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

Note that I'll be talking in the context of interactive dependently typed tactic driven proof assistants, because human ingenuity is important.
Come up with a better way to phrase this

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

Note that crypto is cryptography as in secret messages and bank security, not as in DogeCoin

TODO: Question for Adam: what to put here so people don't box this as SAT/SMT/model checking
Mention that we're doing this restriction for getting human ingenuity into the process

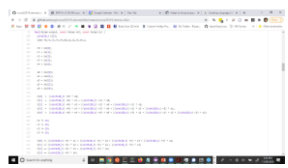
Automating Verification: Why?

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

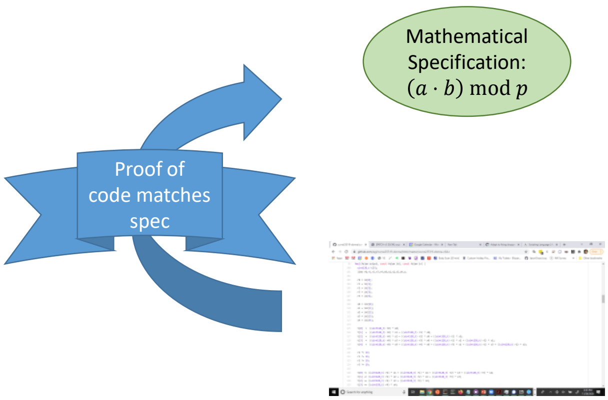
- Reduce costs (server & user)
- More mathematical security
- Keeping up with more powerful attackers

November 30, 2020 5

TODO: Better crypto code screen shot

Arms race where we need to be 100% correct
=> need to keep making new and better crypto as computers get more powerful
Verification will let us deploy new crypto with confidence (c.f. 100% correct)

The Promise of Verification



Verification will let us deploy new crypto with confidence by virtually eliminating the possibility of bugs in the code

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

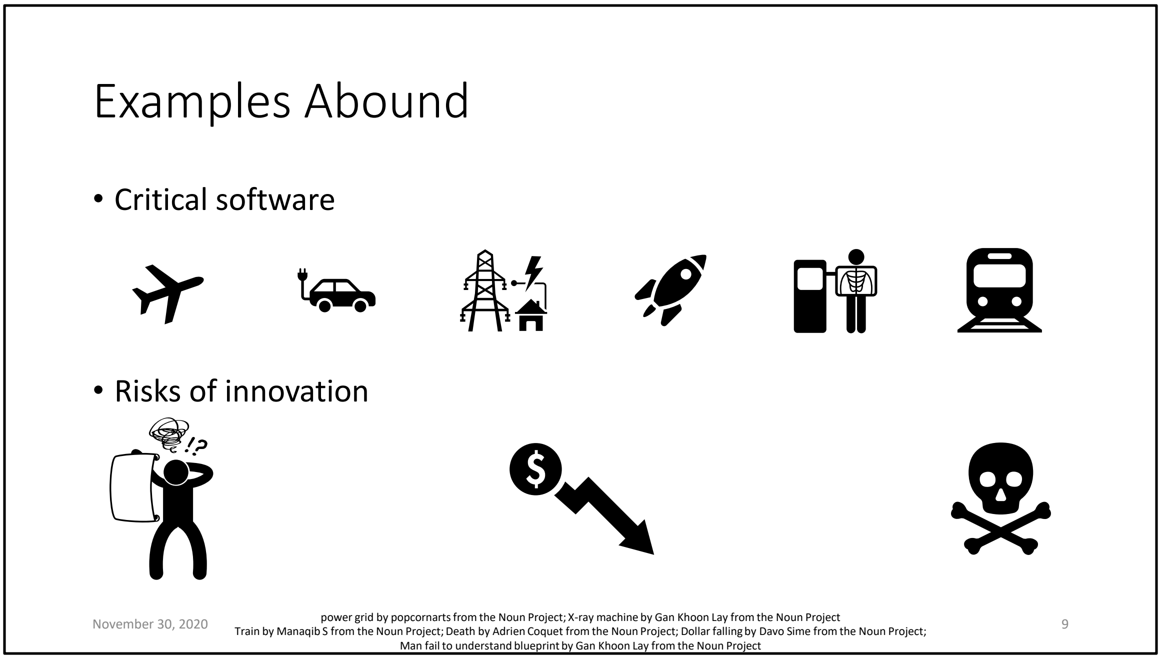
Automation allows us to pay the same upfront cost, but eliminate most of the marginal cost of innovating and deploying new things.
In fiat-crypto, the project that I worked on, automation allows innovation with some parameters, such as the prime modulus, costlessly---there’s no overhead to getting verified code for a new prime.
Additionally, we shrink the overhead of verifying new algorithms to 1×—2× of proof overhead, and the specs are only a couple hundred lines.
This is a big win!

```
$ git ls-files 'fiat-*/*.'{rs,c,java,go} | xargs cloc
      69 text files.
      69 unique files.
       0 files ignored.

github.com/AlDanial/cloc v 1.74  T=0.72 s (95.7 files/s, 282432.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                               31            925           5776          85346
```

```
Rust                  17        327        3080        39837
Go                    15        310        3339        39726
Java                   6        145        1430        23456
---------------------------------------------------------------------------
SUM:                  69       1707       13625       188365
---------------------------------------------------------------------------
```

```
$ echo "defn$(printf '\t')$(coqwc src/Arithmetic/Core.v | head -1)"; for i in $(git ls-files 'src/Arithmetic/*.v'); do echo "$(cat $i | tr '\n' '~' | sed s'/`/,/g' | sed s'/\.[ ~]/`/g' | grep -o 'Definition [^`]*`' | tr '~' '\n' | wc -l)$(printf '\t')$(coqwc $i | tail -1)"; done | sort -h
defn      spec    proof comments
0         102     152       4 src/Arithmetic/MontgomeryReduction/Proofs.v
0          39      39      36 src/Arithmetic/BarrettReduction/Wikipedia.v
0          62      99      38 src/Arithmetic/BarrettReduction/Generalized.v
2          52      71       3 src/Arithmetic/Partition.v
3         181      31       3 src/Arithmetic/ModOps.v
3          49      53      38 src/Arithmetic/BarrettReduction/HAC.v
3          58     133       2 src/Arithmetic/UniformWeight.v
4         134     177      12 src/Arithmetic/ModularArithmeticTheorems.v
7          29      75       1 src/Arithmetic/Primitives.v
7          37      77       3 src/Arithmetic/FancyMontgomeryReduction.v
8          70     163      37 src/Arithmetic/PrimeFieldTheorems.v
18        181     135      14 src/Arithmetic/Freeze.v
35         53      75       1 src/Arithmetic/ModularArithmeticPre.v
39        225     101      21 src/Arithmetic/BaseConversion.v
39         69       0      91 src/Arithmetic/MontgomeryReduction/Definition.v
46        104     172      65 src/Arithmetic/BarrettReduction/RidiculousFish.v
50        197     335      14 src/Arithmetic/BarrettReduction.v
63        482     721      12 src/Arithmetic/WordByWordMontgomery.v
66        357     503      50 src/Arithmetic/BYInv.v
123       552     409      34 src/Arithmetic/Saturated.v
153       512     407      72 src/Arithmetic/Core.v
```



TODO: maybe make loc graphs here with icons

Software important
Bugs bad:
People sad
Money die
People go bye-bye

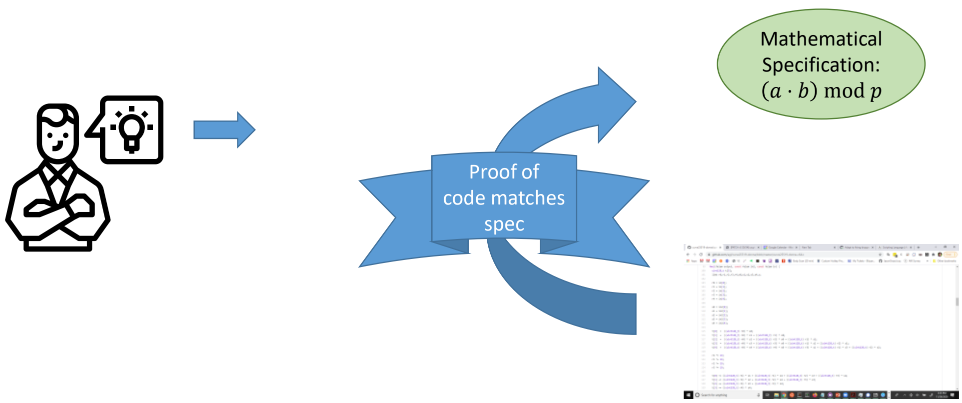
Takeaways

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

TODO: do bold animation thing

Automating Verification: What?

Automating Verification: What?



Thinking by ArmOkay from the Noun Project

Automating verification means writing a script that uses any needed human ingenuity to generate a proof that the code matches the spec.
The computer checks the proof to ensure correctness.
TODO: work on graphic

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”)

I’m now going to briefly talk about the last piece of context for the big problem of asymptotic performance; that piece is the proof engine.
A proof engine lets you declare a goal to prove; issue instructions to make partial progress on proving (interactive; lets you insert ingenuity);
can write scripts to automate issuing of instructions
tracks the progress made and where you are; can issue a trail to be checked by a small kernel / TCB

Automating Verification: How?

Now we move into the "how" of automating verification, which is where the big problem of asymptotic performance shows up.

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

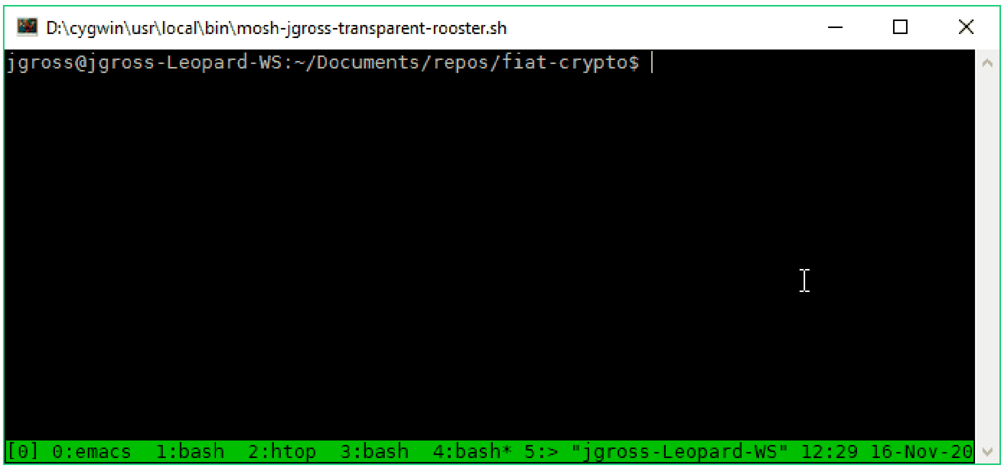
3. Verification should not run forever

   Justification: usability for innovation

November 30, 2020 16

Alternate 3rd: Verification should complete in reasonable time
Alternate 3rd justification: Needs to be checkable in time for industry deadlines, in time to be usable
Alternate 3rd: Coq should not run forever
Alternate 3rd justification: Obvious

Spoiler: It Works Great!

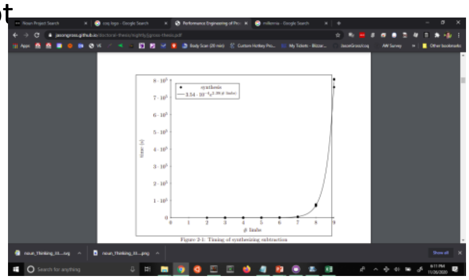


November 30, 2020 17

Our output artifact is actually pretty cool, and can automatically generate basically verified code on the command line, in seconds (not hours or days or weeks), for given just the prime, the bitwidth, and the name of the high-level algorithm

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

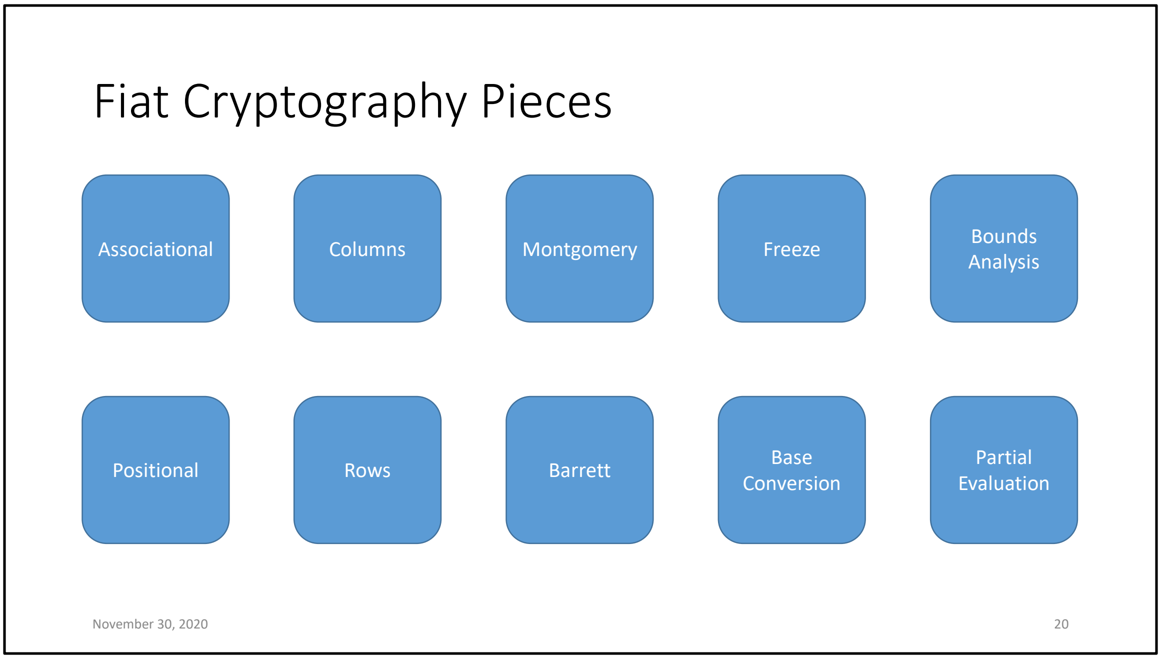


(after bullets)
For example, in an earlier version of fiat-crypto, the automated verification that worked fine on an example of size 2, taking 17 seconds, but on an example of size 17, was projected to take over 4,000 millennia!
TODO: dig up numbers and maybe insert plot?

State-of-the-Art Methodology

- Abstraction to carve up the code into manageable pieces

What I mean by manageable here is that we need to carve the code up into pieces that are small enough that we can actually succeed in writing automation for them, or, if the piece is small enough and conceptually coherent and simple enough, we don't need to automate its verification because the overhead will already be small.



In fiat-crypto, we carved the low-level code up into these neatly-separated conceptually distinct units that you see on the screen, which are small enough to not hit asymptotic issues during interactive verification and during running automated verification.
My colleagues did most of this, and I helped them see how factoring and abstraction impact performance of running the automated verification

```
$ find src/Arithmetic -name "*.v" | xargs coqwc | sort -h
    spec    proof comments
      27       68        0 src/Arithmetic/CoreExtra.v
      29       75        1 src/Arithmetic/Primitives.v
      32       40        0 src/Arithmetic/SaturatedAssociational.v
      37       77        3 src/Arithmetic/FancyMontgomeryReduction.v
      39       39       36 src/Arithmetic/BarrettReduction/Wikipedia.v
      49       53       38 src/Arithmetic/BarrettReduction/HAC.v
      52       71        3 src/Arithmetic/Partition.v
      53       75        1 src/Arithmetic/ModularArithmeticPre.v
      58      133        2 src/Arithmetic/UniformWeight.v
```

```
   62    99    38 src/Arithmetic/BarrettReduction/Generalized.v
   69     0    91 src/Arithmetic/MontgomeryReduction/Definition.v
   70   163    37 src/Arithmetic/PrimeFieldTheorems.v
  102   152     4 src/Arithmetic/MontgomeryReduction/Proofs.v
  104   172    65 src/Arithmetic/BarrettReduction/RidiculousFish.v
  134   177    12 src/Arithmetic/ModularArithmeticTheorems.v
  150   109     9 src/Arithmetic/SaturatedColumns.v
  181   135    14 src/Arithmetic/Freeze.v
  181    31     3 src/Arithmetic/ModOps.v
  197   335    14 src/Arithmetic/BarrettReduction.v
  215   145    58 src/Arithmetic/CoreAssociational.v
  225   101    21 src/Arithmetic/BaseConversion.v
  266   198    14 src/Arithmetic/CorePositional.v
  357   503    50 src/Arithmetic/BYInv.v
  366   243    25 src/Arithmetic/SaturatedRows.v
  482   721    12 src/Arithmetic/WordByWordMontgomery.v
  512   407    72 src/Arithmetic/Core.v
  552   409    34 src/Arithmetic/Saturated.v
 4601  4731   657 total
```

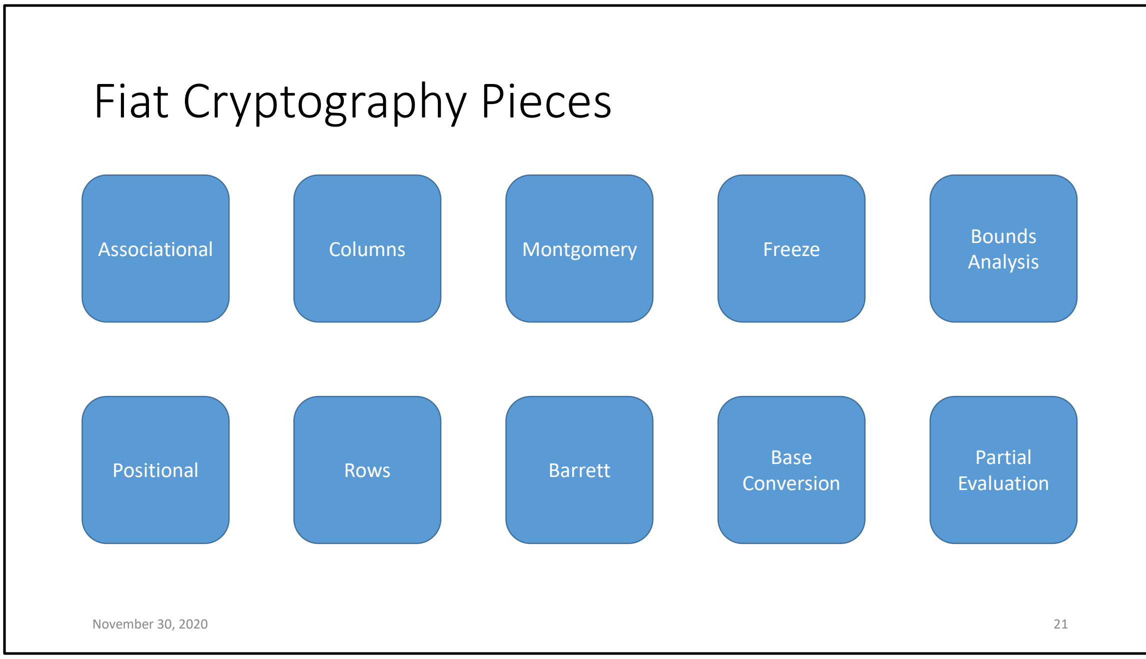
```
$ git ls-files "*.v" | grep -v Util | grep -v Demo | xargs coqwc | sort -h
  spec  proof comments
    5   152     3 src/Rewriter/Language/IdentifiersGenerateProofs.v
    9     0     0 src/Rewriter/Rewriter/Examples/PerfTesting/Settings.v
    9    10     0 src/Rewriter/Rewriter/Examples/PerfTesting/ListRectInstances.v
   16   109     0 src/Rewriter/Language/IdentifiersBasicLibrary.v
   18   705    39 src/Rewriter/Language/IdentifiersGenerate.v
   23     0     3 src/Rewriter/Language/PreCommon.v
   23   539    15 src/Rewriter/Rewriter/ProofsCommonTactics.v
   24  1524    12 src/Rewriter/Language/IdentifiersBasicGenerate.v
   44    33     0 src/Rewriter/Language/PreLemmas.v
   58    15     0 src/Rewriter/Rewriter/Examples/PrefixSums.v
   66     3     3 src/Rewriter/Rewriter/Examples.v
   73     0     6 src/Rewriter/Language/Pre.v
  138   191    41 src/Rewriter/Rewriter/Examples/PerfTesting/LiftLetsMap.v
  142    12    40 src/Rewriter/Rewriter/Examples/PerfTesting/UnderLetsPlus0.v
  171   208    14 src/Rewriter/Rewriter/AllTactics.v
  187   320    22 src/Rewriter/Language/IdentifiersLibraryProofs.v
  203     5    47
src/Rewriter/Rewriter/Examples/PerfTesting/SieveOfEratosthenes.v
  238     0     4 src/Rewriter/Language/UnderLets.v
  273    71     8 src/Rewriter/Rewriter/Examples/PerfTesting/Harness.v
```

```
     299       3     117 src/Rewriter/Rewriter/Examples/PerfTesting/Plus0Tree.v
     543     442      15 src/Rewriter/Rewriter/Wf.v
     582     275      26 src/Rewriter/Language/IdentifiersLibrary.v
     662     875       8 src/Rewriter/Rewriter/InterpProofs.v
     780     299       0 src/Rewriter/Language/Inversion.v
     900      17      54 src/Rewriter/Rewriter/Examples/PerfTesting/Sample.v
     962     946      22 src/Rewriter/Language/Wf.v
    1015     141      24 src/Rewriter/Rewriter/Reify.v
    1220     676       6 src/Rewriter/Language/UnderLetsProofs.v
    1310      95      68 src/Rewriter/Rewriter/Rewriter.v
    1630     138      96 src/Rewriter/Language/Language.v
    1813    1908      31 src/Rewriter/Rewriter/ProofsCommon.v
   13436    9712     724 total
$ git ls-files 'src/AbstractInterpretation/*.v' | xargs coqwc
    spec    proof comments
     519       0      23 src/AbstractInterpretation/AbstractInterpretation.v
     697     763       1 src/AbstractInterpretation/Proofs.v
     452     679       8 src/AbstractInterpretation/Wf.v
      24       0       0 src/AbstractInterpretation/WfExtra.v
     736      65      25 src/AbstractInterpretation/ZRange.v
     276     354       2 src/AbstractInterpretation/ZRangeProofs.v
    2704    1861      59 total
```

```
$ echo "defn$(printf '\t')$(coqwc src/Arithmetic/Core.v | head -1)"; for i in $(git ls-files 'src/Arithmetic/*.v'); do echo "$(cat $i | tr '\n' '~' | sed s'/`/,/g' | sed s'/\.[ ~]/`/g' | grep -o 'Definition [^`]*`' | tr '~' '\n' | wc -l)$(printf '\t')$(coqwc $i | tail -1)"; done | sort –h
```

```
$ echo "defn$(printf '\t')$(coqwc src/Arithmetic/Core.v | head -1)"; for i in $(git ls-files 'src/Arithmetic/*.v'); do echo "$(cat $i | tr '\n' '~' | sed s'/`/,/g' | sed s'/\.[ ~]/`/g' | grep -o 'Definition [^`]*`' | tr '~' '\n' | wc -l)$(printf '\t')$(coqwc $i | tail -1)"; done | sort -h
defn        spec    proof comments
0           102     152       4 src/Arithmetic/MontgomeryReduction/Proofs.v
0            39      39      36 src/Arithmetic/BarrettReduction/Wikipedia.v
0            62      99      38 src/Arithmetic/BarrettReduction/Generalized.v
2            52      71       3 src/Arithmetic/Partition.v
3           181      31       3 src/Arithmetic/ModOps.v
3            49      53      38 src/Arithmetic/BarrettReduction/HAC.v
3            58     133       2 src/Arithmetic/UniformWeight.v
```

4	134	177	12	src/Arithmetic/ModularArithmeticTheorems.v
7	29	75	1	src/Arithmetic/Primitives.v
7	37	77	3	src/Arithmetic/FancyMontgomeryReduction.v
8	70	163	37	src/Arithmetic/PrimeFieldTheorems.v
18	181	135	14	src/Arithmetic/Freeze.v
35	53	75	1	src/Arithmetic/ModularArithmeticPre.v
39	225	101	21	src/Arithmetic/BaseConversion.v
39	69	0	91	src/Arithmetic/MontgomeryReduction/Definition.v
46	104	172	65	src/Arithmetic/BarrettReduction/RidiculousFish.v
50	197	335	14	src/Arithmetic/BarrettReduction.v
63	482	721	12	src/Arithmetic/WordByWordMontgomery.v
66	357	503	50	src/Arithmetic/BYInv.v
123	552	409	34	src/Arithmetic/Saturated.v
153	512	407	72	src/Arithmetic/Core.v



TODO: scale by time taken, include estimates for rewriting-based
TODO: how to time-estimate bounds analysis?

When we're using the proof engine for partial evaluation, the time the proof engine takes to run the proof script just keeps growing and has unacceptable asymptotics

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

TODO: reword slide

Takeaways

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

TODO: do bold animation thing
Now we’re moving on to the main contribution of this talk, reflective partial evaluation

Partial Evaluation and Rewriting

November 30, 2020

26

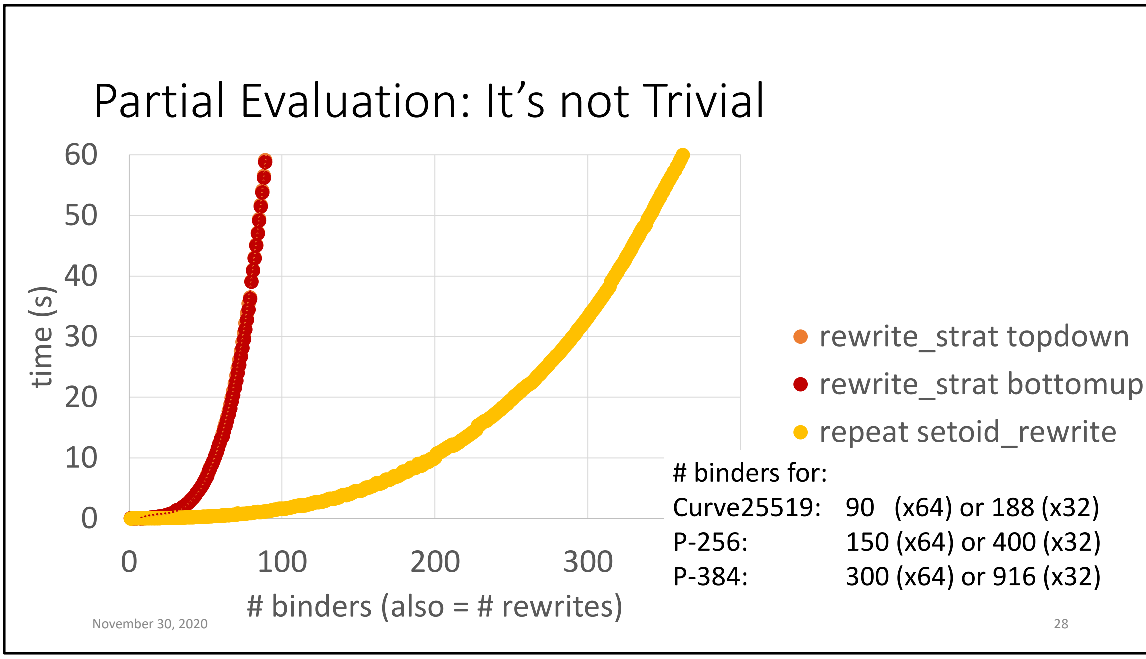
We saw before that this "partial evaluation" piece of fiat-crypto had inadequate asymptotics when done in the proof engine.
Let me tell you what this partial evaluation actually is.

Partial Evaluation: What is it?

- Describe it
- TODO: nounproject it

Partial evaluation is like: if you have instructions for making a peanut butter and jelly sandwich that start with “search the house for the peanut butter, the jelly and the bread”, and if you always keep them in the same place, rather than writing out instructions that start with “search the house”, you can instead write out instructions that start with “fetch the peanut butter from the top shelf of the pantry on the right”.

Notably, partial evaluation is partial; you can partially evaluation equations like $x + 2 + y - x + 6$ into $y + 8$ without knowing what y is.



Note that it's too big to handle interactively with reasonable performance (include perf plots of rewrite)
Note that this is also only about when arithmetic simplification is involved (or code-sharing-preservation)
TODO: note something about topdown and bottomup being almost identical
The underlying reason for this piece being hard is that all of the abstraction barriers that we introduced to carve the problem up into manageable pieces are broken here, so that we get fast low-level code out (this is a general pattern around performant code)
A large piece of my PhD work was making this possible in a way that scales

UnderLetsPlus0

Partial Evaluation and Rewriting: Requirements

- β-reduction
- ιδ-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, with is important for output code performance

Partial Evaluation and Rewriting: Requirements: ιδ-reduction + rewrites

- Example of ιδ-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

TODO: talk about and represent case analysis + recursion + function body inlining

Partial Evaluation and Rewriting: Requirements: Code Sharing Preservation

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

Partial Evaluation and Rewriting: Requirements

- 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

Note for ιδ-reduction + rewrites: 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: Obvious Extra Requirements

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

On performant: note that we don't quite manage this one, but we do a lot better than the interactive solutions

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 (ιδ+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 ⇝ Gallina application
- Expression abstraction ⇝ Gallina abstraction
- Expression constants ⇝ rewriter invocations on η-expanded forms
-

Partial Evaluation and Rewriting: Implementation: Normalization by Evaluation

Goal: Reuse substitution in Gallina for substitution in ASTs

TODO: fuse this with next two slides

Partial Evaluation and Rewriting: Implementation: 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"))

Say: everything not in quotes is Gallina, quoted things are AST in a deeply embedded language

Partial Evaluation and Rewriting: Implementation: 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 $\rightsquigarrow$ Gallina application

Expression abstraction $\rightsquigarrow$ Gallina abstraction

Expression constants $\rightsquigarrow$ rewriter invocations on η-expanded forms

In standard NbE, we just insert constant application at the leaves, rather than rewriter invocations. (Maybe emphasize this more)

Partial Evaluation and Rewriting: Implementation: 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!

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 ιδ+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)

Using extracted code: pattern-matching compilation gives approximately 4x speedup over naive strategy, + another 2x if we pre-reduce the rewriter (which OOMs if we don't use pattern matching compilation (quadratic code size in # of rewrite rules? (approximately due to encoding artifacts (ι-expansion of head symbol)))

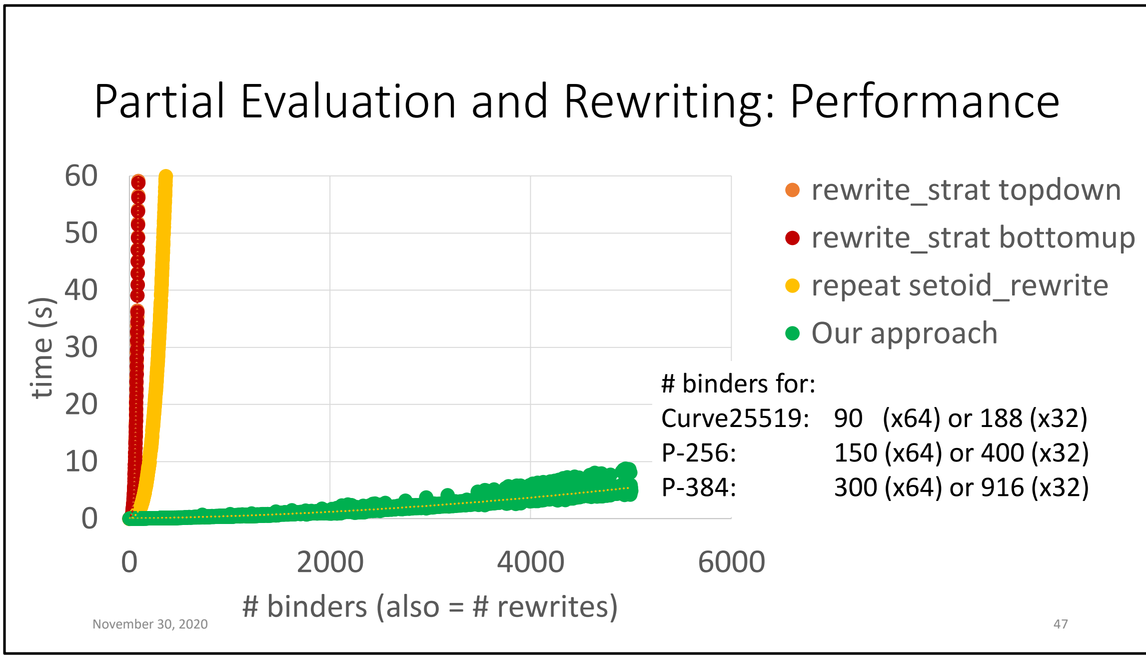
Partial Evaluation and Rewriting: Implementation

- 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 (ιδ+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
-

Note: speech for let-lifting: involves redoing NbE in a let-lifting monad (similar to the CPS monad). Only slightly tricky, but details are too complex to get into the nuance here

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.



Note that it's too big to handle interactively with reasonable performance (include perf plots of rewrite)
The underlying reason for this piece being hard is that all of the abstraction barriers that we introduced to carve the problem up into manageable pieces are broken here, so that we get fast low-level code out (this is a general pattern around performant code)
A large piece of my PhD work was making this possible in a way that scales

UnderLetsPlus0

Takeaways

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

Note that I'll be talking in the context of interactive dependently typed tactic driven proof assistants, because human ingenuity is important.
Come up with a better way to phrase this

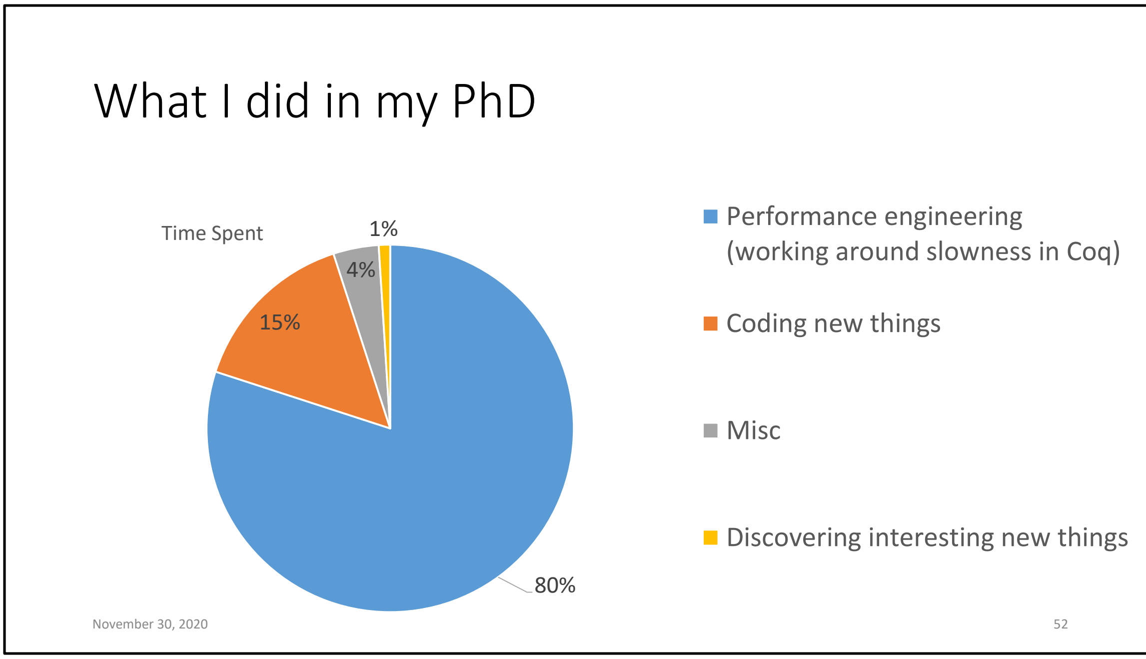
Takeaways

- 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!



TODO FIX SPEAKER NOTES
I want to start off by telling you how I spent my time in my PhD, because I think it is perhaps in some ways a little bit atypical.
PhDs are perhaps usually thought of as being about discovering interesting new things, but in fact only about 1% of my time could be identified as being aimed at that.
About 80% --- the bulk of my time --- was spent effectively working around slowness issues in Coq, aimed at doing performance engineering.
The reason I want to tell you this is that I'm going to be structuring my talk around performance engineering, both how it works currently and how I think it needs to change.

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 it's 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

November 30, 2020 60

Note that I'll be talking in the context of interactive dependently typed tactic driven proof assistants, because human ingenuity is important.
Come up with a better way to phrase this

Thank you for your time and attention!

Questions?