

# Performance Engineering of Proof-Based Software Systems at Scale

Jason Gross

Ph.D. Defense

MIT CSAIL

# Outline

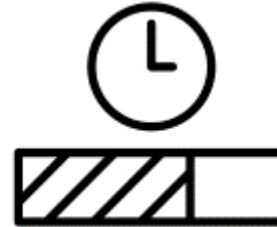
## The Problem



November 30, 2020

Photo by Benjamin Dreyer from Mass Project

## Progress on the Problem: Fiat Cryptography



November 30, 2020

Image by Lili from the Mass Project

30

## Takeaways: What's Working



November 30, 2020

Image by Lili from the Mass Project (2020) has been modified

31

## Takeaways: What Needs to Change



November 30, 2020

Image by Lili from the Mass Project (2020) has been modified

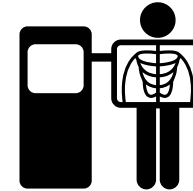
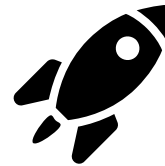
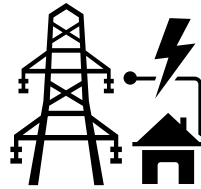
32

# The Problem

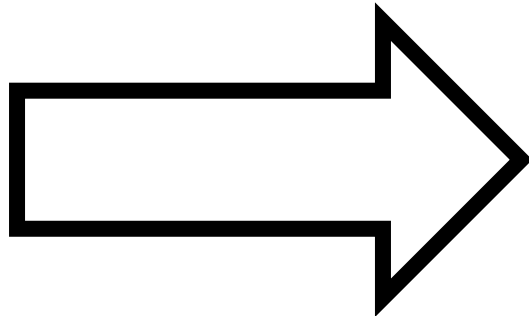
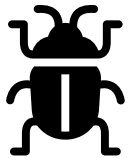


# Why Verification?

- Software is important



- Bugs are bad



# Why Verification?

- Software is important
- Bugs are bad
- Verification promises a fix

# The Current State

- Impressive existing verification, including:
  - CompCert (C compiler)
  - seL4 (microkernel)
  - CertiKOS (operating system)
  - Fiat Cryptography (cryptographic primitives)

# The Current State: What's the Scale?

- Impressive existing verification, including:

- CompCert

5880

- seL4

8700

- CertiKOS

6500

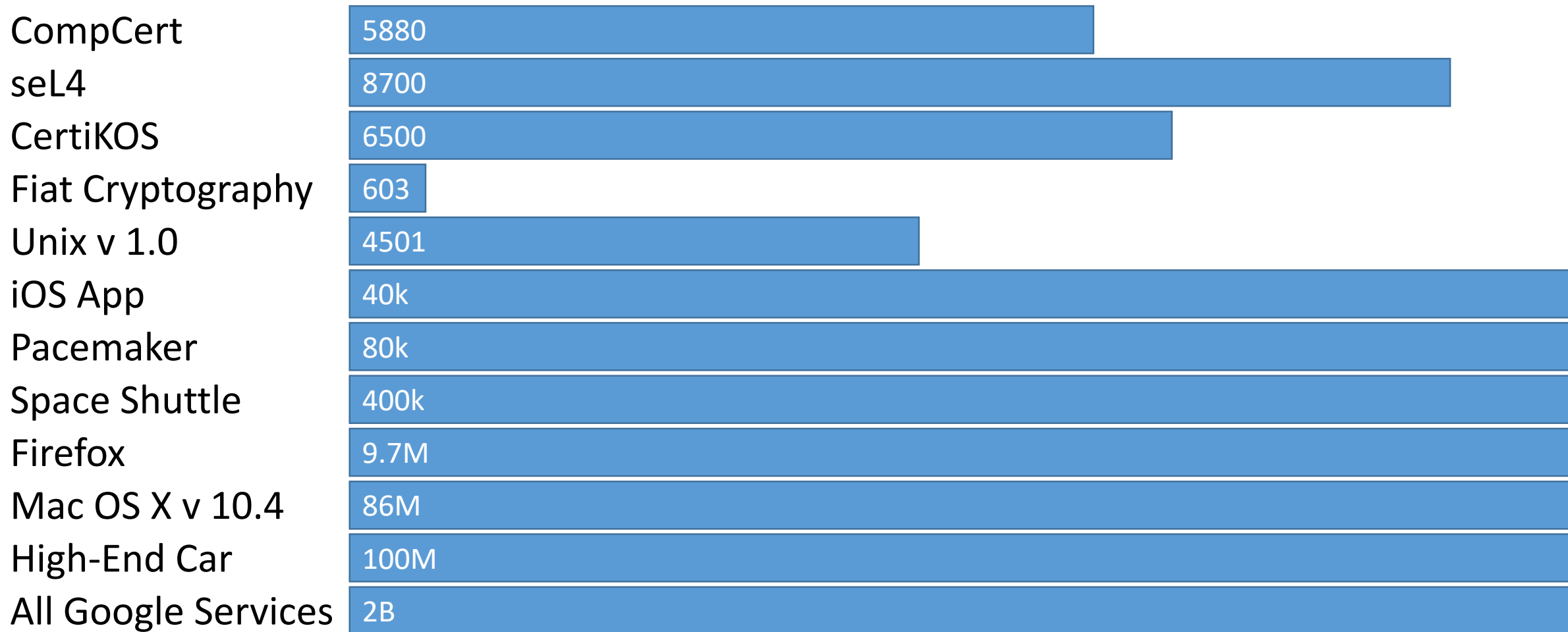
- Fiat

603

Cryptography

Lines of Code Being Verified

# Problem: The Gap Between Academia & Industry



November 30, 2020

Lines of Code



# The Current State: What's the Scale?

- Impressive existing verification, including:

- CompCert



- seL4



- CertiKOS



- Fiat Cryptography



Lines of Code Being Verified

# The Current State: What's the Scale?

- Impressive existing verification, including:

- CompCert



- seL4



- CertiKOS



- Fiat Cryptography



# The Current State: What's the Scale?

- Impressive existing verification, including:

- CompCert

- seL4

- CertiKOS

- Fiat Cryptography



Lines of Code Being Verified

Lines of Specification

# The Current State: What's the Scale?

- Impressive existing verification, including:

- CompCert



- seL4



- CertiKOS



- Fiat Cryptography



# The Current State: What's the Scale?

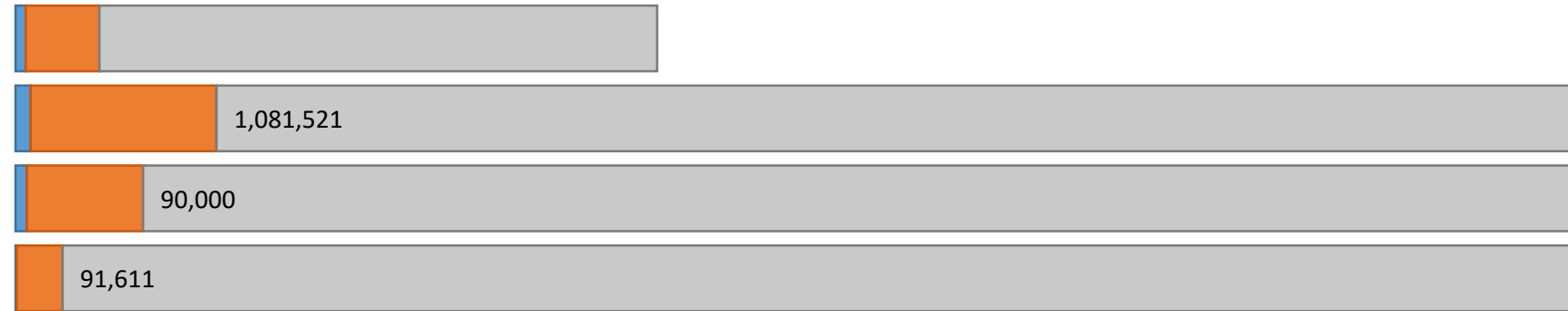
- Impressive existing verification, including:

- CompCert

- seL4

- CertiKOS

- Fiat Cryptography



# The Current State: What's the Scale?

- Impressive existing verification, including:

- CompCert



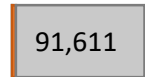
- seL4



- CertiKOS



- Fiat Cryptography



# The Current State: What's the Scale?

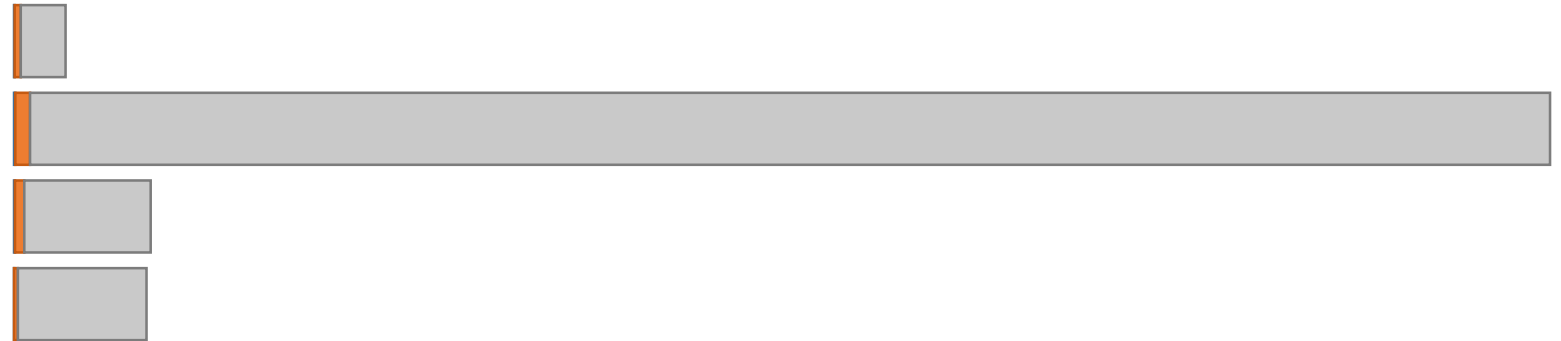
- Impressive existing verification, including:

- CompCert

- seL4

- CertiKOS

- Fiat Cryptography



# The Current State: What's the Scale?

- Impressive existing verification, including:

- CompCert

5880

- seL4

8700

- CertiKOS

6500

- Fiat Cryptography

603

Lines of Code Being Verified



# The Current State: What's the Scale?

- Impressive existing verification, including:

- CompCert

5880

- seL4

8700

- CertiKOS

6500

- Fiat Cryptography

603

- HERE

# The Current State: What's the Scale?

- Impressive existing verification, including:

- CompCert



- seL4



- CertiKOS



- Fiat Cryptography



# The Current State: What's the Scale?

- Impressive existing verification, including:

- CompCert

5880

- seL4

8700

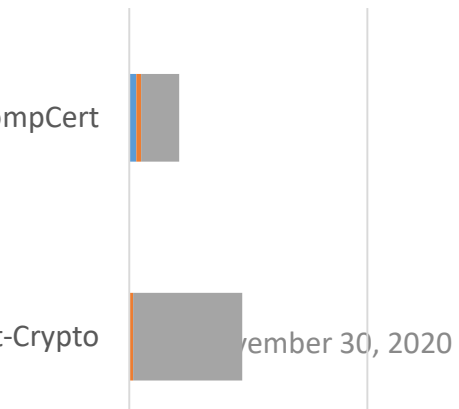
- CertiKOS

6500

- Fiat Cryptography

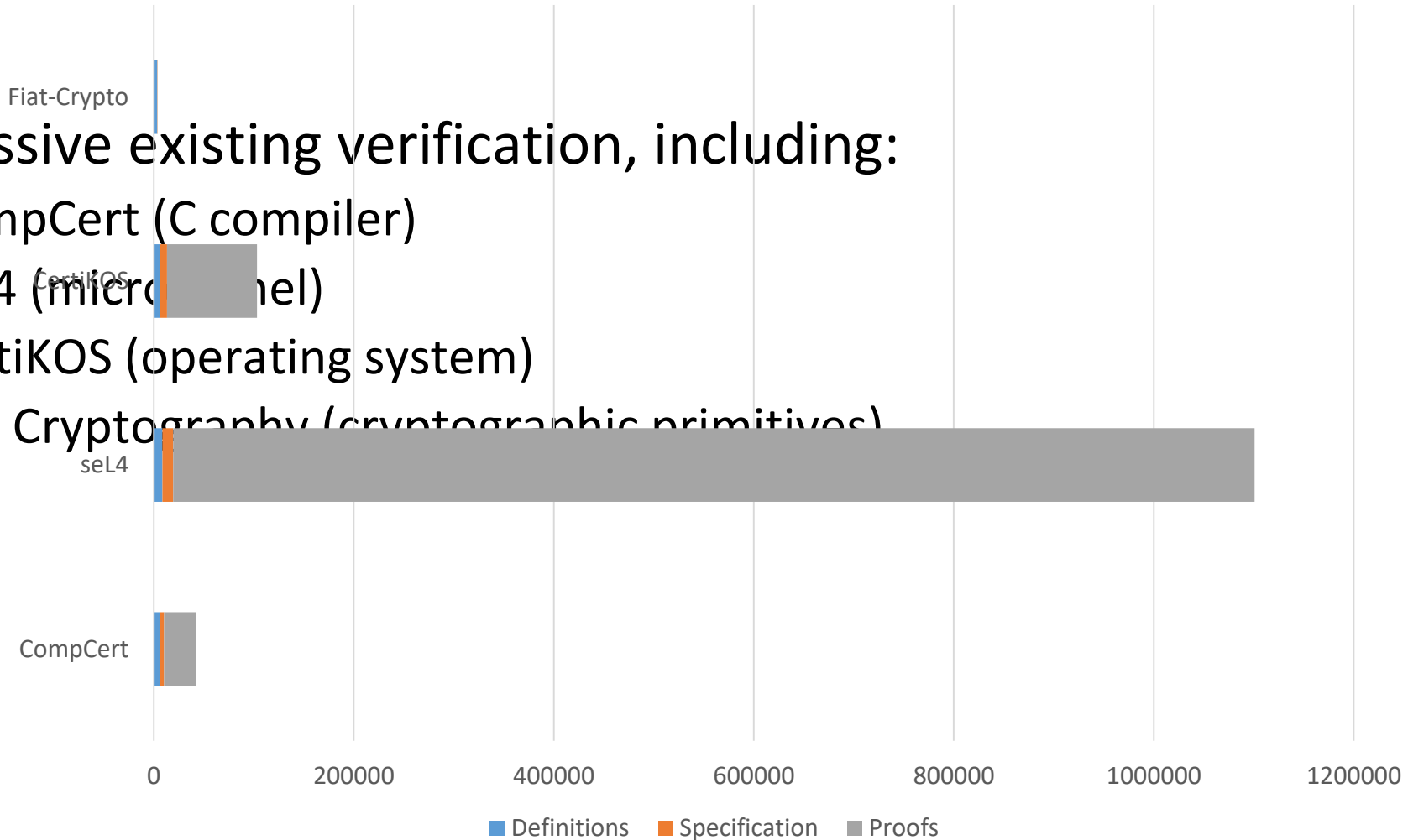
603

Chart Title

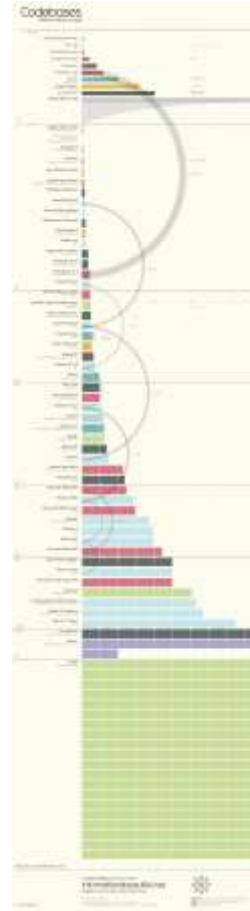


# The Current State: What's the Scale?

- Impressive existing verification, including:
  - CompCert (C compiler)
  - seL4 (microkernel)
  - CertiKOS (operating system)
  - Fiat Cryptography (cryptographic primitives)

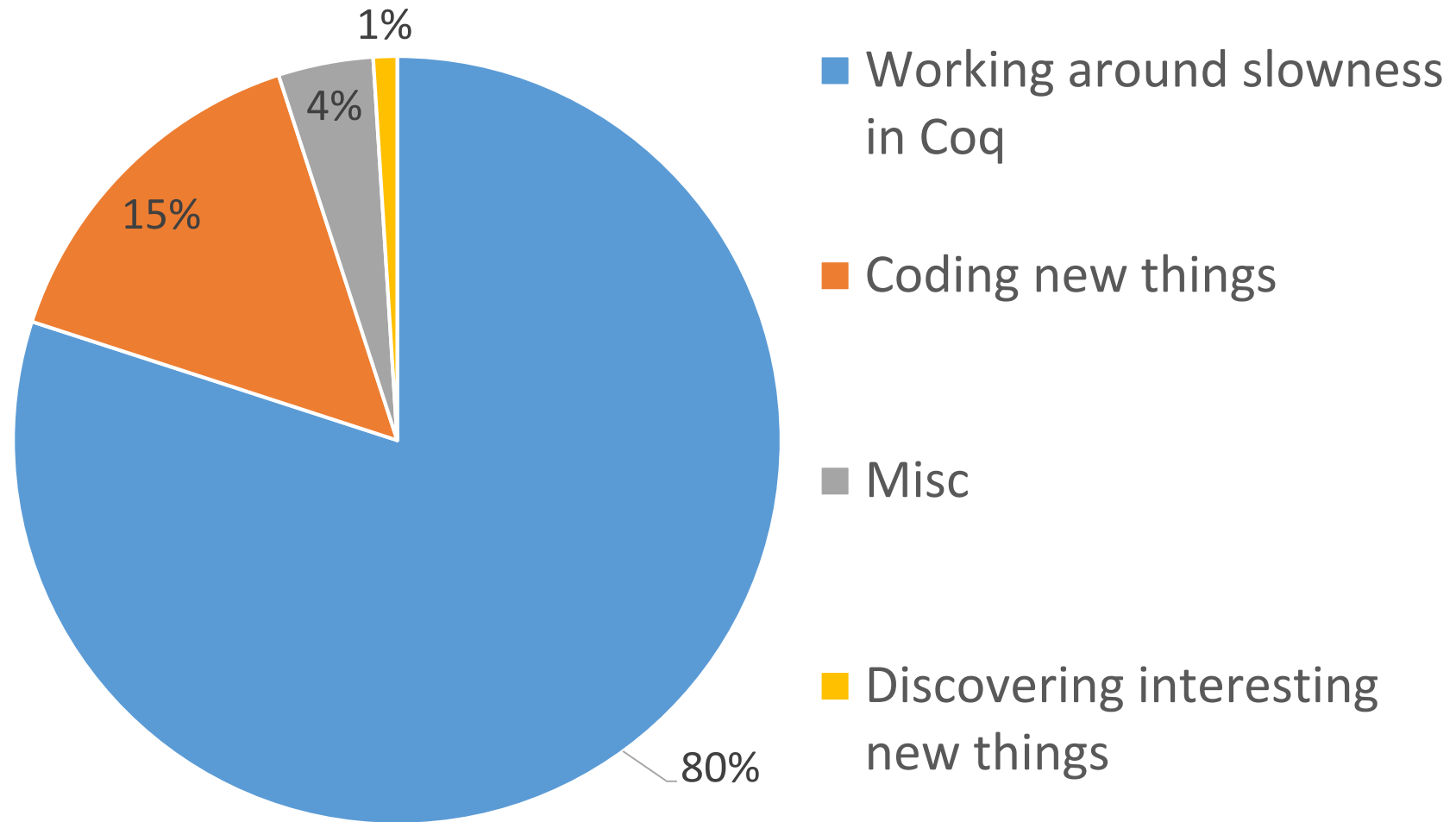


# Problem: The Gap Between Academia & Industry



# Scale of proof in verification

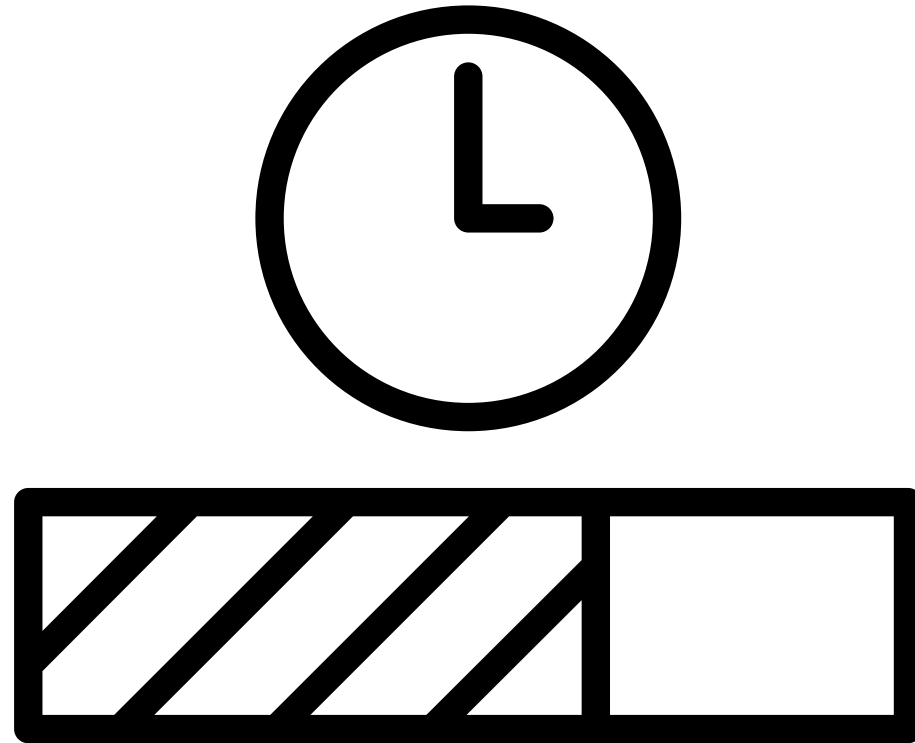
# Time Spent in my PhD



# Super-linear scaling



# Progress on the Problem: Fiat Cryptography



# 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

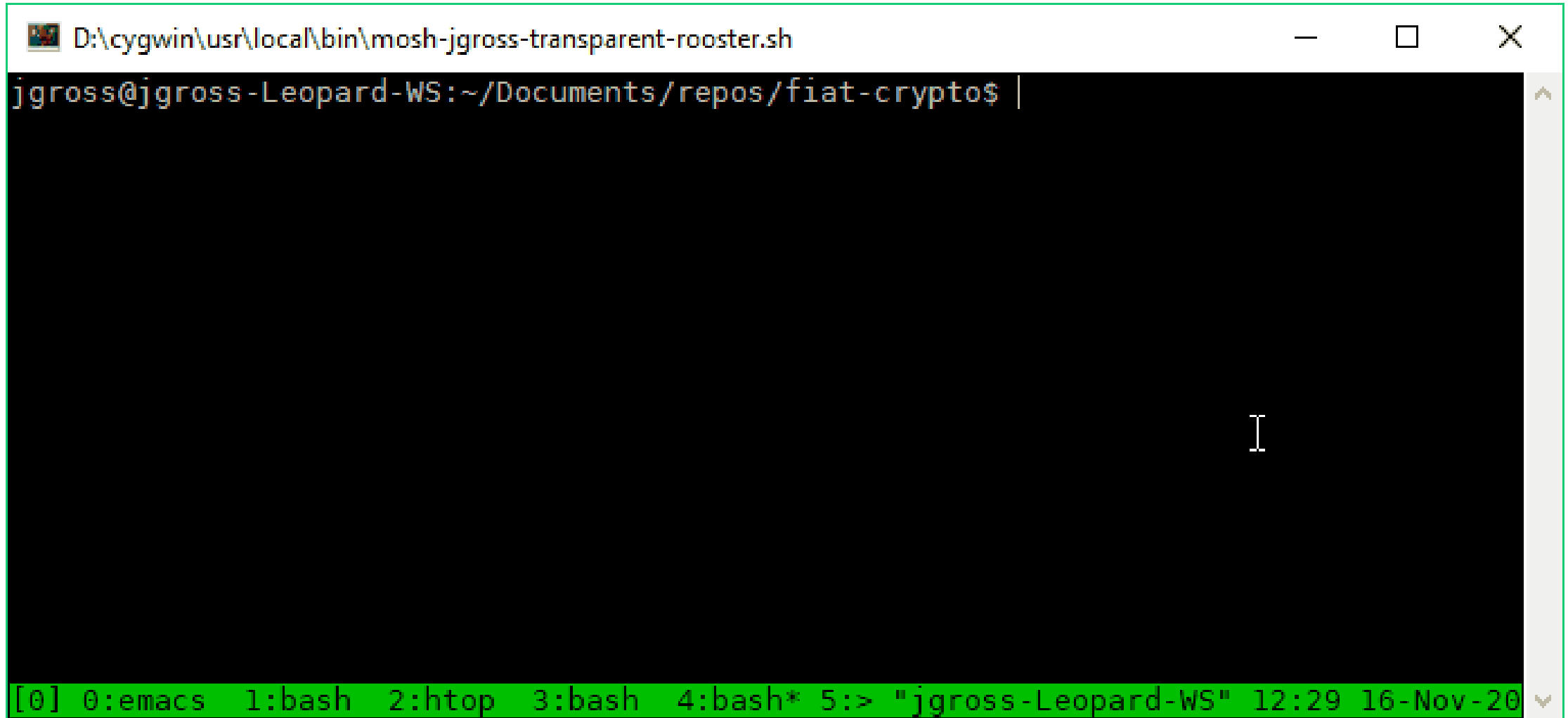
2. Easy to add and prove new algorithm, prime, architecture, ...

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

3. Coq should not run forever

Justification: Obvious

# 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/flat-crypto\$". The terminal area is mostly black with a white cursor. A green status bar at the bottom shows: "[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/flat-crypto$ |
[0] 0:emacs 1:bash 2:htop 3:bash 4:bash* 5:> "jgross-Leopard-WS" 12:29 16-Nov-20
```

# Fiat Cryptography: Methodology

Associationa  
l

Columns

Montgomer  
y

Freeze

Bounds  
Analysis

Positional

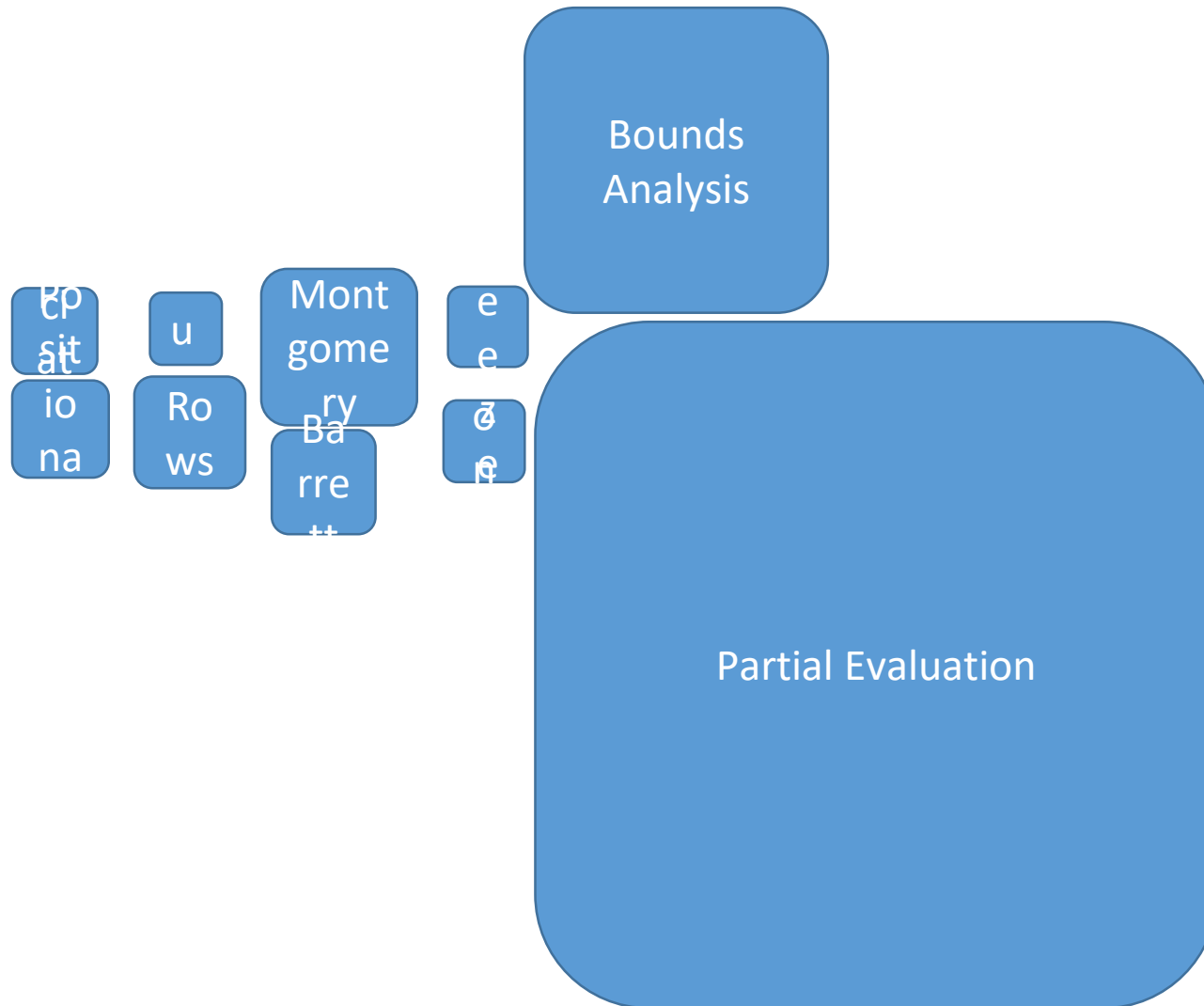
Rows

Barrett

Base  
Conversion

Partial  
Evaluation

# Fiat Cryptography: Methodology



# Fiat Cryptography: Methodology

- Carve up the low-level code into neatly-separated conceptually distinct units that are small enough to not hit asymptotic issues during interactive verification
  - My colleagues did most of this, though I helped them see how factoring and abstraction impact performance of verification effort

TODO: describe abstraction in terms of excessive unfolding, either here or elsewhere

- The remaining chunk is *partial evaluation*

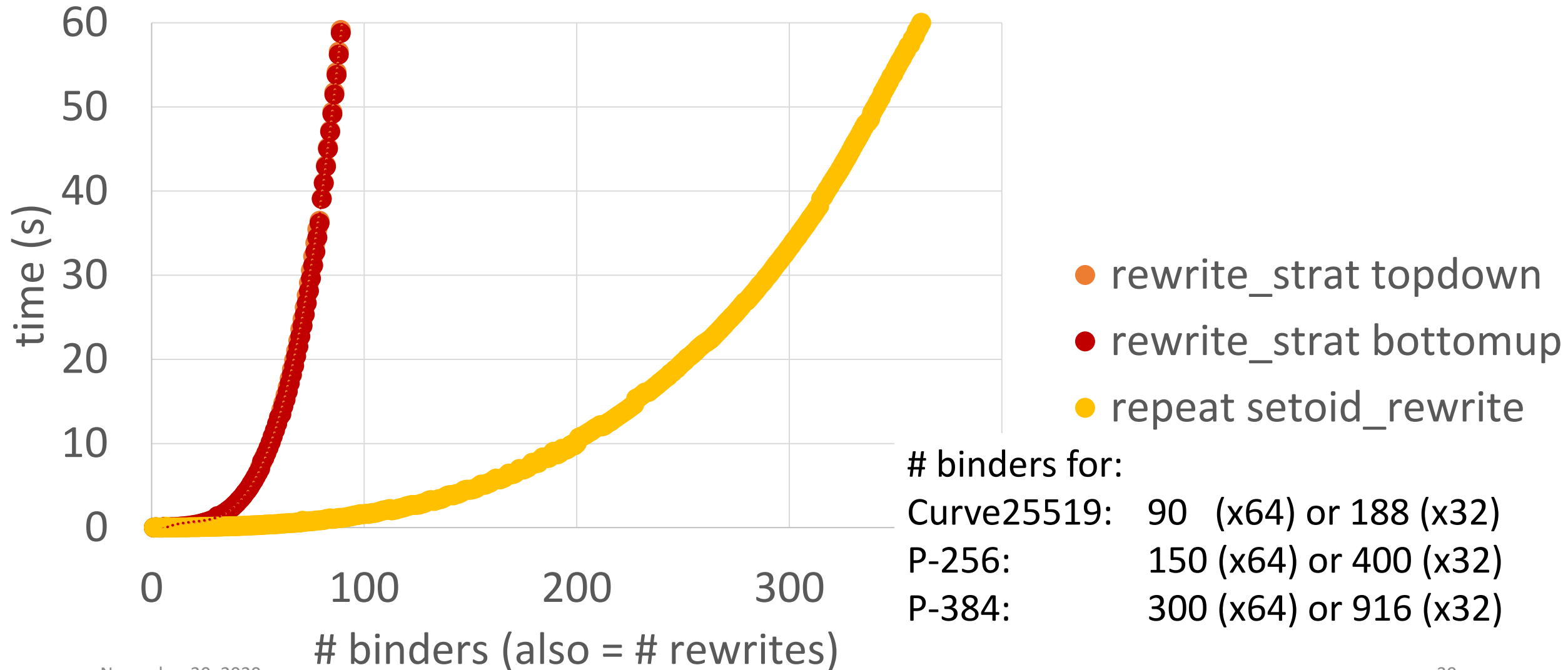
# Partial Evaluation and Rewriting



# Partial Evaluation: What is it?

- Describe it

# Partial Evaluation: It's not Trivial



# Partial Evaluation and Rewriting: Requirements

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

# Partial Evaluation and Rewriting:

## Requirements: $\beta$ -reduction

- Example of  $\beta$ -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  $\beta$  redexes
- Words about how making rewriting efficient and combining it with  $\beta$ -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

- $\beta$ -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: Extra Obvious 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  $\beta$ ) + 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  $\beta$
- Pull slides from RQE???
- Expression application  $\rightsquigarrow$  Gallina application
- Expression abstraction  $\rightsquigarrow$  Gallina abstraction
- Expression constants  $\rightsquigarrow$  rewriter invocations on  $\eta$ -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  $\eta$ -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)

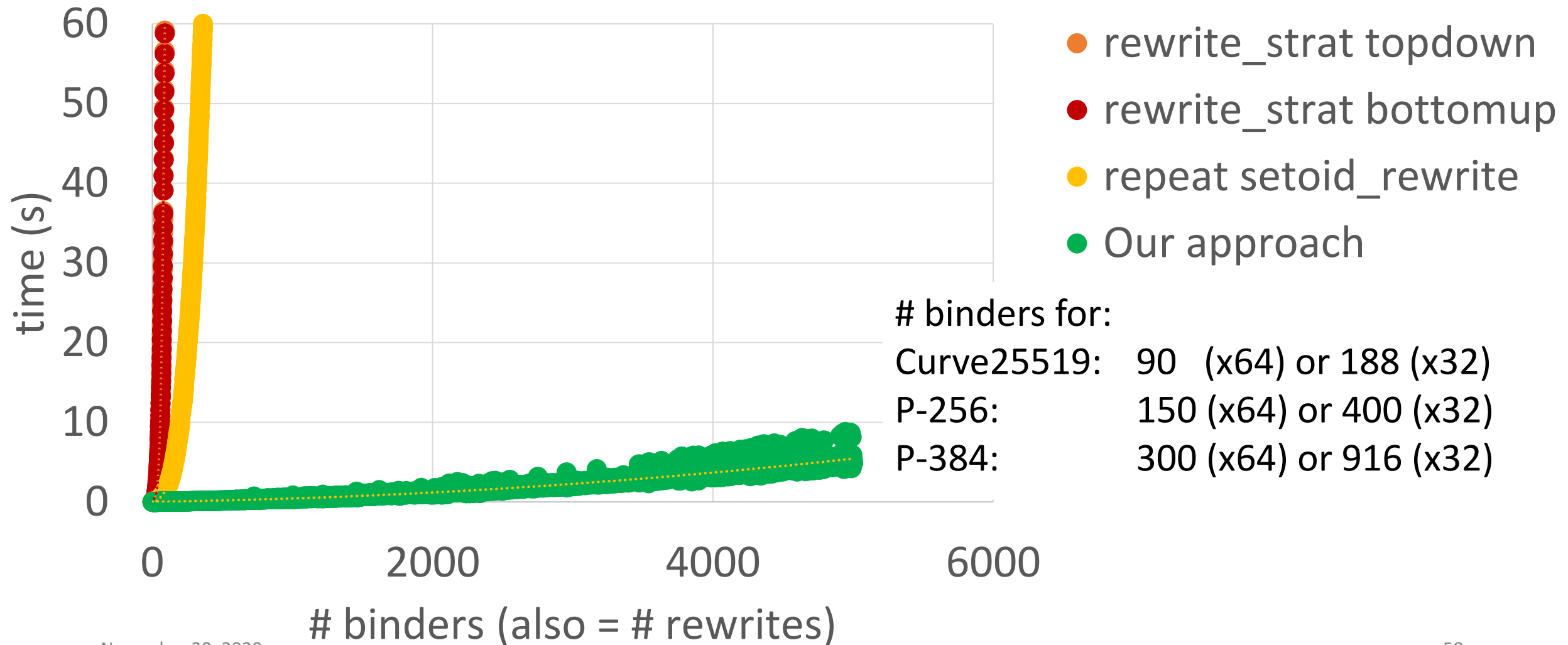
# Partial Evaluation and Rewriting: Implementation

- 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  $\beta$ ) + 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: What's Working

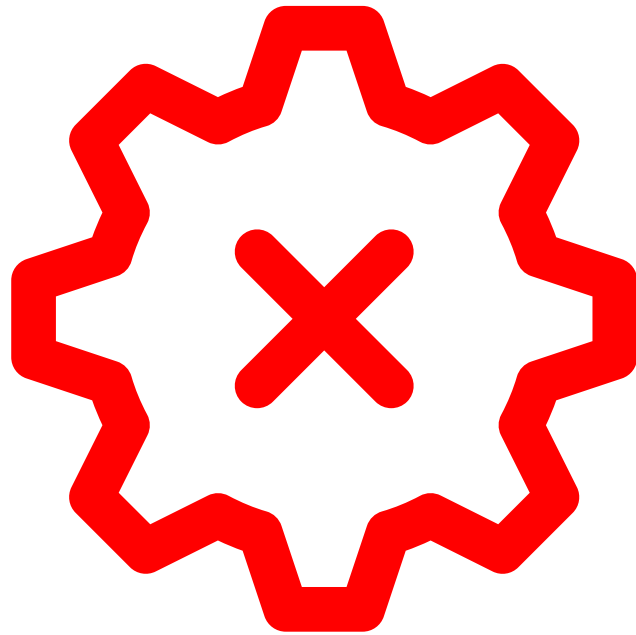


# Takeaways

- Asymptotic scaling of interactive proof assistant response is a real problem!
- Current method---of working around it by breaking the problem into small enough chunks and solving remaining chunks with specialized reflective solutions---does seem to work.
  - I'm not optimistic about being able to break things down enough to handle them all interactively; even in very abstract math (category theory), I ran into issues where the way mathematicians did it was not sufficient. Furthermore basically all code that needs to be fast, and a great deal of systems code, inlines definitions in a way that causes performance issues.
  - And reflective automation requires enormous investment of effort for each new problem.
  - Because reflection is not a proof engine made of small pieces that can each be said to make progress towards proving a goal, it's not easy to mix-and-match.
  - I needed let-lifting; prior work had already done NbE and reflective rewriting, but fusing them with let-lifting in a performant way seems to have required re-engineering them almost from scratch.

# Takeaways:

## What Needs to Change





# Takeaways: What Needs to Change

- Does it have to be this way?
  - No! (I hope)
  - No one seems to be studying why proof engines are asymptotically slow!
  - It's not just accident; there are good reasons that obvious solutions have the wrong asymptotics, and there's so much going on that it's not even clear yet what the specification of "adequate performance" is.
  - 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.

# Q & A

Thank you for your time and  
attention!

Questions?