15 = ((arg1[4]) * 0x2);
x16 = ((arg1[3]) * 0x2);
x17 = ((arg1[2]) * 0x2);
x18 = ((arg1[1]) * 0x2);
x19 = ((uint64_t)(arg1[9]) * (x1 * 0x2));
x20 = ((uint64_t)(arg1[8]) * x2);
x21 = ((uint64_t)(arg1[8]) * x1);
x22 = ((arg1[7]) * ((uint64_t)x2 * 0x2));
x23 = ((arg1[7]) * x5);
x24 = ((uint64_t)(arg1[7]) * (x7 * 0x2));
x25 = ((uint64_t)(arg1[6]) * x2);
x26 = ((arg1[6]) * x5);
x27 = ((uint64_t)(arg1[6]) * x8);
x28 = ((uint64_t)(arg1[6]) * x10);
x29 = ((arg1[5]) * ((uint64_t)x2 * 0x2));
x30 = ((arg1[5]) * x5);
x31 = ((arg1[5]) * ((uint64_t)x8 * 0x2));
x32 = ((arg1[5]) * x11);
x33 = ((uint64_t)(arg1[5]) * (x13 * 0x2));
x34 = ((uint64_t)(arg1[4]) * x2);
x35 = ((arg1[4]) * x5);
x36 = ((uint64_t)(arg1[4]) * x8);
x37 = ((arg1[4]) * x11);
x38 = ((uint64_t)(arg1[4]) * x14);
x39 = ((uint64_t)(arg1[4]) * (arg1[4]));
x40 = ((arg1[3]) * ((uint64_t)x2 * 0x2));
x41 = ((arg1[3]) * x5);
x42 = ((arg1[3]) * ((uint64_t)x8 * 0x2));
x43 = ((uint64_t)(arg1[3]) * x12);
x44 = ((uint64_t)(arg1[3]) * (x14 * 0x2));
x45 = ((uint64_t)(arg1[3]) * x15);
x46 = ((uint64_t)(arg1[3]) * ((arg1[3]) *
0x2));
x47 = ((uint64_t)(arg1[2]) * x2);
x48 = ((arg1[2]) * x5);
x49 = ((uint64_t)(arg1[2]) * x9);
x50 = ((uint64_t)(arg1[2]) * x12);
x51 = ((uint64_t)(arg1[2]) * x14);
x52 = ((uint64_t)(arg1[2]) * x15);
x53 = ((uint64_t)(arg1[2]) * x16);
x54 = ((uint64_t)(arg1[2]) * (arg1[2]));
x55 = ((arg1[1]) * ((uint64_t)x2 * 0x2));
x56 = ((uint64_t)(arg1[1]) * x6);
x57 = ((uint64_t)(arg1[1]) * (x9 * 0x2));
x58 = ((uint64_t)(arg1[1]) * x12);
x59 = ((uint64_t)(arg1[1]) * (x14 * 0x2));
x60 = ((uint64_t)(arg1[1]) * x15);
x61 = ((uint64_t)(arg1[1]) * (x16 * 0x2));
x62 = ((uint64_t)(arg1[1]) * x17);
x63 = ((uint64_t)(arg1[1]) * ((arg1[1]) *
0x2));
x64 = ((uint64_t)(arg1[0]) * x3);
x65 = ((uint64_t)(arg1[0]) * x6);
x66 = ((uint64_t)(arg1[0]) * x9);
x67 = ((uint64_t)(arg1[0]) * x12);
x68 = ((uint64_t)(arg1[0]) * x14);
x69 = ((uint64_t)(arg1[0]) * x15);
x70 = ((uint64_t)(arg1[0]) * x16);
x71 = ((uint64_t)(arg1[0]) * x17);
x72 = ((uint64_t)(arg1[0]) * x18);
x73 = ((uint64_t)(arg1[0]) * (arg1[0]));
x74 = (x73 + (x55 + (x48 + (x42 + (x37 +
x33)))));
x75 = (x74 >> 26);
x76 = (uint32_t)(x74 & UINT32_C(0x3ffffff));
x77 = (x64 + (x56 + (x49 + (x43 + x38))));
x78 = (x65 + (x57 + (x50 + (x44 + (x39 +
x19)))));
x79 = (x66 + (x58 + (x51 + (x45 + x20))));
x80 = (x67 + (x59 + (x52 + (x46 + (x22 +
x21)))));
x81 = (x68 + (x60 + (x53 + (x25 + x23))));
x82 = (x69 + (x61 + (x54 + (x29 + (x26 +
x24)))));
x83 = (x70 + (x62 + (x34 + (x30 + x27))));
x84 = (x71 + (x63 + (x40 + (x35 + (x31 +
x28)))));
x85 = (x72 + (x47 + (x41 + (x36 + x32))));
x86 = (x75 + x85);
x87 = (x86 >> 25);
x88 = (uint32_t)(x86 & UINT32_C(0x1ffffff));
x89 = (x87 + x84);
x90 = (x89 >> 26);
x91 = (uint32_t)(x89 & UINT32_C(0x3ffffff));
x92 = (x90 + x83);
x93 = (x92 >> 25);
x94 = (uint32_t)(x92 & UINT32_C(0x1ffffff));
x95 = (x93 + x82);
x96 = (x95 >> 26);
x97 = (uint32_t)(x95 & UINT32_C(0x3ffffff));
x98 = (x96 + x81);
x99 = (x98 >> 25);
x100 = (uint32_t)(x98 &
UINT32_C(0x1ffffff));
x101 = (x99 + x80);
x102 = (x101 >> 26);
x103 = (uint32_t)(x101 &
UINT32_C(0x3ffffff));
x104 = (x102 + x79);
x105 = (x104 >> 25);
x106 = (uint32_t)(x104 &
UINT32_C(0x1ffffff));
x107 = (x105 + x78);
x108 = (x107 >> 26);
x109 = (uint32_t)(x107 &
UINT32_C(0x3ffffff));
x110 = (x108 + x77);
x111 = (x110 >> 25);
x112 = (uint32_t)(x110 &
UINT32_C(0x1ffffff));
x113 = (x111 * UINT8_C(0x13));
x114 = (x76 + x113);
x115 = (uint32_t)(x114 >> 26);
x116 = (uint32_t)(x114 &
UINT32_C(0x3ffffff));
x117 = (x115 + x88);
x118 = (fiat_25519_uint1)(x117 >> 25);
x119 = (x117 & UINT32_C(0x1ffffff));
x120 = (x118 + x91);
out1[0] = x116;
out1[1] = x119;
out1[2] = x120;
out1[3] = x94;
out1[4] = x97;
out1[5] = x100;
out1[6] = x103;
out1[7] = x106;
out1[8] = x109;
out1[9] = x112;
}
```

Partial Evaluation is slow



- rewrite_strat topdown
- rewrite_strat bottomup
- repeat setoid_rewrite

binders for:

Curve25519:	90 (x64) or 188 (x32)
P-256:	150 (x64) or 400 (x32)
P-384:	300 (x64) or 916 (x32)

What is a proof engine?

- Declare a goal to prove
- Issue instructions to make partial progress on proving
- Can write scripts to automate issuing of instructions
- Tracks the progress and current state
- Can issue a trail (proof certificate) to be checked by a small checker (“kernel”)

Our Approach

- Dig deep to find the places of asymptotic blowup
- Understand the precise source of the blowup
- Fuse the different compiler passes deeply

Requirements for Partial Evaluation

- β-reduction
- ιδ-reduction + rewrites
- code sharing preservation

β-reduction

- Useful for eliminating function call overhead in the generated code, which is important for output code performance

- Example: ((λ x. x + 5) 2) ⇝ 2 + 5

ιδ-reduction + rewrites

- Useful for precomputation and eliminating function call overhead
- Arithmetic simplification necessary for getting right asymptotics of generated lines of code in fiat-crypto (quadratic vs. quartic)
- Example:
  map (λ x. x + 5) [0; 1; z] ⇝ [(λ x. x + 5) 0; (λ x. x + 5) 1; (λ x. x + 5) z]
  - Note that this leaves β redexes
  - Without β-reduction, this can blow up code size
- Fusing rewriting with β-reduction in a way that scales

Code Sharing Preservation

- Necessary for avoiding exponential blowup in generated code size

- Example:
  map f (**let** y := x + x **in let** z := y + y **in** [z; z; z])
  ⇝ **let** y := x + x **in let** z := y + y **in** map f [z; z; z]
  ⇝ **let** y := x + x **in let** z := y + y **in** [f z; f z; f z]

- Fusing this with β- and ι- reduction

Compiler passes

- β-reduction
  - eliminating function call overhead
- ιδ-reduction + rewrites
  - inlining definitions to eliminate function call overhead
  - arithmetic simplification
- code sharing preservation
  - to avoid exponential blowup in code size

Extra Requirements

- Verified
  - Without extending the TCB
- Performant
  - Should not introduce extra super-linear factors

The compiler passes need to be fused

- Needed to achieve adequate asymptotic performance!
- Separating out rewriting results in quartic rather than quadratic loc
- Separating out ι-reduction (constant propagation) results in *enormous* code-size blowup
- Separating out code-sharing-preservation results in *enormous* code-size blowup

Implementation

- Reflective for performant and verified
- Normalization by Evaluation (NbE) (for β)

  + let-lifting monad (code-sharing)

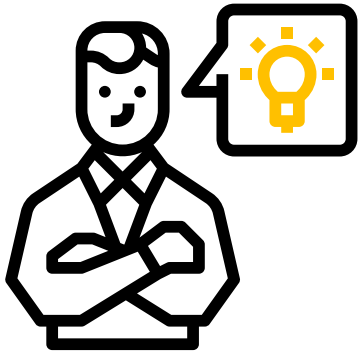
  + rewriting (ιδ+rewrite)

Proof by Reflection

- Most steps in the proof engine make partial progress towards a goal and leave behind a trail
- Coq's proof engine has a highly optimized primitive step for validating the output of a computation
- Phrasing the goal so that we can just validate the output of a computation
  - Verifying the process, rather than having an ad-hoc process that leaves behind a trail verifying the output

Non-Reflection Example

```
Inductive is_even:ℕ→ℙ := |zero_even : is_even 0 |two_plus_even n : is_even n → is_even (2+n).
```



```
Goal is_even 9002.
```

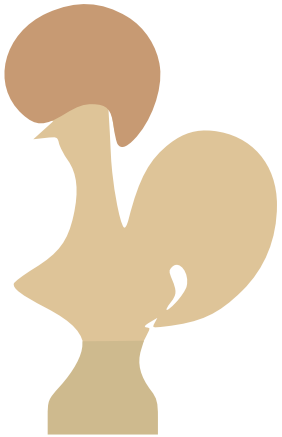
```
repeat constructor
```

```
Qed
```

Goal is:
`is_even 9002`

Current Proof is:

```
two_plus_even 9000
(two_plus_even 8998
(two_plus_even 8996
(two_plus_even 8994 …
```



Reflection Example: Up-Front Work

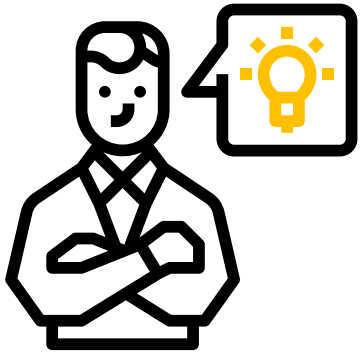
```
Inductive is_even:ℕ→ℙ := |zero_even : is_even 0 |two_plus_even n : is_even n → is_even (2+n).
```

```
Inductive parity := even | odd.
```

```
Definition flip_parity p
:= match p with even => odd | odd => even end.
```

```
Fixpoint parity_of (n : nat) : parity :=
match n with
| O => even
| S n' => flip_parity (parity_of n') end.
```

```
Lemma parity_of_correct
: ∀ n, parity_of n = even → is_even n.
Proof.
  intro n; assert (H' : match parity_of n with
                        | even => is_even n
                        | odd => is_even (S n) end).
  { induction n as [|n IH]; cbn; try constructor.
    destruct (parity_of n); cbn; try constructor; try assumption. }
  intro H; rewrite H in H'; assumption.
Qed.
```

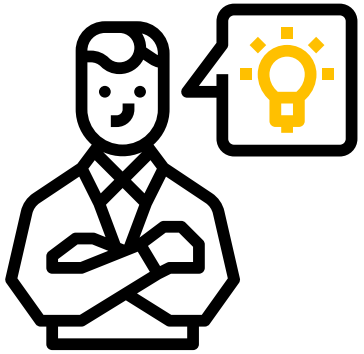


Thinking modified from image by ArmOkay from the Noun Project CC BY 3.0; Coq logo from https://calebstanford.com/2019/01/15/coq-vector-image/

Reflection Example

```
Inductive is_even:ℕ→ℙ := |zero_even : is_even 0 |two_plus_even n : is_even n → is_even (2+n).
Inductive parity := even | odd.
Fixpoint parity_of : ℕ → parity
```

```
Lemma parity_of_correct
: ∀ n, parity_of n = even → is_even n.
```



```
Goal is_even 9002.
```

```
apply parity_of_correct
```

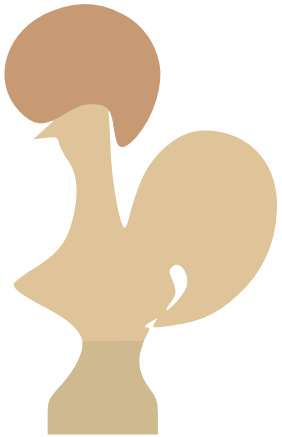
```
vm_compute; reflexivity.
```

```
Qed
```

Goal is:
`is_even 9002`

Goal is:
`parity_of 9002 = even`

Current Proof is:
`parity_of_correct 9002 (eq_refl even)`



Why reflective rewriting?

- Reflective rewriting is asymptotically faster
- The trail left by proof-engine-based rewriting is super-linear in the size of the code being transformed
- Tracking the goal incurs super-linear overhead in the number of binders
- Recursively computing only the output is asymptotically faster
- Side benefit: we can extract it to OCaml to run as a nifty command-line utility

Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

Example: Turn “(λ z p n x. z + (x + (p + n))) 0 1 (-1)” into

```
(λ z p n x. (λ a b. rewrite(“+”, a, b))
  z ((λ a b. rewrite(“+”, a, b))
       x ((λ a b. rewrite(“+”, a, b))
            p n)))) (rewrite(“0”)) (rewrite(“1”)) (rewrite(“-1”))
```

Expression application ⇝ Gallina application

Expression abstraction ⇝ Gallina abstraction

Expression constants ⇝ rewriter invocations on η-expanded forms

Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

Example: Turn “(λ z p n x. z + (x + (p + n))) 0 1 (-1)” into

```
(λ z p n x. (λ a b. rewrite(“+”, a, b))
  z ((λ a b. rewrite(“+”, a, b))
       x ((λ a b. rewrite(“+”, a, b))
            p n)))) (rewrite(“0”)) (rewrite(“1”)) (rewrite(“-1”))
```

Then reduce!

Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

Example: Turn “(λ z p n x. z + (x + (p + n))) 0 1 (-1)” into

(λ z p n x. (λ a b. rewrite(“+”, a, b))
  z ((λ a b. rewrite(“+”, a, b))
      x ((λ a b. rewrite(“+”, a, b))
          p n)))) (rewrite(“0”)) (rewrite(“1”)) (rewrite(“-1”))

Then reduce!

⇝ (λ x. rewrite(“+”, “0”, rewrite(“+”, “x”, rewrite(“+”, “1”, “-1”))))

Let-Lifting

- Let-Lifting monad for code-sharing-preservation
- Assignment + return; bind is derived
- Rewrote NbE in this Let-Lifting monad
- Haven’t seen it in the literature, but it’s not too tricky
- Automatic ι-reduction was too tricky to figure out, so I hard-coded the cases we needed for fiat-crypto

Rewriting

- For ιδ+rewrite
- Using Parametric Higher-Order Abstract Syntax (PHOAS) to deal with binders allows delaying rewriting
- We thus achieve complete rewriting in a single pass when the rewrite rules form a DAG
  - We have extra magic for when they don't. The magic is called "fuel" and "try again".

More Features

- Select rewrite rule based on Coq's pattern matching so we don't need to walk the entire list of rewrite rules at every identifier/constant node just to see which ones apply
- On-the-fly emission of a type of codes for relevant constants
- Partial evaluation on the generated rewriter (further 2x efficiency)

Implementation

- Reflective for performant and verified
- Normalization by Evaluation (NbE) (for β)

  + let-lifting monad (code-sharing)

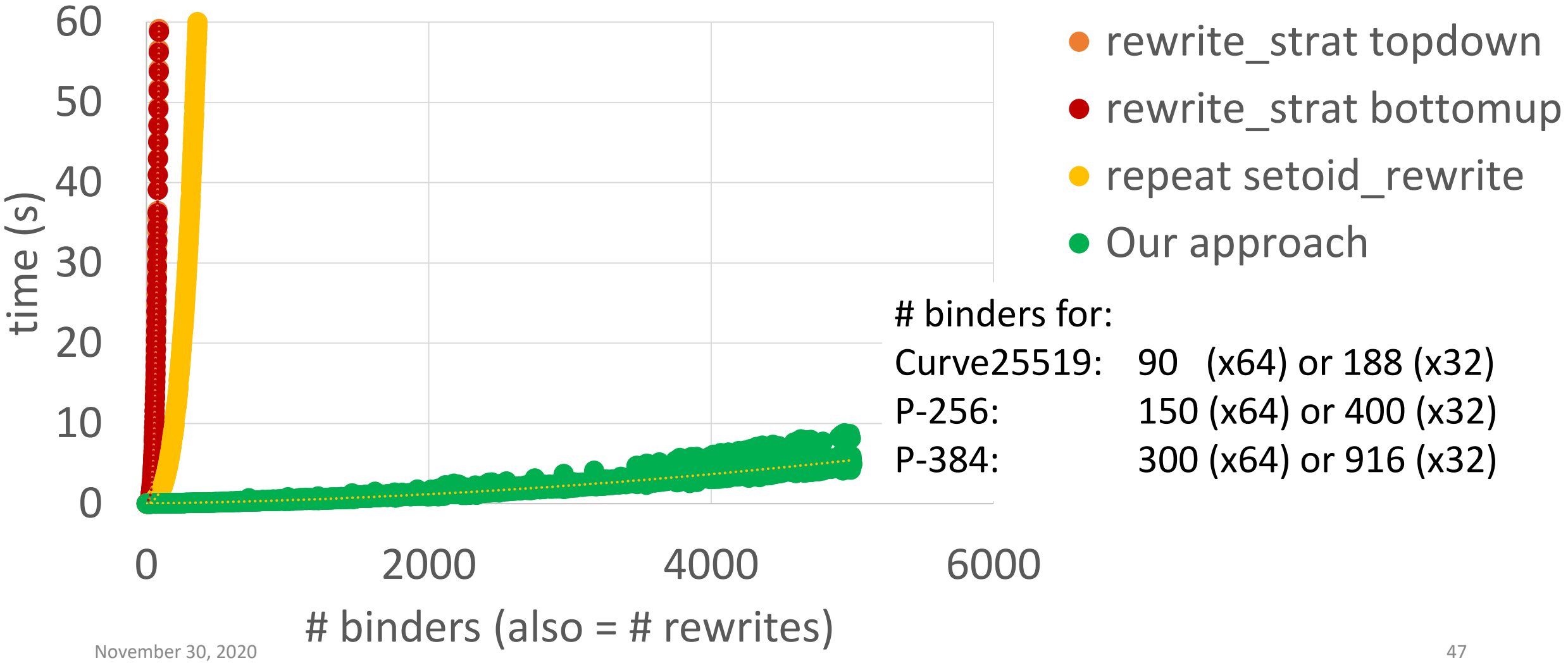
  + rewriting (ιδ+rewrite)

  + more features

Evaluation

- It works!
- It's performant!

Performance



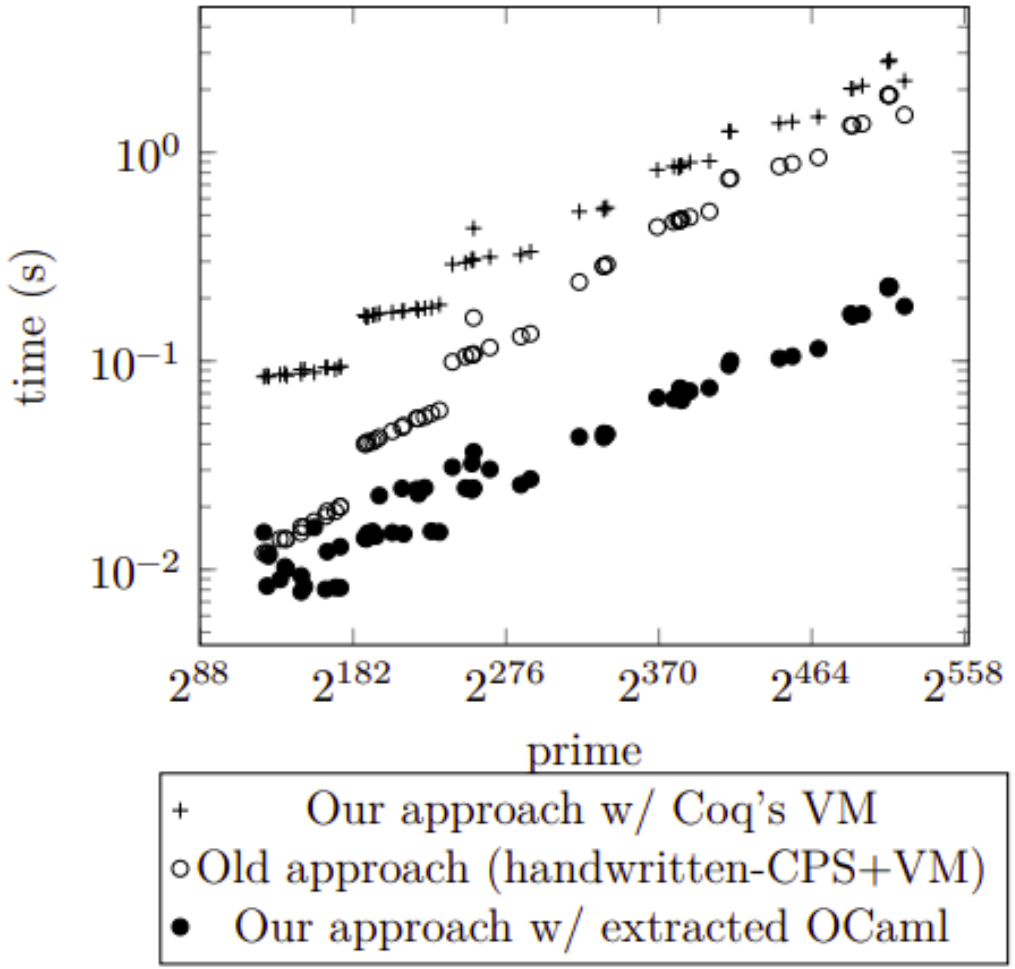
binders for:

Curve25519: 90 (x64) or 188 (x32)

P-256: 150 (x64) or 400 (x32)

P-384: 300 (x64) or 916 (x32)

Performance on Fiat Cryptography



Our Approach

- Dig deep to find the places of asymptotic blowup
- Understand the precise source of the blowup
- Fuse the different compiler passes deeply

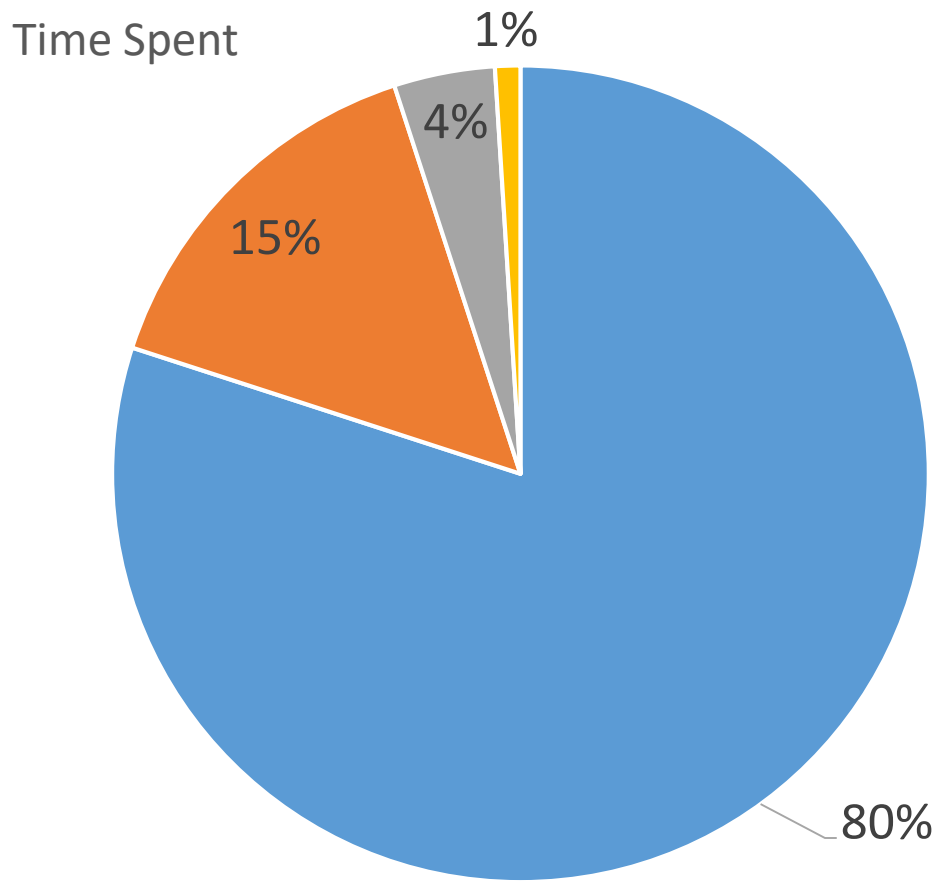
Takeaways

- Opportunity: Automate Verification to Enable Innovation
- Big Problem: Asymptotic Performance
- My Contribution: Reflective Partial Evaluation
- Important Next Steps

Let’s take a step back

- We succeeded, but this was very hard
- All of this to work around inadequate asymptotic performance of the proof engine
- This is typical!

What I did in my PhD



- Performance engineering (working around slowness in Coq)
- Coding new things
- Misc
- Discovering interesting new things

Our current approach to performance

- Using abstraction to prevent excessive unfolding
- Carving out the proof engine…
- …and replacing it with reflection

Abstraction is not enough

- Systems code is often written in an adversarial context
- Symmetric crypto code is often written empirically
- Performant code breaks abstraction barriers

Reflection will not save us

- Using a proof assistant is for easily inserting human ingenuity to prove a broad range of things
- Using reflection is essentially giving up “easy” part
- As problems get bigger and harder and we need more ingenuity, it won’t be cost-effective to do it reflectively
- Already in the partial evaluator I hit the same performance-scaling issues that I was trying to avoid by writing it in the first place (albeit at a smaller and surmountable scale)

Can we avoid carving out the proof engine?

- Where is the performance issue?
- Turns out that it's pretty far from the problem we're solving
  - (This should be obvious, because if it wasn't, reflection wouldn't help.)
  - Example: evar instance allocation has nothing to do with correctness of a given C algorithm
- In my experience, it's not about generating a proof trail and it's not even really about individual steps being slow
  - It's about asymptotics of accessing and updating data being tracked
  - Sometimes just walking the term repeatedly is too much overhead

Not just an engineering challenge

- “Don’t make stupid choices” isn’t enough to get good asymptotic performance
- Try writing `rewrite_strat`
  - inside the tactic engine
  - every step considered as progress towards proving something
  - linear in # of binders + # of rewrite locations + size of term
  - really hard, maybe impossible!
- We need to systematically study proof engines with an eye towards asymptotic performance!

Next Questions about Proof Engines

- Where does the performance overhead really come from?
- What things are people not currently doing due to performance overhead?
- What is an adequate set of primitives?
- What are acceptable thresholds on asymptotic behavior?
- Is it possible to achieve adequate performance simultaneously on all the primitives?

My hope

I think solving this problem—getting the basics of proof engines right, asymptotically—will drastically accelerate the scale of what we as a field can handle, and bring verification closer to its promise and potential of enabling innovation in industry.

Thank you for your time and attention!

Questions?