1 General Information

Zero-knowledge proofs are a beautifully paradoxical cryptographic object: with them, you can convince someone of the truth of a statement *without revealing why* it is true. In recent years, zero-knowledge proofs have started to make the transition from theoretical cryptography research to practice. A decade of work from academics and practitioners has resulted in advanced zero-knowledge proof constructions—most often called zero-knowledge succinct noninteractive arguments of knowledge, or zkSNARKs—and built systems with which practical use cases can be addressed. Modern ZKPs can prove, with modest overheads, the correct execution of a rich class of computer programs, and even entire CPUs.

Today, though, doing this requires interacting with a quickly-evolving research area instantiated in complex toolchains that integrate a "frontend"—a compiler that turns a human-readable computation into an algebraic representation that our protocols can prove—with a "backend"—the zero-knowledge proof protocol itself. Today, these toolchains require a great deal of expertise to use well. A useful point of comparison is that ZKPs are today roughly where computer programming was in the 1950s and 60s, in the sense that using them well still requires breaking abstraction boundaries and understanding what's going on "under the hood". For ZKPs, this means understanding how the program translation happens, how the ZKP proves it for a specific input, and how the underlying mathematical computations work. Thus, despite their great promise, ZKPs have been slow to translate to applications due to the complexity of the tools.

This class is an accelerated one-semester introduction to modern zero-knowledge proofs. It is designed to give students a strong understanding of the ZKP pipeline described above. In addition to lectures, students will gain this understanding by completing a semester-long programming project: building a functional modern zero-knowledge proof software stack from scratch in Rust. Thus, though the class will touch on many deep ideas from the theory of zero-knowledge proofs and cryptography, it is not a traditional cryptography theory class: it will focus just as much on algorithmic, security, and systems questions as it does on formal proofs.

The "theory track". Because the underlying theory of ZKPs is quite deep and rewarding, students with a more purely mathematical bent may want to focus solely on that instead of on programming. To accommodate these students, in the first iteration of the class we will be giving students an optional "theory track" in which they work through a series of handwritten assignments on the course content. (Students interested in research on zero-knowledge proofs are strongly encouraged to do both the programming projects and the theoretical exercises.) However, *all* students must complete the first two programming projects.

Students will work in teams of at most three on the programming project, but must work alone on theory-track assignments if they choose to do them.

Setting expectations. This class is designed to cover a great deal of technical content in depth, but also quickly. **This will be a high-workload, difficult class.** Students should expect to struggle to understand the material on occasion, and to work on the projects for many hours. However, one of the most important takeaways of the course is that there is no magic in zkSNARKs—despite their reputation as a kind of "moon math", with time and patience, anyone can master them.

1.1 Learning Goals

The goal of this course is for students to gain experience with the following concrete technical skills:

- The internals of zero-knowledge proofs, including on-paper design and software implementation.
- Implementing mathematical computations like modular arithmetic, polynomials, and elliptic curves.

- Engineer a large-scale Rust codebase.
- Representing a computation in constraints.
- Measure, diagnose, and fix performance bottlenecks in ZKP software.

1.2 Email Contact and Times

Paul's email is paulgrub@umich.edu. Chad's email is cmlsharp@umich.edu.To contact the course staff via email, please include **[EECS498W26]** in the subject of the email. This makes it *vastly* easier to manage course emails, and increases the likelihood of receiving a timely reply!
Times for the course are as follows:
Lectures: Mon/Wed 10:30–noon, EECS 3427
Discussion: Fri 1:30–2:30pm, DOW 2150
Office Hours: Paul: Wednesdays 9-10:30am, BBB 4790
Chad: Thursdays 1pm-2pm, location TBD (see Piazza)

1.3 Materials

All online resources, lecture notes, homework uploads and downloads, Q&A, etc. can be found at the following locations:

- Lecture notes: `https://github.com/pag-crypto/EECS498598-W26-ZKP`
- Piazza: `https://piazza.com/class/mk2zfohuzae6ug/`
- Autograder: `https://autograder.io/web/course/365`

The following books are not required, but may be useful:

- Proofs, Arguments, and Zero Knowledge, by Justin Thaler.
- A Computational Introduction to Number Theory and Algebra, by Victor Shoup. Available here.
- A Graduate Course in Applied Cryptography, by Dan Boneh and Victor Shoup. Draft available at https://toc.cryptobook.us/
- Building Cryptographic Proofs from Hash Functions, by Alessandro Chiesa and Eylon Yogev. Available here.
- Foundations of Cryptography, Vol. 1 and 2 by Oded Goldreich.
- A Course in Cryptography, by Rafael Pass and abhi shelat. Freely available here

1.4 Prerequisites

The only formal prerequisite for this course is EECS 281 or an equivalent course. This course will require students to use a blend of technical skills. Students should be comfortable with reading and writing algorithms and formal definitions. Students should be able to read and understand proofs. The ability to write proofs will not be required to pass the course, but will be needed for the optional written assignments. Students should understand how to design software and work on large-scale codebases. No prior experience with Rust is required.

Helpful prior courses—none of which are formal prerequisites, but the more the better—include:

- EECS 376 (Foundations of Computer Science)
- EECS 475 and/or 575 (Introduction to Cryptography),
- EECS 388 (Introduction to Computer Security),
- EECS 483 (Compilers),
- Any Mathematics courses on discrete probability, algebra, or number theory.

2 Course Overview and Policies

This class's instruction will consist of lectures by Prof. Grubbs and discussion sections led by Chad.

Discussions. There will be one discussion section per week. The discussion will provide more background and cover some topics from class in more depth; they will also be the students' main source of training in the Rust programming language. Many discussions will be in the form of live-coding sessions that teach course topics and Rust in tandem.

This class will be taught in person. Students are expected to attend class and participate in person. In-class participation will be part of your final grade.

- Lectures and discussions will be recorded, and the recordings will be available on Canvas. The course staff will try to make the recordings available as quickly as possible. Class participation is required, and will factor into the final grade. The easiest way to fulfill the participation requirement is to come to class and ask questions, but there are other ways to participate—for example, asking and answering questions on Piazza, or coming to office hours and asking good questions. Please be courteous and abide by all campus policies regarding course recordings.[1]
- PDFs of slides, including the instructor's markups, will be made available on the course Github and on Canvas.
- The course's "main" office hours will be in person. For students who cannot attend, either due to illness or visa-related time zone difficulties, some additional office hours may be available by appointment.

[1] In particular: "Course lectures may be audio/video recorded and made available to other students in this course. As part of your participation in this course, you may be recorded. If you do not wish to be recorded, please contact the instructor the first week of class to discuss alternative arrangements," and "Students are prohibited from recording/distributing any class activity without written permission from the instructor, except as necessary as part of approved accommodations for students with disabilities. Any approved recordings may only be used for the student's own private use."

- If you feel sick, please wear a face mask to class. If you feel very sick, do not come to class.

These plans are subject to change on short notice.

2.1 Grading

Grades will be entirely determined by your performance on the projects and/or optional "theory track" assignments, and in-class participation. The programming projects will be graded for correctness, both via autograder and by inspecting your code manually. Manual inspection of your code is much easier if you write lots of good comments—even if your code is not fully functional, you can get partial credit if the comments help us understand what you were trying to do. The written assignments will be graded on *correctness*, *clarity*, and *conciseness*, and **must** be typeset in LaTeX (templates will be made available). It is good practice to start any longer solution with an informal (but accurate) "proof summary" that describes the core idea(s). This will help the reader—and you!—understand your solution better. The course will use standard thresholds for A/B/C/etc. grades.

Extra credit. There will be several ways to earn extra credit in the course. At the course staff's discretion, extra credit will be given for particularly clever optimizations or proofs, or for particularly efficient solutions to the projects. From time to time there will be optional questions on projects and homeworks; these are more for students who wish to learn more, and do not count for extra credit.

The lectures in this course are not yet typeset. Students who wish to earn extra credit can write LaTeX scribe notes for these lectures. More points will be awarded for higher-quality notes. Inquire with the instructor for more information.

2.2 Academic Honesty

For the projects, you may collaborate *in depth* only with your partners. You may discuss the assignment at a high level with students outside your group, but you should not share code or specific, low-level details of the solutions with them. External sources—especially the recommended readings and textbooks—are allowed on the projects, though you will learn more without them. You can use LLMs if you wish, but the course staff can practically guarantee they won't be very useful.

On written homework assignments, should you choose to do them, collaboration and consultation with external sources is allowed and encouraged, subject to the following conditions: first, you must first understand the problem on your own and make an initial reasonable attempt to solve it. Second, you must write your own solution, and list your collaborators/sources for each problem.

If you turn in code or a proof that you cannot explain orally, you will fail the assignment.

Students must also abide by the College of Engineering's honor code. There is no hard-and-fast list of (dis)honest conduct. When in doubt, err on the side of caution, or ask the instructor. Dealing with academic dishonesty is unpleasant for everyone involved, so please follow these policies!

The most important course policy. The final, and undoubtedly most important, course policy is: *you must treat the course staff, other students, and yourself with respect and compassion.*

3 Schedule

The course will be broken loosely into units, each covering a number of topics within a certain broad theme. The approximate plan is as follows.

- **Foundations:** Overview of course. Review of algebra (groups and fields), polynomial arithmetic, elliptic curves, and probability. Cryptography basics.
- **ZKP building blocks:** Rank-one constraint systems. Interactive oracle proofs. Polynomial commitments.
- **Getting to zkSNARKs:** Sigma protocols. The Fiat-Shamir transform. Special soundness. Optimizations.
- **ZKP programming:** Verifying a computation vs. executing it. Efficient R1CS representations. Using non-determinism. Accessing memory. Designing scalable systems.
- **Applications and advanced topics:** Lattice-based ZKPs. Folding schemes and incrementally verifiable computation. Zero-knowledge middleboxes. Anonymous credentials.

3.1 Project Schedule

The project is split into five milestones that build on each other. The current plan is as follows:

- **Milestone 1:** mathematical primitives. Implement a library for big integer arithmetic, modular arithmetic, elliptic curve group operations, and polynomials over finite fields. (three weeks)
- **Milestone 2:** the backend ZKP, part 1. Implement Delphian, a succinct interactive argument of knowledge for R1CS satisfiability. This will require two key subprotocols: an implementation of sumcheck and an elliptic-curve-based polynomial commitment. (two weeks)
- **Milestone 3:** the backend ZKP, part 2. Transform Delphian into Delphian-NIZK by implementing the Fiat-Shamir transform and hiding variants of sumcheck and the polynomial commitment. At the end of this milestone, students will have built a zkSNARK library that can prove computations represented as R1CS. (two weeks)
- **Milestone 4:** the frontend. Build a system that can take a program in a human-readable representation and transform it efficiently into an R1CS statement that is then proven by Delphian-NIZK. Programs will support arithmetic on field elements as well as "non-native" data types like 32-bit integers and bits; they will also support limited kinds of memory accesses. (two weeks)
- **Milestone 5:** an application. Students will use their frontends to implement a hash-preimage proof.

With the remaining time at the end of the semester, students will be strongly encouraged, but not required, to survey the literature to find improvements for some of the performance bottlenecks in the completed system, and solve them. For example, students could design more efficient gadgets for bitwise operations and investigate the performance improvement for different computations.