

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
- State-of-the-Art Methodologies
 - Abstraction & Reflection
 - Reflective Partial Evaluation
- Important Next Steps

Structure of this Defense

- Automating Verification*: Why?
- Automating Verification*: What?
- Automating Verification*: How?
 - Fiat Crypto
 - Partial Evaluator & Rewriter
- Automating Verification*: What next?

* Interactive dependently typed tactic-driven proof assistants

Automating Verification: Why?

Innovation in Cryptography

Not Tinkering

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

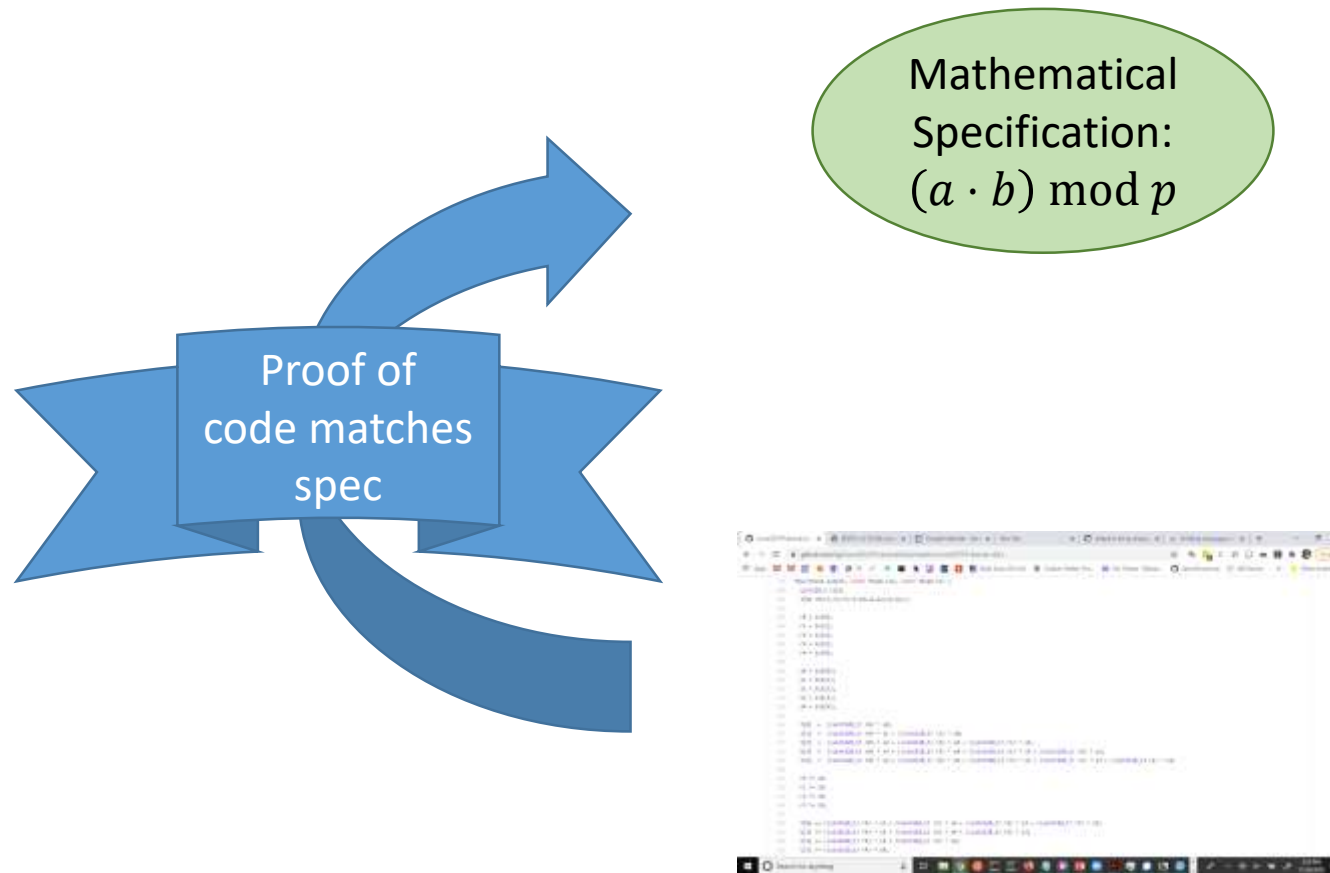
Mathematical
Specification:
 $(a \cdot b) \bmod p$



Tinkering

- Reduce costs (server & user)
- More mathematical security
- Keeping up with more powerful attackers

The Promise of Verification



```
1 // Compute (a * b) mod p using a naive algorithm
2 int naive_mod_mult(int a, int b, int p) {
3     int res = 0;
4     for (int i = 0; i < b; i++) {
5         res = (res + a) % p;
6     }
7     return res;
8 }
9
10 // Compute (a * b) mod p using a more efficient algorithm
11 int mod_mult(int a, int b, int p) {
12     int res = 0;
13     while (b > 0) {
14         if (b & 1) {
15             res = (res + a) % p;
16         }
17         a = (a * 2) % p;
18         b = b / 2;
19     }
20     return res;
21 }
22
23 // Compute (a * b) mod p using a naive algorithm
24 int naive_mod_mult(int a, int b, int p) {
25     int res = 0;
26     for (int i = 0; i < b; i++) {
27         res = (res + a) % p;
28     }
29     return res;
30 }
31
32 // Compute (a * b) mod p using a more efficient algorithm
33 int mod_mult(int a, int b, int p) {
34     int res = 0;
35     while (b > 0) {
36         if (b & 1) {
37             res = (res + a) % p;
38         }
39         a = (a * 2) % p;
40         b = b / 2;
41     }
42     return res;
43 }
44
45 // Compute (a * b) mod p using a naive algorithm
46 int naive_mod_mult(int a, int b, int p) {
47     int res = 0;
48     for (int i = 0; i < b; i++) {
49         res = (res + a) % p;
50     }
51     return res;
52 }
53
54 // Compute (a * b) mod p using a more efficient algorithm
55 int mod_mult(int a, int b, int p) {
56     int res = 0;
57     while (b > 0) {
58         if (b & 1) {
59             res = (res + a) % p;
60         }
61         a = (a * 2) % p;
62         b = b / 2;
63     }
64     return res;
65 }
```

The Overhead of Verification

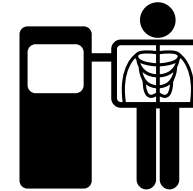
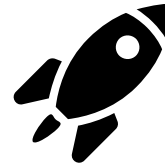
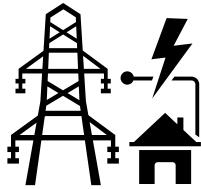
- 10×—100× overhead
 - Plots of overhead here (incl fiat crypto)

The Promise of Automating Verification

- 10×—100× overhead
 - Plots of overhead here (incl fiat crypto)
- Automation will let us eliminate marginal overhead
 - Plot of fiat-crypto generating 188365 loc without increasing verification

Examples Abound

- Critical software



- Risks of innovation

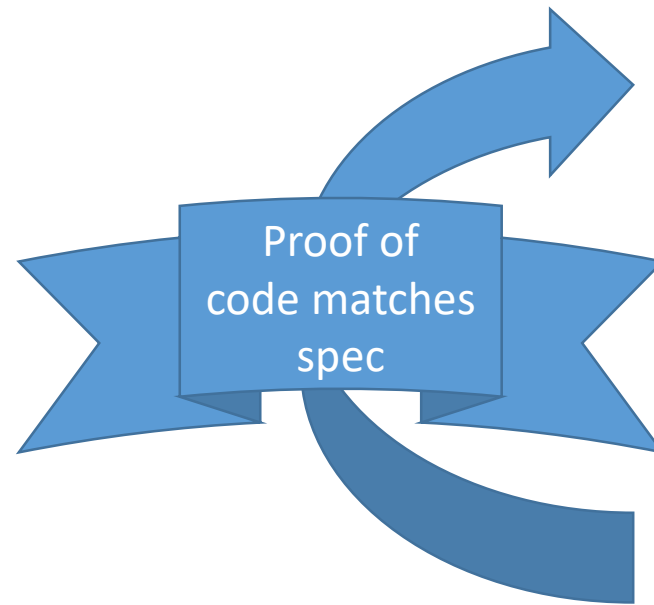
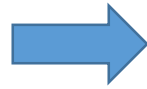
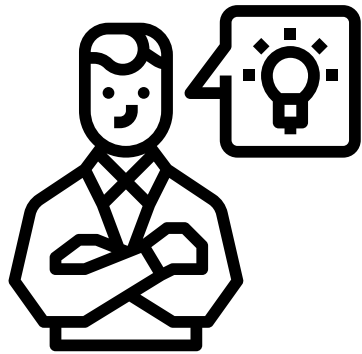


Takeaways

- Opportunity: Automate verification to enable innovation
- Big Problem: Asymptotic performance
- State-of-the-Art Methodologies
 - Abstraction & Reflection
 - Reflective Partial Evaluation
- Important Next Steps

Automating Verification: What?

Automating Verification: What?



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



Automating Verification: What is 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” or “trusted code base”)

Automating Verification: How?

Fiat Cryptography: The Goal

- Generate verified low-level cryptographic primitives
- Where this is important:



Fiat Cryptography: Desiderata

1. Code we verify must be fast and constant time

Justification: server load, security

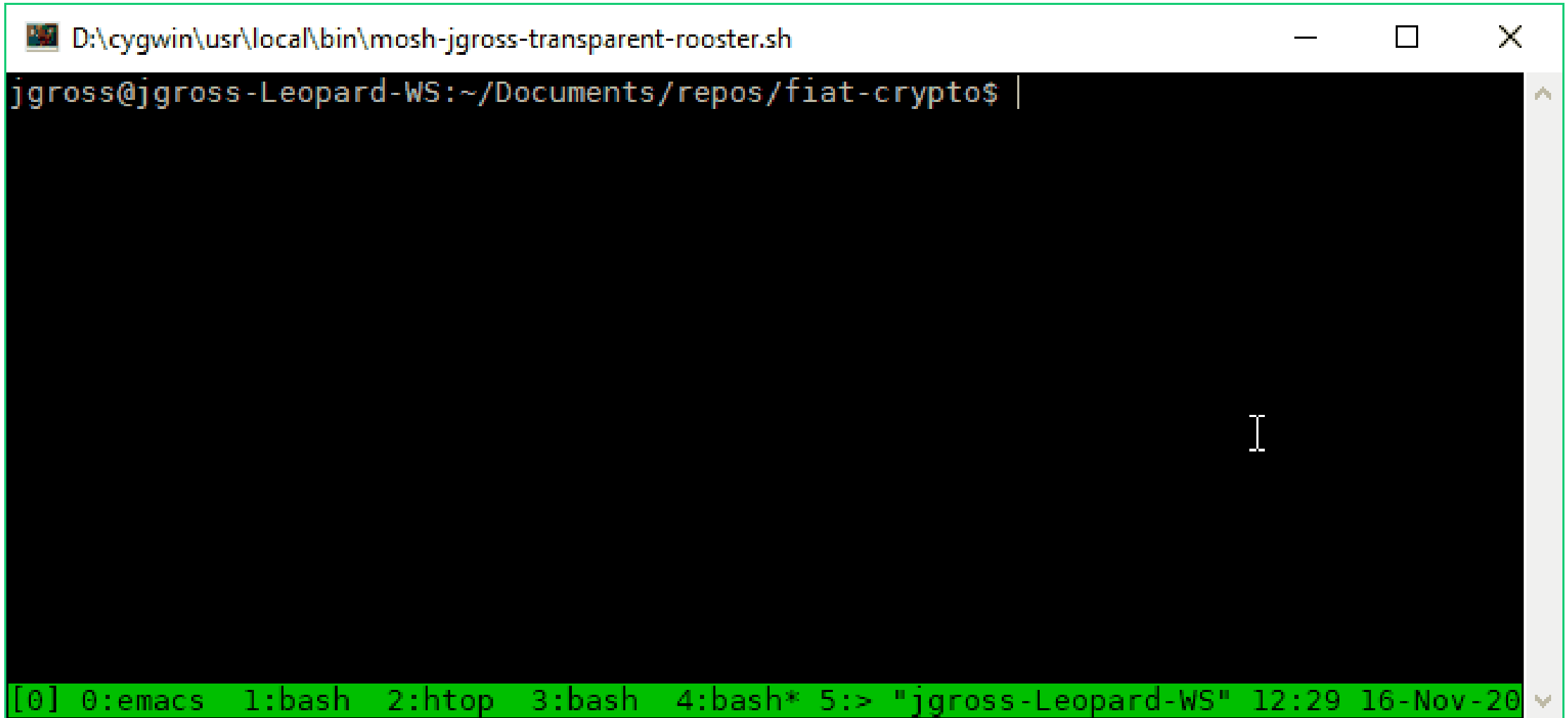
2. Easy to add and prove new algorithm, prime, architecture, ...

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

3. Verification should not run forever

Justification: usability for innovation

Spoiler: It Works Great!

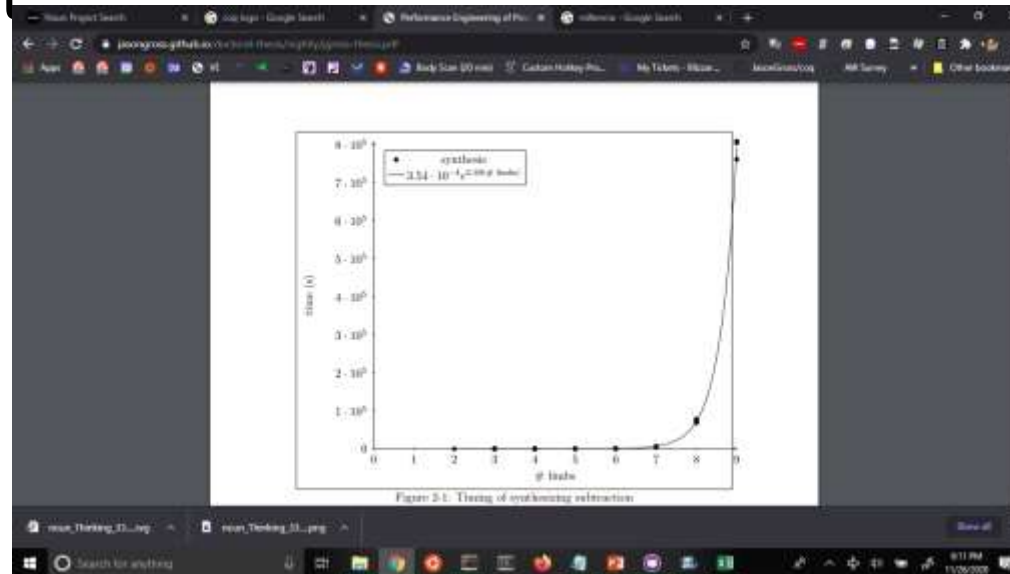


A terminal window titled "D:\cygwin\usr\local\bin\mosh-jgross-transparent-rooster.sh" with standard window controls. The prompt is "jgross@jgross-Leopard-WS:~/Documents/repos/fiat-crypto\$". The terminal area is mostly black with a white cursor. At the bottom, a green status bar displays: "[0] 0:emacs 1:bash 2:htop 3:bash 4:bash* 5:> "jgross-Leopard-WS" 12:29 16-Nov-20".

```
D:\cygwin\usr\local\bin\mosh-jgross-transparent-rooster.sh
jgross@jgross-Leopard-WS:~/Documents/repos/fiat-crypto$ |
[0] 0:emacs 1:bash 2:htop 3:bash 4:bash* 5:> "jgross-Leopard-WS" 12:29 16-Nov-20
```

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
- TODO: better plot



State-of-the-Art Methodology

- Abstraction to carve up the code into manageable pieces

Fiat Cryptography Pieces

Associational

Columns

Montgomery

Freeze

Bounds
Analysis

Positional

Rows

Barrett

Base
Conversion

Partial
Evaluation

Fiat Cryptography Pieces

Associational

Columns

Montgomery

Freeze

Bounds
Analysis

Positional

Rows

Barrett

Base
Conversion

Partial
Evaluation

The Big Problem in Automating Verification

- The problem is asymptotic performance

State-of-the-Art Methodology

- Abstraction to carve up the code into manageable pieces
- Reflection to handle the remaining pieces

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
- Reflection is about phrasing the goal in such a way that we can reduce it to validating the output of a computation
 - Example: compute the parity of a number; prove evenness by validating the computed parity
- Reflection is about verifying the process, rather than having an ad-hoc process that leaves behind a trail verifying the output

Takeaways

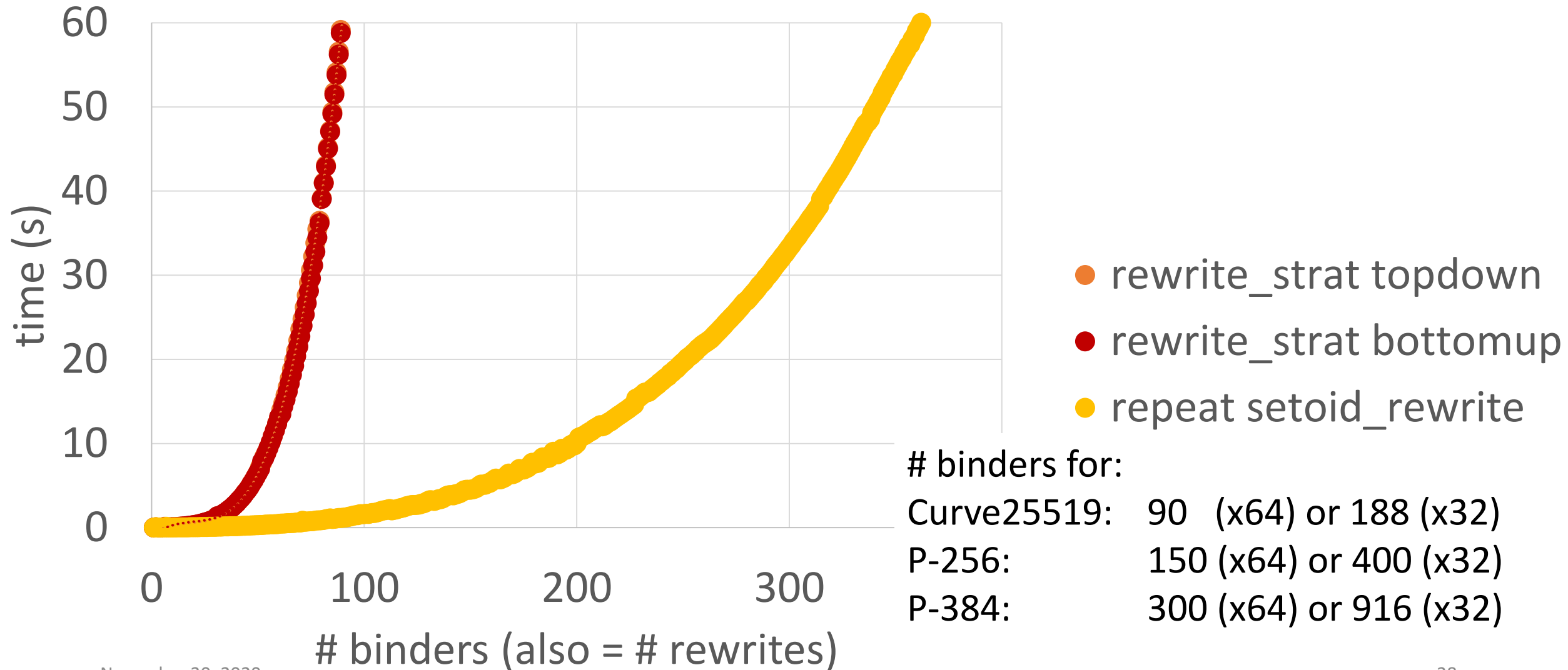
- Opportunity: Automate verification to enable innovation
- Big Problem: Asymptotic performance
- State-of-the-Art Methodologies
 - Abstraction & Reflection
 - Reflective Partial Evaluation
- Important Next Steps

Partial Evaluation and Rewriting

Partial Evaluation: What is it?

- Describe it
- TODO: nounproject it

Partial Evaluation: It's not Trivial



Partial Evaluation and Rewriting: Requirements

- β -reduction
- $\lambda\delta$ -reduction + rewrites
- code sharing preservation

Partial Evaluation and Rewriting:

Requirements: β -reduction

- Example of β -reduction with $((\lambda x. x + 5) 2)$
- Words about how termination is non-trivial and interesting
- Note that this is useful for eliminating function call overhead in the generated code, which is important for output code performance

Partial Evaluation and Rewriting:

Requirements: $\lambda\delta$ -reduction + rewrites

- Example of $\lambda\delta$ -reduction + rewrites with `(map (\lambda x. x + 5) [1; 2; 3])`
 - Note that this leaves β redexes
- Words about how making rewriting efficient and combining it with β -reduction in a way that scales is interesting (idk what to say here though)
- Words for arithmetic rewriting in fiat-crypto: without this we get quartic asymptotics of the # lines of code rather than merely quadratic, so it's not really acceptable to save for a later stage

Partial Evaluation and Rewriting:

Requirements: Code Sharing Preservation

- Example of let-lifting with $(\text{map } f (\text{let } y := x + x \text{ in let } z := y + y \text{ in } [z; z; z]))$
- to avoid exponential blowup in code size

Partial Evaluation and Rewriting: Requirements

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

Partial Evaluation and Rewriting: Obvious Extra Requirements

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

Partial Evaluation and Rewriting: Implementation

- Reflective so as to not extend the TCB and to perform fast enough
 - Side benefit: we can extract it to OCaml to run as a nifty command-line utility
- Normalization by Evaluation (NbE) (for β) + let-lifting (code-sharing) + rewriting ($\lambda\delta$ +rewrite)
 - Note that we use some tricks for speeding up rewriting such as pattern-matching compilation, on-the-fly emitting identifier codes so that we can use Coq's/OCaml's pattern matching compiler, pre-evaluating the rewriter itself
 - TODO: Spend some slides talking about these
- TODO: how much of this do I throw up ahead of time???

Partial Evaluation and Rewriting: Implementation

- Reflective so as to not extend the TCB and to perform fast enough
 - Side benefit: we can extract it to OCaml to run as a nifty command-line utility
- Talk about what reflection is, small TCB of Coq, checks proofs, interactive steps leave behind large proofs to justify their work. (Talk about how reflection is about verifying the process, rather than having an ad-hoc process that leaves behind a trail verifying the output. This is actually asymptotically faster in some cases such as rewriting because the trail of verification, unless done very cleverly, involves super-linear duplication of the term being rewritten. Also, proof building in Coq is so slow in general, both the asymptotics and to a lesser extent the constant factors, that even when we have to run the whole process again at Qed-time, reflection still comes out massively ahead performance-wise)

Partial Evaluation and Rewriting:

Implementation: Normalization by Evaluation

- Normalization by Evaluation (NbE) is for β
- Pull slides from RQE???
- Expression application \rightsquigarrow Gallina application
- Expression abstraction \rightsquigarrow Gallina abstraction
- Expression constants \rightsquigarrow rewriter invocations on η -expanded forms
-

Partial Evaluation and Rewriting: Implementation: Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

Partial Evaluation and Rewriting: Implementation: Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

Example: Turn “ $(\lambda z p n x. z + (x + (p + n))) 0 1 (-1)$ ” 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"))
```

Partial Evaluation and Rewriting: Implementation: Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

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

$(\lambda z p n x. (\lambda a b. \text{rewrite}("+", a, b))$
 $z ((\lambda a b. \text{rewrite}("+", a, b))$
 $x ((\lambda a b. \text{rewrite}("+", a, b))$
 $p n)))) (\text{rewrite}("0")) (\text{rewrite}("1")) (\text{rewrite}("-1"))$

Expression application \rightsquigarrow Gallina application

Expression abstraction \rightsquigarrow Gallina abstraction

Expression constants \rightsquigarrow rewriter invocations on η -expanded forms

Partial Evaluation and Rewriting: Implementation: Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

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

$(\lambda z p n x. (\lambda a b. \text{rewrite}("+", a, b))$
 $z ((\lambda a b. \text{rewrite}("+", a, b))$
 $x ((\lambda a b. \text{rewrite}("+", a, b))$
 $p n)))) (\text{rewrite}("0")) (\text{rewrite}("1")) (\text{rewrite}("-1"))$

Then reduce!

Partial Evaluation and Rewriting: Implementation: Let-Lifting

- For code-sharing-preservation
- Some words about LetIn monad
- Some words about re-doing NbE in the LetIn monad, which is like the CPS monad
- Haven't seen it in the literature, but it's not too tricky

Partial Evaluation and Rewriting:

Implementation: Rewriting

- For $\iota\delta$ +rewrite
- Perhaps the most interesting component is that fusing rewriting with NbE in PHOAS allows us to delay rewriting and 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”.)

Partial Evaluation and Rewriting:

Implementation: Rewriting: More Features

- We select which rewrite rule to use based on Coq's pattern matching, which means that we don't need to walk the entire list of rewrite rules at every identifier/constant node just to see which ones apply
- We enable this efficiency by on-the-fly emission of a type of codes for the constants we care about (seems like a new way of doing things not present elsewhere in the literature)
- We further gain efficiency (about 2x) by doing partial evaluation on the generated rewriter itself (using Coq's built-in mechanisms)

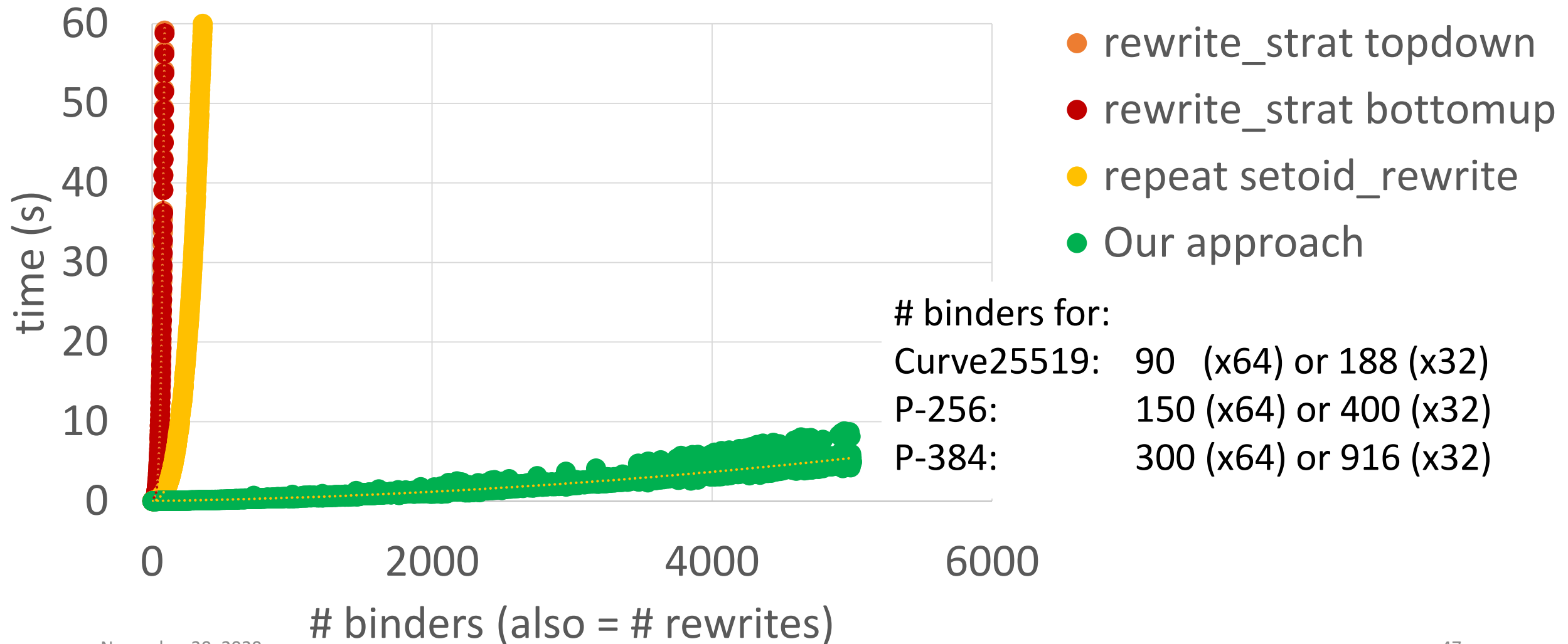
Partial Evaluation and Rewriting: Implementation

- Reflective so as to not extend the TCB and to perform fast enough
 - Side benefit: we can extract it to OCaml to run as a nifty command-line utility
- Normalization by Evaluation (NbE) (for β) + let-lifting (code-sharing) + rewriting ($\lambda\delta$ +rewrite)
 - Note that we use some tricks for speeding up rewriting such as pattern-matching compilation, on-the-fly emitting identifier codes so that we can use Coq's/OCaml's pattern matching compiler, pre-evaluating the rewriter itself
 - TODO: Spend some slides talking about these
-

Partial Evaluation and Rewriting: Evaluation

- It works!
- It's performant!
- It seems like it would also solve one of the two performance issues that killed the parser-synthesizer I worked on for my masters.

Partial Evaluation and Rewriting: Performance



Takeaways

- Opportunity: Automate verification to enable innovation
- Big Problem: Asymptotic performance
- State-of-the-Art Methodologies
 - Abstraction & Reflection
 - Reflective Partial Evaluation
- Important Next Steps

Takeaways

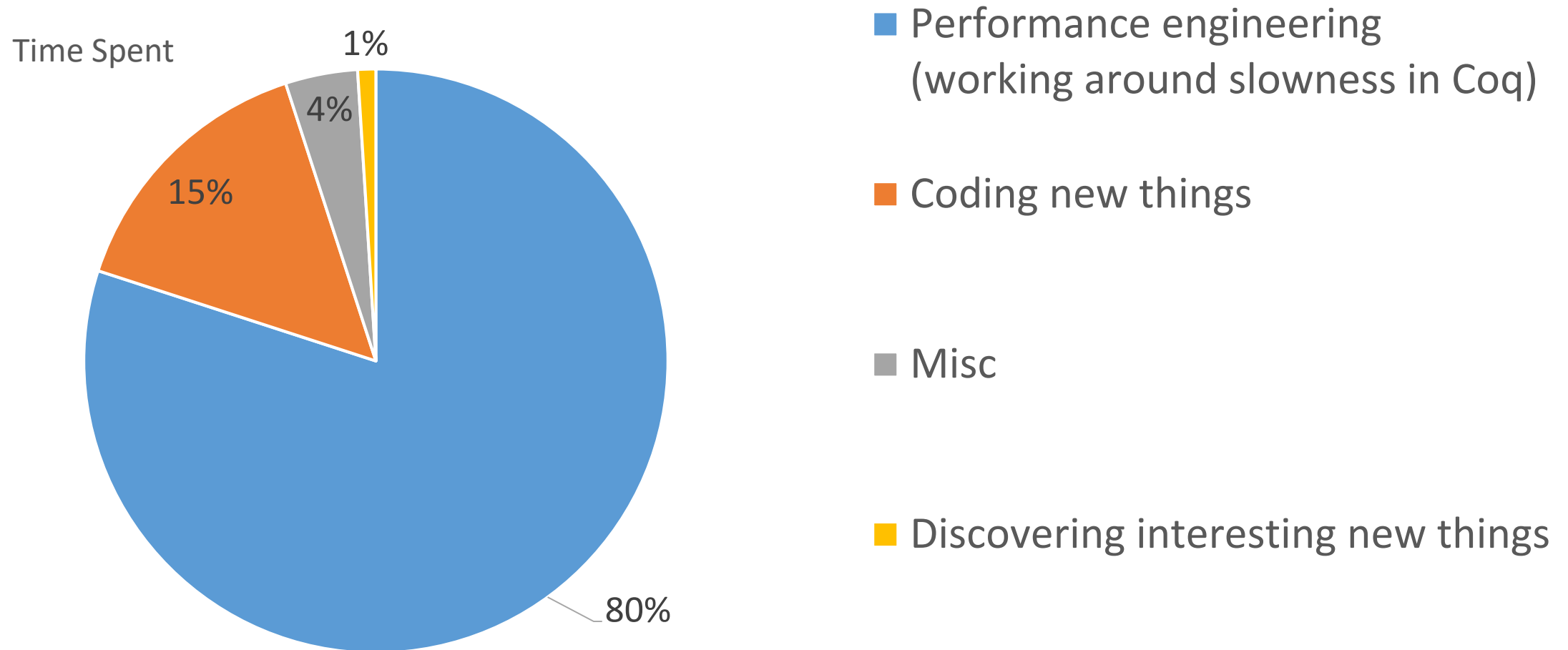
- Opportunity: Automate verification to enable innovation
- Big Problem: Asymptotic performance
- State-of-the-Art Methodology:
 - Abstraction: to carve the problem into manageable pieces
 - Reflection: to handle the remaining parts that are too big
- Important Next Steps
 - “We need to get the basics right”

Automating Verification: Next Steps

Let's take a step back

- We succeeded, but this was very hard
- Stats about weeks of work and lines of code changed in rewriter
- All of this to work around inadequate asymptotic performance of the proof engine
- This is typical!

What I did in my PhD



What we've been doing when performance is bad

- We've been carving out the proof engine
- And replacing it with reflection

Reflection will not save us

- Using a proof assistant is for inserting human ingenuity
- Using reflection is essentially giving that up
- 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
- It's not just accident; there are good reasons that obvious solutions have the wrong asymptotics

Aside: Why did reflection help at all?

- Reflection helps *because* it's solving a more limited problem
- This means reflective proof engine isn't enough
- Power and performance: choose one

Automating Verification: Next Steps

- As a field, we need to study proof engines with an eye towards asymptotic performance
- “Don’t make stupid choices” isn’t enough to get good asymptotic performance
 - Try to write a version of `rewrite_strat` inside the tactic engine in a way where every step can be considered as progress towards proving something, in a way that is linear in # of binders + # of rewrite locations + size of term
 - Highly non-trivial!

Proof Engines: Next Questions

- Adequate set of primitives?
- Adequate asymptotics?
- How do we evaluate adequate?
- Is it possible to achieve adequate performance simultaneously on all the primitives?
 - With backtracking?
- What are the requirements on something to be a “proof engine”?
 - My current take is “every step makes partial progress towards proving something” and “error messages about proof validity are local”
- Where does the overhead actually come from?
- What things are people not currently doing due to performance overhead?

Proof Engines: Future-Oriented Takeaways

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

Takeaways

- Opportunity: Automate verification to enable innovation
- Big Problem: Asymptotic performance
- State-of-the-Art Methodologies
 - Abstraction & Reflection
 - Reflective Partial Evaluation
- Important Next Steps

Thank you for your time and
attention!

Questions?