CS 839: Systems Verification
Lecture 1

Tej Chajed

Welcome to CS 839, Systems Verification!

My name is Tej. You can and should call me Tej. It rhymes with "page". You don't need to say "Professor".

Task: fill out your name tent

First name, in ALL CAPS
Pronouns, if you like
Write on both sides

- Write one name only, whatever you want people to call you
- Write pronouns to the side
- Write on both sides, for people next to you

5min

Introduce yourself to your "pod"

- Name, grad/undergrad, year
- **What's your go-to comfort food OR favorite pizza topping?**
- **What's the last programming project you worked on?**

10min

Who am I?

- started in 2023
- research is on verifying systems
- hobbies include cooking, cycling, reading, social deception games

my comfort food is aloo paratha, favorite pizza topping is probably ricotta cheese

What is this class about?

"program proofs" = code + specification + proof

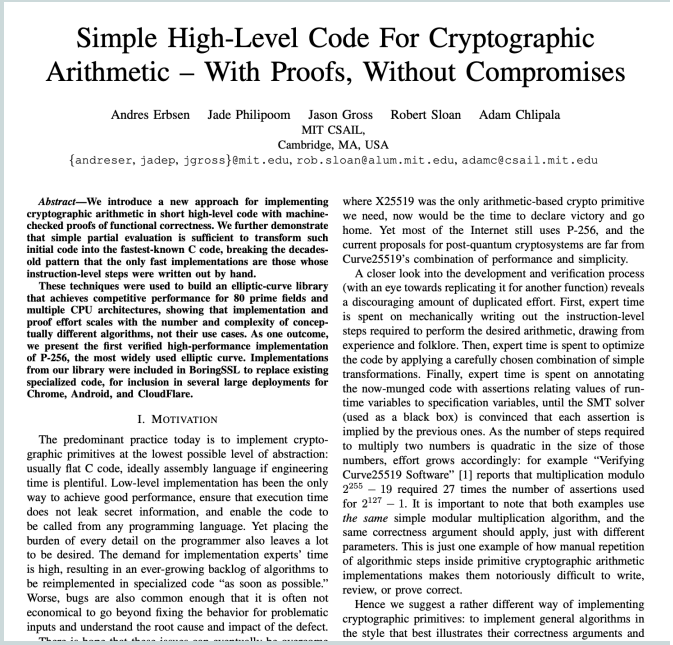
In this you'll learn to verify programs: prove they do what they're intended to do.

Most of this class is about learning how to state what a program does, and how to prove that a program does that thing.

Today we'll cover the broader motivation, what it means to do verification, and some details about how the class works.

Success story: verified cryptography

- security critical
- clear specification
- hard to test
- hard to fix bugs



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

Andres Erbsen Jade Philipoom Jason Gross Robert Sloan Adam Chlipala
MIT CSAIL,
Cambridge, MA, USA
{andreser, jadep, jgross}@mit.edu, rob.sloan@alum.mit.edu, adamc@csail.mit.edu

***Abstract*—We introduce a new approach for implementing cryptographic arithmetic in short high-level code with machine-checked proofs of functional correctness. We further demonstrate that simple partial evaluation is sufficient to transform such initial code into the fastest-known C code, breaking the decades-old pattern that the only fast implementations are those whose instruction-level steps were written out by hand.**

These techniques were used to build an elliptic-curve library that achieves competitive performance for 80 prime fields and multiple CPU architectures, showing that implementation and proof effort scales with the number and complexity of conceptually different algorithms, not their use cases. As one outcome, we present the first verified high-performance implementation of P-256, the most widely used elliptic curve. Implementations from our library were included in BoringSSL to replace existing specialized code, for inclusion in several large deployments for Chrome, Android, and CloudFlare.

I. Motivation

The predominant practice today is to implement cryptographic primitives at the lowest possible level of abstraction: usually flat C code, ideally assembly language if engineering time is plentiful. Low-level implementation has been the only way to achieve good performance, ensure that execution time does not leak secret information, and enable the code to be called from any programming language. Yet placing the burden of every detail on the programmer also leaves a lot to be desired. The demand for implementation experts' time is high, resulting in an ever-growing backlog of algorithms to be reimplemented in specialized code "as soon as possible." Worse, bugs are also common enough that it is often not economical to go beyond fixing the behavior for problematic inputs and understand the root cause and impact of the defect.

where X25519 was the only arithmetic-based crypto primitive we need, now would be the time to declare victory and go home. Yet most of the Internet still uses P-256, and the current proposals for post-quantum cryptosystems are far from Curve25519's combination of performance and simplicity.

A closer look into the development and verification process (with an eye towards replicating it for another function) reveals a discouraging amount of duplicated effort. First, expert time is spent on mechanically writing out the instruction-level steps required to perform the desired arithmetic, drawing from experience and folklore. Then, expert time is spent to optimize the code by applying a carefully chosen combination of simple transformations. Finally, expert time is spent on annotating the now-munged code with assertions relating values of run-time variables to specification variables, until the SMT solver (used as a black box) is convinced that each assertion is implied by the previous ones. As the number of steps required to multiply two numbers is quadratic in the size of those numbers, effort grows accordingly: for example "Verifying Curve25519 Software" [1] reports that multiplication modulo $2^{255} - 19$ required 27 times the number of assertions used for $2^{127} - 1$. It is important to note that both examples use *the same* simple modular multiplication algorithm, and the same correctness argument should apply, just with different parameters. This is just one example of how manual repetition of algorithmic steps inside primitive cryptographic arithmetic implementations makes them notoriously difficult to write, review, or prove correct.

Hence we suggest a rather different way of implementing cryptographic primitives: to implement general algorithms in the style that best illustrates their correctness arguments and

Formal methods have been used on software that is really deployed and used. Some high-profile examples include CompCert, a verified C compiler and seL4, a verified microkernel. The example I'll use is verified cryptography that was deployed in Chrome – a similar project was used for verified cryptography in Mozilla Firefox.

Success story: Fiat Crypto in Google Chrome

code: addition on numbers, six 48-bit representation

spec: $(a + b) \bmod 2^{255} - 19$

Cryptography turned out to be a good fit for verification. Modern encryption is based on something called an elliptic curve – all you need to know for the purpose of this explanation is that you have something that looks like a number and you need to implement operations like adding two numbers and scaling a number by a constant. However, there are several constraints: first, the implementation must be very fast. Second, for security the code must take the same time regardless of the input. Third, in order to get high performance, you have to write custom code for each architecture the code is deployed to.

The alternative to verification was extensive manual review by experts: once per combination of architecture and elliptic curve. This was both unscalable and error-prone. Testing is inadequate because the bugs are in corner cases; the behavior for one number isn't the same as another, and a motivated attacker *will* find the inputs that are buggy.

The specification here is extremely clear: the math that defines elliptic curve is a handful of equations, even though the code is very complex.

One note about this project: Google used Fiat Crypto in the Chrome browser, but they actually didn't deploy it on their servers because it still had a tiny performance cost. Two big differences: one, 1% more for encryption is a noticeable cost for Google on the server, and second, deploying bug fixes to the client is much harder (or impossible) compared to servers which can be updated in under an hour.

Systems we might want to be correct

- operating systems
- network stack
- cryptography
- distributed, replicated storage
- web browser

Why should we care about software correctness? Especially important for systems software, which supports many applications, and if there's a bug in the system it's likely to impact many applications and users.

In these cases, worth putting more effort into validating the software.

Big questions this class is about

1. How do we *know* a program is correct beyond testing?
2. What does it mean for a program to be "correct"?
3. How can we reason about concurrency and handle all possible executions?

"how do we know" is also "how do we convince others"

The main question of the class is the first one: how do we get mathematical certainty of a program's correctness, beyond the standard approach of testing it on a number of inputs. Embedded in this question is also a broad philosophical question about the nature of proof; while we won't explore this in depth, we will give highly rigorous proofs (machine-checked) that can be a standard for any other proofs you think about.

Example: hashmap

why is Load correct without acquiring the lock?

how do you know?

how do you convince someone?

```
type HashMap struct {
    clean *atomic.Pointer[map[K]V]
    mu    *sync.Mutex
}

func (h *HashMap) Load(key K) (V,
bool) {
    clean := h.clean.Load()
    value, ok := clean[key]
    return value, ok
}

// Clone the input map by copying all
values.
func mapClone(m map[K]V) map[K]V {
... }

func (h *HashMap) Store(key K, value
V) {
    h.mu.Lock()
    dirty := mapClone(h.clean.Load())
    dirty[key] = value
    h.clean.Store(dirty)
    h.mu.Unlock()
}
```

2min time to stare at the code and think about it

Active learning

You will be active in class. This will include thinking (without me talking), writing, talking to a partner or your pod, and sharing with the class.

Research shows that active learning improves learning outcomes, but it decreases the *perception* of learning.

Exercise: correctness of binary search

5min convince yourself

10min combine arguments with your pod

8min share-out

```
1 func BinarySearch(s []uint64,
2     needle uint64) (int, bool) {
3     var i, j = 0, len(s)
4     for i < j {
5         mid := (i + j) / 2
6         if s[mid] <= needle {
7             i = mid + 1
8         } else {
9             j = mid
10        }
11    }
12    return i, i < len(s) && s[i] == needle
13 }
```

Exercise

```
1 func BinarySearch(s []uint64,
2     needle uint64) (int, bool) {
3     var i, j = 0, len(s)
4     for i < j {
5         mid := (i + j) / 2
6         if s[mid] <= needle {
7             i = mid + 1
8         } else {
9             j = mid
10        }
11    }
12    return i, i < len(s) && s[i] == needle
13 }
```

- **5min** convince yourself
- **10min** combine arguments with your pod
- **8min** share-out

  23min total

Note to self: the code has some bugs. Line 6 should be `<` and not `<=` (easy to find with testing). Signed `(i + j) / 2` is incorrect if exceeding 2^{63}.

Precondition that list is sorted.

Why take this class?

- emerging field
- math + engineering
- verification mindset
- active learning?

Emerging field with exciting new developments.

Combination of math + engineering – that's what got me excited about it. You may find this fun. It's certainly a change from normal systems classes, it's not AI, and it's also different from theory in both the way we do the math and the applications.

Will influence how you think about any complex software: what is its specification? Can you even imagine a proof that it meets the spec? Can you try to test it?

If you want to do research in this area, this class is intended to be great preparation. Being able to use these powerful techniques lets you tackle almost anything, even if others can't.

Hopefully it's fun! Especially the active learning is intended to make the class engaging.

Future of verification

- Distributed storage
- Web browsers
- Operating systems (and hypervisors and containers)

This is an emerging field. Let me give a couple areas where I think verification could reach.

Amazon S3: has a simple specification, great internal complexity, not losing customer data is very high priority. Amazon is interested in and developing tools for proving parts of it.

Browsers: browsers are critical infrastructure that allow us to safely runs lots of untrusted code. There are a few scattered efforts to verify buggy and approachable parts of the browser, but a more comprehensive effort seems well deserved if it could be sustainable.

Operating systems (plus other isolation mechanisms like hypervisors and containers): when running systems software, we use isolation containers like the OS, hypervisors, and Docker/Kubernetes. These are also trusted and critical pieces of infrastructure. Similar to browsers, can we verify they provide the intended isolation?

Making reliable software

category	approaches
post-deployment	beta testing, bug reporting
social	code review, pair programming
methodological	testing, version control, style checking
technological	static analysis, fuzzing, property-based testing
mathematical	type systems, formal verification

Spectrum of approaches you might take to making software reliable.

Many of these are good software engineering practice but not routinely practiced or discussed in computer science curriculum.

Let me highlight a couple and what they're good at.

Post deployment: rare bugs that depend on configuration or user behavior might only be found in deployment. Automated bug reporting is excellent for finding and fixing crash bugs especially (or anything easily detected). Using beta testing and staged deployment mitigates the cost of such bugs to a smaller population. (See CrowdStrike which did *not* do this.)

Testing: testing is great! it's low effort and high reward

Formal verification: if you have a concurrent or distributed system that's responsible for storing user data, proving the correctness of your program on all inputs might actually be the most efficient way to get sufficient confidence.

Reliability requires a spectrum of approaches

- Less formal techniques are less expensive
- Even a formal argument has holes
  - did you prove the right thing?
  - do your assumptions match reality?

Not a tradeoff: should use multiple approaches.

Keep in mind the goal: how much effort you put into reliability and how you achieve it depends on your goals and what would go wrong if you had a bug.

If the consequences are minimal OR you could quickly respond with bug fixes (eg, website UI), might be worth relying on bug testing rather than thorough testing. If testing is too difficult BUT bugs are consequential, might have thorough code reviews. In a distributed systems, bugs can be hard to trigger in testing but have big impact in production; for a storage system the consequence might be lost data and thus worth putting large effort into preventing.

Verification

code + spec + proof

All verification involves code, a specification of the intended behavior of the code, and a proof that shows the code meets the spec.

The proofs in this class will be interesting compared to proofs you've done so far because we'll write them in a computer, and a program called an interactive theorem prover will check the proofs (and help us write it in the first place). This might seem unrelated to program verification, and in a way it is – we could instead prove the correctness of programs the same way we prove theorems in math, on paper. However, program verification deserves the higher assurance of machine-checked proofs because the proofs will have many more details than a typical mathematical proof, and it's easy to overlook a case or premise that needs to be proven. Since the entire goal was to have reliable software, we don't want to shift "is my program correct?" to "is my proof correct?". Furthermore, doing the entire thing in a computer will allow a much tighter connection to the code we run, so we make sure to capture all the behaviors of the code we wrote rather than something abstract on paper.

How this class works

1. Read lecture notes
2. Come to lecture, do in-class work, ask questions
3. Do the assignments
4. Use office hours and Piazza when stuck

You need to do the work to get anything out of this class. Do the reading so you're ready to participate in lecture. Come ready to work on exercises (sometimes on paper, sometimes in Rocq), with your peers. Do the assignments, and ask for help when stuck.

You need to put some time into the assignments to get anything out of this class. I'm giving flexibility in the assignments, but you need to work on them each week to finish them. The time is also there for you to ask when you get stuck.

The software we use is experimental and you won't find all the answers in the notes or online. I think this is a valuable experience, even though I also strive to make the notes as useful as possible.

Norms

- Be curious and present
- Create a safe space for being wrong
- Actively participate

Working agreements

- Use names
- Start and end on time
- Limit electronics

Ask questions in lecture. After today, lecture will be less structured, have a lot more demos. Time management is my job, not yours. I may take your question offline or defer the answer either if it's a longer discussion or I'm not sure about the answer.

Come prepared for class: read the lectures notes ahead of time as much as possible (I intend for them to be ready ~1 week in advance), and/or make progress on the assignment. Be present and ready to learn. Eat breakfast/lunch, sleep, do whatever you need to do to be awake and alert.

Who are you?

Fill out background survey on paper or Google Forms

8min

What's next

Lecture 2: Rocq Prover
Get started on Assignment 1 (at least setup)

Next Tuesday we'll get started with learning the Rocq Prover. I strongly encourage you to get a head start on assignment 1 – at least get the setup done.