# Modern Zero-Knowledge

Fareed Sheriff

January 30, 2023

## Prerequisites

The course attempts to introduce material from the ground up, but there is some prereq material I'm not completely solid on, specifically, the number theory portion (and probably the JS portion as well, but I'll learn that soon as well). For number theory review, I use https://crypto.stanford.edu/pbc/notes/numbertheory/.

### Modular Arithmetic

Modular arithmetic is arithmetic done over the ring of integers $\mathbb{Z}_p$ for some number (and later prime) $p$. Addition, subtraction, and multiplication are done $\bmod p$. Division is done through multiplicative inverses where we want there to be one answer to $z = {}^{x}/_{y} \bmod p$ (we want the quotient to be unique). This is only possible if the denominator is coprime with $p$ because otherwise, we could a distinct $y' = y$ such that ${}^{x}/_{y} = {}^{x}/_{y'}$, but this means that $z$ is not unique. Therefore, we want the dividend to be coprime with $p$. For prime groups, this is not a problem because $p$ is prime, so $y$ is always coprime with $p$. Formally, for division over $y$ to be defined, we need there to exist a multiplicative inverse $y^{-1}$ of $y$ in $\mathbb{Z}_p$, which allows us to define ${}^{x}/_{y}$ as $xy^{-1}$. Formally, $y$ has a multiplicative inverse if there exists $w$ such that $yw = 1 \bmod p$. We note that $w = y^{-1}$ is unique if it exists because by contradiction $yw' = yw \implies w' = w$.

#### 1.1.1 Euclid's Algorithm

Euclid's algorithm allows us to find modular inverses, making division methodical. It just checks that two numbers are coprime by finding their gcd efficiently. If the gcd is 1, then a multiplicative inverse exists; otherwise, there is none. Euclid's algorithm for two integers $a, b$ (we assume $a > b$ wlog) by recursively applying the algorithm on $a \bmod b, b$ until we get either 0 or 1; when we get 0, the other number is the gcd of $a, b$ and when we get 1, 1 is the gcd of $a, b$. Alternatively, we recurse until we get 0; the other number is either 1 or a positive integer. The runtime of Euclid's algorithm is $\mathcal{O}(\log n)$ because every two steps, we shrink the max integer by at least a factor of 2. We can actually find $x, y$ such that $c = ax + by$ for arbitrary $c$ using the extended Euclidean algorithm for choice of $a, b, c$. Alongside Euclid's algorithm, we maintain $x_i, y_i$ with $x_0 = 1, y_0 = 0, x_1 = 0, y_1 = 1$, then update $x_i = x_{i-2} - q_{i-1}x_{i-1}, y_i = y_{i-2} - q_{i-1}y_{i-1}$ where $q_{i-1}$ is the bigger of the two numbers at step $i - 1$. This is equivalent to collecting all multiples of $a, b$ used to reduce to the gcd through the algorithm and storing them in $x, y$. At the end of the Euclidean algorithm, $x, y$ satisfy $1 = ax + by$, so for $b = p$, $x$ is the multiplicative inverse of $a$. Extended Euclid gives us a way to perform integer division in $\mathbb{Z}_p$ by calculating $y^{-1}$ and multiply $xy^{-1}$ to get $z = {}^{x}/_{y}$ where $x = yz = yxy^{-1}$.

#### 1.1.2 Chinese Remainder Theorem

The CRT forms a bijection between $\mathbb{Z}_n$ where $n$ is the product of prime powers (which is literally every number) and $(\mathbb{Z}_{p_1^{s_1}}, \cdots, \mathbb{Z}_{p_k^{s_k}})$ where $p_i$ is the $i$th prime in the decomposition of $n$ and $s_i$ is the power of this prime in the decomposition. We first show a bijection between $\mathbb{Z}_{pq}$ and $(\mathbb{Z}_p, \mathbb{Z}_q)$ for coprime $p, q$. We claim there is a unique solution $x \in \mathbb{Z}_{pq}$ for the system $x \equiv a \bmod p, x \equiv b \bmod q$. Clearly, the reverse direction must be true because for $x$, there is one $a \equiv x \bmod p, b \equiv x \bmod q$. We claim that for $y = aqq_1 + bpp_1 \bmod pq$ where $q_1 = q^{-1} \bmod p, p_1 = p^{-1} \bmod q$. Clearly, $y \equiv a \bmod p, y \equiv b \bmod q$ because the $qq_1, pp_1$ cancels mod the prime and either $a$ or $b$ is left over. We now show that there are no other solutions. If $y' \equiv a \bmod p$, then $y' - y \equiv 0 \bmod p$. Likewise, if $y' - y \equiv 0 \bmod q$. Therefore, $y' - y \equiv 0 \bmod pq$. As a result, there is a unique solution. We therefore have a bijection between $\mathbb{Z}_{pq}$ and $(\mathbb{Z}_p, \mathbb{Z}_q)$. That is, there is a

specific solution $(a, b) \in (\mathbb{Z}_p, \mathbb{Z}_q)$ associated with each $c \in \mathbb{Z}_{pq}$. Furthermore, for $c_1, c_2 \in \mathbb{Z}_{pq}$, we have $(a_1, b_1), (a_2, b_2) \in (\mathbb{Z}_p, \mathbb{Z}_q)$ and $c_1 + c_2 = (a_1 + a_2, b_1 + b_2)$, $c_1 c_2 = (a_1 a_2, b_1 b_2)$.

We now restate the CRT in its general form: for a system of multiple equations mod $m_1, \cdots, m_n$ for $n$ pairwise coprime integers with system $x \equiv a_1 \bmod m_1, \cdots, x \equiv a_n \bmod m_n$, there is a unique solution for $x \bmod M, M = \prod_i m_i$. We define $b_i = M m_i^{-1} (\bmod M)$ with $b_i' = b_i^{-1} \bmod m_i$. Then, we have a unique solution $x \equiv \sum_i a_i b_i b_i' \bmod M$ using the same logic. This forms a bijection between a product of primes and the Cartesian product of the prime factors. We can generalizw this to prime powers from primes and then to products of prime powers.

### 1.1.3 Units, Orders, & Primality Testing

We define a unit: a unit is an element with a multiplicative inverse in $\mathbb{Z}_n$. As noted before, $u$ has a multiplicative inverse in $\mathbb{Z}_n$ iff $\gcd(u, n) = 1$. We consider the group $\mathbb{Z}_n^*$; this group contains only elements coprime with $n$ and as a result, the order of the group is the number of elements coprime with $n$ including $n$. We define this number to be $\phi(n)$, Euler's totient. Note that $\phi(n)$ isn't necessarily even because it is possible for a number to be its own multiplicative inverse (like 2 mod 3). For prime $p$, $\phi(p) = p - 1$ because every element but $p$ is coprime with $p$. For powers of $p$, the totient is $p^k - p^{k-1}$ because there are $p^k$ elements and there are $p^{k-1}$ multiples of $p$ in $[p^k]$. We now consider prime product $pq$. For $x \in \mathbb{Z}_{mn}$ with $a = x \bmod p, b = x \bmod q$, we recall by CRT that $x$ is a unit in $\mathbb{Z}_{pq}$ iff $a, b$ are units. Therefore, $\mathbb{Z}_{pq}^*$ forms a bijection with $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$ and $\phi(pq) = \phi(p)\phi(q)$. Combining this with $\phi(p^k) = p^k - p^{k-1}$, we get $\phi(n)$ for $n = p_1^{k_1} \cdots p_n^{k_n}$ equals $(p_1^{k_1} - p_1^{k_1 - 1}) \cdots (p_n^{k_n} - p_n^{k_n - 1})$.

We now consider the discrete log problem; it is easy to exponentiate a number given base $a$ and power $b$ in field $\mathbb{Z}_n$ through binary exponentiation in $\mathcal{O}(\log b)$. However, it is computationally hard to find the power $b$ given $a$ and $c = a^b$ where $b \in \mathbb{Z}_n$. Euclid's algorithm allows us to find multiplicative inverses, but there do not exist polytime algorithms to find the discrete log. We note that we don't care about the behavior of nonunits under exponentiation because we can reduce this to the behavior of units: using CRT, each element $a \in \mathbb{Z}_n$ corresponds to some element $(a_1, \cdots, a_m)$ with $a$ nonunit if at least one of the $a_i$ is a multiple of $p_i$. This means that in at most $k_i$ exponentiation steps, the $i$th member reaches 0.

We know some things about powers of a unit $a \in \mathbb{Z}_n^*$: $a^k = 1$ for some $k$; furthermore, we denote the smallest $x \,\text{s.t.}\, a^x \equiv 1 \bmod n$ the order of $a$ in $\mathbb{Z}_n$. Fermat's little theorem says that $a^p = a \bmod p$ for any $a \in \mathbb{Z}_p$ for prime $p$. This is because $(x+y)^p \equiv x^p + y^p \bmod p$ since by binomial theorem, we multiply all but the first and last term by $p$, so only the first and last terms are left. Likewise, $(x_1 + \cdots + x_n)^p \equiv x_1^p + \cdots + x_n^p \bmod p$. As a result, for $a = \sum^a 1$, $a^p = (\sum^a 1)^p = \sum^a 1^p = \sum^a 1 = a \bmod p$. Euler's theorem is a special case of Lagrange's theorem for group orders: $a^{\phi(n)} \equiv 1 \bmod n$ ($a$ to the group order of $\mathbb{Z}_n$ equals the identity). This means that we can calculate $a^{-1}$ using binary exponentiation (also known as repeated squaring) to be $a^{\phi(n)-1}$. We now consider the RSA problem, which is similar to the discrete log problem: give $c = a^b, n, b$, we must find $a$. We can do this by finding $d = b^{-1}$, then raising $c^d$ to recover $a$ since $c^d = a^{bd} = a^{bb^{-1}} = a$. However, this requires knowing $\phi(n)$ and there is no known way to find this efficiently without knowing the factorization of $n$.

There are two well-known primality tests: the Fermat test and the Miller-Rabin test. If $n$ is prime, then for all $a \in \mathbb{Z}_n$, $a^{n-1} \equiv 1 \bmod n$. The Fermat test chooses a random $a \in [1, \cdots, n-1]$ and sees if $a^{n-1} \equiv 1 \bmod n$. If this does not hold, $n$ must be composite; however, it is possible for $n$ to be composite and still pass this test; in fact, it is possible for $n$ to be composite and pass this test $\forall a \in \mathbb{Z}_n$. Such numbers are known as Carmichael numbers. The Miller-Rabin test checks for primality by noting that $n$ is prime iff $x^2 \equiv 1 \bmod n$ is solved by $x = \pm 1$. If $a^{n-1} \equiv 1$, then we check that $a^{\frac{n-1}{2}} \equiv \pm 1$. We repeat this process by iteratively halving $n - 1$ until we get an odd number; if $n$ is prime, we will either get a sequence of 1s and -1s. The Miller-Rabin test chooses a random $a \in \mathbb{Z}_n$ and tries this process. For any composite $n$, the probability $n$ passes the test is at most $1/4$ though the actual probability is far less. Thus, rerunning the Miller-Rabin test yields an exponentially-low probability of nonprimality.

### 1.1.4 Generators & Cyclic Groups

A unit $g$ is called a generator or primitive root if for every $a \in \mathbb{Z}_n^*$, there is some $k$ such that $g^k \equiv a \bmod n$. We claim that $\mathbb{Z}_p^*$ contains $\phi(p-1)$ generators for prime $p$. In general for every divisor $d \mid p-1, \mathbb{Z}_p^*$ contains

$\phi(d)$ elements of order $d$.

When $\mathbb{Z}_n^*$ has a generator, we call it cyclic because it can be represented as $\langle g \rangle$ where $g$ is a generator that can generate the entire group. A subgroup is a nonempty set closed under the same operation as the original group with inverses as well. Also, every subgroup contains 1. We claim that any element in $\mathbb{Z}_n^*$ can be used to generate the cyclic subgroup $\langle a \rangle$. Any group is a subgroup of itself as is $\langle 1 \rangle$; these are both called improper subgroups: the first is not a strict subgroup and the last is a trivial subgroup. Lagrange's theorem says the order of any subgroup divides the order of its group; this is because we consider cosets $Ha$ for $H \leq \mathbb{Z}_n^*$, $a \in \mathbb{Z}_n^*$ and note that there are $\frac{\phi(n)}{|H|}$ such distinct cosets exist and together must partition the group. All subgroups of cyclic groups are cyclic. For cyclic group $G$ with order $n$, there are $\phi(n)$ generators because we need the generator under repeated exponentiation to reiterate over a previous value and this is only achieved when the order of a power $g^k$ of generator $g$ is coprime with the exponent $k$ since otherwise it wouldn't be a generator ro some subgroup. For any positive integer n, $\sum_{d|n} \phi(d)$.

We can form an isomorphism between cyclic groups with the same order. Furthermore, by CRT we can form an isomorphism between $\mathbb{Z}_n^*$ and the direct product of prime power groups and these prime power groups are all cyclic.

### 1.1.5 Quadratic Residues

We say $a \in \mathbb{Z}_n$ is a quadratic residue if there is some $x$ such that $x^2 \equiv a$; otherwise, $a$ is a quadratic nonresidue. Distinguishing a residue from a nonresidue mod $pq$ is an open problem. GM encryption uses this assumption. We assume $p$ is an odd prime. For generator $g \in \mathbb{Z}_p^*$, we recall that $a \in \mathbb{Z}_p^* \equiv g^k, k \in [0, p-2]$ since $g^{p-1} \equiv 1 \equiv g^0$ and we cycle again from $k > p-2$. For even $k = 2m$, $(g^m)^2 = a$ means $a$ is a quadratic residue; since $p$ is odd, exactly half of $[0, \cdots, p-2]$ is even. Therefore, at least half of the elements of $\mathbb{Z}_p^*$ are quadratic residues. Not only does $b^2 = a$ but so does $(-b)^2$. Therefore, every quadratic residue has two associated roots. This means at most half of the elements of $\mathbb{Z}_p^*$ are quadratic residues since otherwise there are more roots than elements. Therefore, exactly half of $\mathbb{Z}_p^*$ are quadratic residues because we also showed at least half of $\mathbb{Z}_p^*$ are residues. Furthermore, the residues are all even powers of $g$ (all even powers are residues and the number of even powers is half of $p-1$, so there are no other elements that can be residues). Exponentiating a residue by $\frac{p-1}{2}$ yields 1 because for $a = g^{2k}$, we get $a^{\frac{p-1}{2}} = g^{k(p-1)} = 1^k = 1$; exponentiating a nonresidue by the same results in -1 because we are left with $a^{\frac{p-1}{2}} = g^{k(p-1)+\frac{p-1}{2}} = g^{\frac{p-1}{2}}$. We know that the square of this is 1, so this equals either 1 or -1. However, $g$ has order $p-1$, so $g^{\frac{p-1}{2}} \neq 1$; therefore, it equals 1. This is Euler's criterion: $a$ is a residue iff $a^{\frac{p-1}{2}} = 1$ and it is a nonresidue if this is congruent to $-1$. We can represent this with the Legendre symbol for odd primes $p$ and integer $a$ as

$$\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}}$$

If $a$ is a residue mod $p$, $\left(\frac{a}{p}\right) = 1$; if $a$ is a nonresidue, $\left(\frac{a}{p}\right) = -1$. If $a \equiv 0 \bmod p$, then $\left(\frac{a}{p}\right) = 0$. For any integer $b$, $\left(\frac{a}{p}\right)\left(\frac{b}{p}\right) = \left(\frac{ab}{p}\right)$. For any $r$ coprime with $p$, $\left(\frac{ar^2}{p}\right) = \left(\frac{a}{p}\right)$.

## Miscellaneous Prerequisites

This includes security-related topics like Merkle trees and rainbow tables.

## Rainbow Tables & Salting

A rainbow table takes a table of hashed passwords and attempts to recover the passwords. The goal is to find some string that equals any of the hashes in a given hash table. Rainbow tables do this by maintaining hash chains, which are chains of hashes with a starting value that is repeatedly hashed to form the chain. Good hash functions do not produce cycles in the hash chain; creating the chain is easy, but recovering a value that hashes to a password partway through the chain is done by hashing the starting value repeatedly until we get to the element before the correct hash. We create hash chains (also known as key chains because they are different from true hash chains) by using a possible key derivation function and reduction function,

then repeatedly converting a value to a key, reducing to another value, and repeating. Rainbow tables consider many pairs of $r_i, k_i$ where $r_i$ is a reduction function and $k_i$ is a hash function. These functions are based on reasonable values for the associated hash function used to create the original hash table.

Rainbow tables are ineffective against hash functions that use salting. Salting involves generating some large random value $s$ that we call a salt; we add this value to the password before hashing it. This prevents rainbow attacks on hash tables because any initial value must be large and therefore, there is a high chance of the key chains in the rainbow table being distributed through the key universe but we know that the password-plus-salt combo must be large since the salt was large so we waste a lot more time and space computing irrelevant information, making rainbow tables no better than brute force attacks.

### Merkle Trees & Hash Lists

A hash list is just a list of hashes; for large files, it's sometimes difficult to hash the entire file, so we hash individual blocks of memory. Sometimes, we further hash to create a top hash, which hashes the hashes of the blocks of the file. The concept of hash lists is used in torrent files where you compare the hash list of the file to the provided hash list to verify its authenticity. Hash lists are useful for repairing blocks of memory since we don't need to redownload the entire file if part of the file is corrupted but rather just part of the file.

A Merkle tree or hash tree contains a bunch of hash lists. It's a binary tree formed on a hash list represented as an array; it's effectively a segment tree built on the hash list. Merkle trees act as commitment schemes where the root verifies a file's authenticity and the message being committed is the original file along with its hash list.

## Introduction to ZK

Zero-knowledge proves that a statement is in a language without revealing any other relevant information to the verifier. ZKPs are in many cases statistically significant evidence that the prover's proof or nonproof of a statement is correct and it is possible for a malicious prover to give a false proof of a statement, but the probability of this happening in a well-designed ZKP should be less than $1/2$. A nice 3-coloring proof on an arbitrary graph is to query any two adjacent vertices' colors and check that they are different; the soundness probability is like $3/4$ because for planar graphs, 4 colors is the maximum needed and we can make the graph almost 3-colorable and the completeness probabiity is 1 because the probability of a 3-colorable graph having a true proof is 1. The zero-Knowledge property of the proof system says that we actually learn nothing about the solution; revealing any individual pair of adjacent vertex colors is clearly zero-knowledge because the color choices are randomized except for the fact that they must be different. The computationally-bounded verifier could just as easily choose any two distinct colors for adjacent vertices and would therefore learn nothing extra from the prover's vertex color pair.

We now learn about zkSNARKs, the new big thing in applied cryptosystems. zkSNARK stands for zero-knowledge succinct non-interactive argument of knowledge. It is basically a non-interactive ZKP, but there is apparently a technical difference between proof and argument. A zkSNARK has a very short proof; it can be verified within a few milliseconds with a length of only a few hundred bytes for large statements. Zcash uses zkSNARKs to validate monetary transactions; the new version of Zcash uses the Orchard shilded payment protocol, which uses the Halo 2 ZKP. This system has no information exchange necessary for public encryption/exchange (public key exchange coupled with private key merging to form a verification string that ensures the two parties in the transaction are valid parties).

To have zk privacy in Zcash, the transaction validity function must answer whether the transaction is valid without revealing any other information. This is done in the Sprout and Sapling shielded pools by encoding the network's consensus rules into zkSNARKs. zkSNARKs turn what you want to prove into an algebraic equation and proves that they know the solution to this equation. The first step is to break down the logical steps of what we want to prove into their primitive operations (like addition and multiplication and stuff) to create arithmetic circuits. This is the computation/arithmetic circuit stage.

The next stage is the R1CS (rank 1 constraint system) stage. This checks that the values travel through the circuit correctly. We can verify this with a quadraatic arithmetic program (QAP), which converts the

constraints that need to be checked into a polynomial. Because when an identity doesn't hold between polynomials, it doesn't hold between most points, we only need to check that the output of two polynomials match at a randomly-chosen point and this happens with low probability if the two polynomials don't match. zkSNARKs used homomorphic encryption and elliptic curve pairing to blindly evaluate polynomials. To make this zero-knowledge, we add random shifts in the polynomial. We actually use zero knowledge proofs all the time, like with digital signatures.

zkSNARKs allow you to generate zkps for any kind of math proof. They are special in that we can create them in linear time and they can be independently verified by others in constant time without any interaction. This is nice for blockchains because it means people can create local proofs then upload them for constant-time verification.

Dark Forest is a game that uses zkSNARKs to obfuscate other player information. It ensures you need to spend resources to obtain other players' information. There is a "fog of war" that you can clear with resources. Players upload hashes of their coordinates to the Ethereum blockchain. Players also have to submit zk proofs that their moves are valid. SNARKs take in a function and output a zkp for the function. We write SNARK functions as circuits in the Circom language. The output of this program is a proving key and a verifying key. The proving key lets a prover computing the function represented by the circuit to generate a proof that they know the preimage of the function. The verifying key allows anyone to inspect the proof and tell that the prover actually carried out the computation. SNARKs can only use multiplication, addition, and subtraction. We use SnarkJS to generate proving and verification code in JS and Solidity.

Group signatures are signatures that prove somebody is a part of a group of people. ZKMessage was created as a proof-of-concept that zkSNARKs can be used for group signatures.

There are a host of articles I don't take notes on because they primarily discuss implementation details and the purpose of various new cryptocurrency-related anonymization protocols. SNARK-friendly hash functions are functions most of whose operations are arithmetic operations rather than bit operations because bit operations are costly to convert to arithmetic operations.

We look at how Dark Forest is implemented. The `init` circuit starts with players picking some $(x, y)$ pair for their location, computing the hash $h$ of this location, then send $h$ along with a zkp that they computed the hash from a valid location. A valid location is anywhere that is within a radius $r$ circle from the origin. This means the SNARK needs to prove both that $h(x, y) = h$ and $x^2 + y^2 \leq r^2$. The `init` Circom circuit is as follows

```
include "./mimcsponge.circom"
include "./comparators.circom"

template Main() {
    signal private input x;
    signal private input y;
    signal input r;

    signal output h;

    /* check x^2 + y^2 < r^2 */
    component comp = LessThan(64);
    signal xSq;
    signal ySq;
    signal rSq;
    xSq <== x * x;
    ySq <== y * y;
    rSq <== r * r;
    comp.in[0] <== xSq + ySq
    comp.in[1] <== rSq
    comp.out === 1;

    /* check MiMCSponge(x,y) = pub */
    component mimc = MiMCSponge(2, 220, 1);
```

```
25
26      mimc.ins[0] <== x;
27      mimc.ins[1] <== y;
28      mimc.k <== 0;
29
30      h <== mimc.outs[0];
31  }
```

The first import imports the MiMC hash and associated functions; the second imports comparators. The private inputs are $x, y$ and the public input is radius $r$. The output is the hash. The code ensures that the parameters are satisfied. A number of articles/tasks require I install and play around with code from GitHub repos but this code is in JS and I feel that playing around with code in a language I do not understand does not teach me anything. The powers of tau ceremony is an event in which everyone samples randomness and make it part of something like a Zcash parameter. This makes it hard for attackers to decrypt the randomness because to do so would require getting the randomness from everyone involved. An air-gapped machine is one that is physically disconnected from all networks. The perpetual powers of tau ceremony is one that is always ongoing.

The Fiat-Shamir heuristic turns an interactive zkp into a non-interactive one by simulating the verifier. This is done by replacing the verifier's randomness with the output of a hash function whose inputs are the parameters of the proof as well as the randomness used by the prover. The final verification step done by the verifier in the interactive zkp can then be done by anyone after the prover calculates everything. It is necessary that a fixed random generator can be constructed through only the data known to both parties for this to happen.

We formally discuss what a zkp is. We define $\Sigma = \{0,1\}$ with language $L$. $L \in \mathcal{NP}$ means there exists a deterministic algorithm $V : \Sigma^* \times \Sigma^* \to \{0,1\}$ and $x \in L$ means there exists a $P \in \Sigma^*$ such that $V(x,P) = 1$ with $|P|$ polynomial in the length of $x$ and $P$ is a short proof that $x \in L$. A zkp proof system for $L$ is a pair $(P,V)$ that is complete, sound, and zero-knowledge. A complete system is one for which for all $x \in L$, $V$ says yes after interacting with the prover. A sound system is one for which for all $x \notin L$ and all provers $P^*$, a verifier says no after interacting with $P^*$ with probability at least $1/2$. A perfect system is one such that for all verifiers $V^*$ there exists a randomized p. p. t. simulator $S^*$ such that for all $x \in L$, $\{\mathsf{transcript}((P,V^*)(x))\} = \{S^*(x)\}$ where $S^*$ simulates $V^*$'s responses to $P$. Note that the perfect property is equivalent to the zero-knowledge property because if you can simulate $V^*$, then $V^*$ doesn't learn anything it couldn't figure out itself already — $V^*$ cannot learn anything other than the fact that $x \in L$.

A zkp for DDH (the decisional Diffie-Hellman assumption) $\langle g, g^a, g^b, g^{ab}\rangle$ where $g \in G$ is of order $q$. The Chaum-Pederson zk protocol denotes $\langle g, A, B, C\rangle = \langle g, g^a, g^b, g^{ab}\rangle$. $V$ first draws $s \leftarrow \mathbb{Z}_q$ and sends $commit(s)$. $P$ draws $r \leftarrow \mathbb{Z}_q$, then sends $y_1 = g^r, y_2 = B^r$. $V$ then sends $s$ to open the commitment. $P$ sends $z = r + as \bmod q$. Finally, $V$ checks that $g^z = A^s y_1$ and $B^z = C^s y_2$. The proof of zzk can be done by defining a simulator $S^*$ such that $S^*$ first picks a random $z \leftarrow \mathbb{Z}_q$, then runs $V^*$ to get $commit(s)$, sends random $y_1, y_2$, $V^*$ sends $s$ to open the commitment, then sets $y_1' = {}^{g^z}/_{A^s}, y_2' = {}^{B^z}/_{C^s}$ and outputs $(commit(s), y_1', y_2', s, x)$.

We again review zkSNARKs. The first main step in getting a zkSNARK is converting the problem to a quadratic arithmetic program (QAP). This step is often very difficult. We use as example proving that we know the solution to cubi $x^3 + x + 5 = 35$. We start with our function

```
1  def qeval(x):
2      y = x**3
3      return x+y+5
```

The first step is a flattening process that converts the above function into a function composed by only additions, subtractions and multiplications. This yields

```
1  def qeval_(x):
2      xx = x*x
3      xxx = xx*x
4      xxxpx = xxx+x
5      ~out = xxxpx+5
```

where $\sim$`out` is the output. We now convert this to a r1cs (rank-1 constraint system). An r1cs is a sequence of three vectors $(a, b, c)$ and the solution $s$ to an r1cs is a vector that must satisfy $(s \cdot a)(s \cdot b) - s \cdot c = 0$. Rather than having just one constraint, we have multiple constraints one for each logic gate. We define variable mapping $\langle x, xx, xxx, xxxpx, 1, \sim out \rangle$. We represent $(a, b, c)$ as a vector triple for the first gate $(xx = x * x)$ as $\langle 1,0,0,0,0,0 \rangle, \langle 1,0,0,0,0,0 \rangle, \langle 0,1,0,0,0,0 \rangle$. The second gate $(xxx = xx * x)$ is $\langle 1,0,0,0,0,0 \rangle, \langle 0,1,0,0,0,0 \rangle, \langle 0,0,1,0,0,0 \rangle$. The third gate $(xxxpx = xxx + x)$ is done as follows: $\langle 0,1,1,0,0,0 \rangle, \langle 0,0,0,0,1,0 \rangle, \langle 0,0,0,0,1,0,0 \rangle$. The final gate $(\sim out = xxxpx+5)$ is $\langle 0,0,0,1,1,0 \rangle, \langle 0,0,0,0,1,0 \rangle, \langle 0,0,0,0,0,$ Each gate is $a \cdot b - c = 0 \rightarrow a \cdot b = c \leftrightarrow c = a \cdot b$. This yields our r1cs with four constraints. The witness is the assignment to all variables. We know that $x = 3$ is a solution to $x^3 + x + 5 = 35$, so our witness is the assignment to all variables including input, output, and interval variables $\langle 3, 9, 27, 30, 1, 35 \rangle$. We stack all the constraints together to get our complete r1cs. We convert this to a qap, which uses the same logic but represents stuff with polynomials instead of inner products. This transformation involves Lagrange interpolation.

Polynomials are cool because they are a single object that can contain an unbounded amount of information. Similarly, a single equation between polynomials can represent an unbounded number of equations between numbers because if $A(x) + B(x) = C(x)$, it is true that $A(x') + B(x') = C(x') \forall x'$. This is where Lagrange interpolation comes in. If we want to check a system of equations where the lhs is the sum of two numbers and the rhs is a single number, we can use Lagrange interpolation to construct a polynomial $A(x)$ that represents the first set of numbers on the lhs, $B(x)$ for the second set of numbers on the lhs, and $C(x)$ for the rhs set of numbers so that $A(x)+B(x) = C(x)$. The factor theorem for polynomial long division says that we can express any polynomial as $P(x) = Z(x)H(x)$ where $Z(x) = (x - x_1)(x - x_2) \cdots (x - x_n)$ where $P(x_i) = 0$ and $H(x)$ is a polynomial — we can factor $P(x)$ into a polynomial with 0s at the same place as $P(x)$ and some other arbitrary polynomial.

We now discuss how to verify equations with polynomials faster than brute-force checking each coefficient through polynomial commitment schemes. A polynomial commitment is basically a special way to hash a polynomial where the hash has the property that you can check equations between polynomials by checking equations between their hashes. These polynomial commitments are fully homomorphic, meaning we can add and multiply commitments to verify multiplication and addition of polynomials. Furtheromre, given a commitment, $x, y$ and a supplemental proof/witness, we can verify that $P(x) = y$. According to the Schwartz-Zippel lemma, if some equation on polynomials holds true at a random coordinate, then it holds true with high probability at all coordinates. The three most-used polynomial commitment schemes are Kate, bulletproofs, and FRIs.

PLONK (Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge) is a new general-purpose zkp scheme. PLONK includes enhancemens that improve usability of these types of proofs. While PLONK still requires a trusted setup procedure, it is universal and updateable, so you just need one trusted setup for the entire scheme rather than one separate trusted setup for each program you want to prove and it is possible for multiple parties to participate in the setup as long as one of the members is honest. All participants don't even need to be known ahead of time. It also only relies on polynomial commitments, specifically Kate commitments. Where SNARKs use QAPs, PLONKs convert a problem of form "give me $x$ for $P$ such that $P(x)$ yields $Y$" again into a set of problems that are later converted into a polynomial using a similar technique to QAPs.

There is a textbook on elliptic curve pairings that starts with elliptic curves as groups and goes all the way to pairings. I will review it over time. I watch every lecture in the lectures section, starting with the introduction section.

## 2022 Applied ZK Learning Group Lectures

### 2.1.1 Introduction to ZK

The development environment used is node.js version 14 or higher. I use node.js 19. Most materials in the lectures use circom 1.0, but circom 2.0 was recently released and to use it, I need Rust available.

We start with an overview of Ethereum. Ethereum is a computer, the Ethereum Virtual Machine or EVM. The cryptocurrency is called Ether. The network of people using Ether is connected to the EVM. Each node contains a copy of the EVM state and when a transaction occurs, the state at the current node

gets updated, pushed to the EVM, then propagated to everyone else. The amount of ether paid to do some computation is the function of the length of the computation. This means that if somebody wants to execute a long script, they will have to pay a lot of money. Rather than requesting a computation by writing new code everytime, developers upload programs called smart contracts to EVM storage and will execute when certain conditions are filled for a fee. A dapp (decentralized app) run on the Ethereum blockchain (the transcript of Ethereum's computations) has zero downtime in that the conditions for a script to execute force the script to execute immediately once it's incorporated into the blockchain.

The applied zk learning group is an experimental educational program for engineers, academics, and people who are interested in applied zk cryptography to build applications. We cover conceptual introduction to zkps and zkSNARKs and examples of applied zkSNARKs. The focus of this learning group is applied zk. We mostly use the circum + snarkjs stack. zk theory is not the focus of this class and notably a lot of lecturers are not experts on zk theory. circom + snarkjs is built by an organization called iden3. The first two weeks are structured workshops and the last two are hands-on building a project and guest talks from zk ecosystem speakers. Everyone has to build something. There are approximately four lectures each week. No question is too basic and people come from a wide range of backgrounds. zk crypto is more important and general than people think it is and it is easier to use than people realize it is. we focus on nizkps (noninteractive zero-knowledge protocol). We consider an example for map 3-coloring (this ensures our soundness error is manageable by 4-color theorem on planar graphs). The goal is to convince somebody you have a valid 3-coloring without revealing any information. You then reveal two neighboring regions and show that the colors are different. The completeness error is 0 and the soundness error is like $1/3$. Another example is digital signatures. The holy grail of zkps is getting some arbitrary function $f$ and output $y$ and proving you know $x$ such that $f(x) = y$.

We now discuss homomorphic encryption. Many servies store encrypted versins of your data. The problem is that you can't do computations on encrypted data because it's effectively random. Homomorphic encryption allows you to solve this problem because the encryption of a function is the function of the encryption. Homomorphic encryption is like zk for arbitrary functions. One application of zkSNARKs is dark forest, which is a game that uses zkps to implement a 'fog of war'. Loopring is a zk rollup application. It allows you to verify something very quickly by generating a zkp for a linear time verification (for example) such that this zkp is constant-time, so you can just roll up the computation required to verify. zk rollups are one of the big long-term expected performance gains in Eth2 (new Ethereum standard).

### 2.1.2 Circom Workshop 1

We have zkSNARK with inputs $x_1, x_2, x_3, x_4$, intermediate values $y_1 = x_1 + x_2, y_2 = y_1 * x + 3$, and output value $out = y_2 - x_4$. The prover is producing a proof that some constraints are satisfied on these seven values. The SNARK prover has 7 inputs and will generate a signature saying that $y_1 == x_1 + x_2, y_2 == y_1 + x_3, y_2 == out + x_4$. circom offers some syntactic sugar. Note that *out* is a specific value that we need the inputs to satisfy. Not all computations can be broken into plus and times. Luckily, the constraint system is the only system that cares about this; that is, verification is the only system limited to using only multiplication and addition. circom uses a specific prime called the baby-jubjub-prime that is 254 bits as its basis for modular arithmetic; this prime is the standard prime used in the EVM and Ethereum blockchain because it's associated with a set of standard elliptic curves. The prime has to satisfy some elliptic curve properties.

To make a dapp, you have zkSNARK circuit code that goes into a circom compiler to generate a proving key passed to the client and a verification key passed to the verifier (like a server verifier in nodejs). The client generates a proof on specific inputs and passes it to the server/verifier. A proving key has .zkey file extension and a verifier key has a .vkey.json or .vkey file extension. circom is short for circuit compiler. snarkjs is a library that exposes functions whose type signatures with inputs to keys and output is a bunch of elliptic curve points or something.

We now experiment with the GitHub repo https://github.com/0xPARC/circom-starter, which contains an example of how to use circom and snarkjs together. The number of constraints does not affect gas usage because verification is constant. The hash.circom file contains a sample hash function; this is clearly not a very good hash function. We declare variables using `signal` and we can define them further as `private` or `output` types where if a variable is not explicitly declared `private`, it is assumed to be `private`, and

`output` signals are public. We assign with `<-` and constrain with `===`. The hardhat-circom plugin allows us to actually use this circom file through running `yarn circom:dev`. This generates a set of proving and verifying keys from `hash.circom`, then generates a zk proof for the inputs in `hash.json`. Instead of doing a single arrow `<-`, you can do a double arrow `<==` and this both assigns a variable and creates a constraint with the same format.

The snarkjs repo has a 26-step tutorial to describe all the steps required to create and verify a zkSNARK. hardhat-circom is made to make some of the steps in the tutorial run as single commands. It is a zkSNARK devtool. Under the hood, + does not count as a constraint. When we run the compilation process, we generate a lot of other files. circom will also generate some Solidity code. The input to the prover is the computation trace, known as the witness, and is in the .wtns file along with the proving key .zkey. The .ptau file is basically entropy to create the proving and verification keys. public.json is a list of public output signals or something. iden3 is behind circom.

We now look at the circom documentation. circom is a compiler that compiles circuits and outputs the circuit as a set of constraints as well as stuff needed to compute different zk proofs. circomlib is a library of circom templates with lots of circuits like comparators, hash functions, signatures, and stuff like that. circomlibjs is a JS library that makes programs to compute the witness of different circomlib circuits. circomtester is an npm package that gives you tools to test circom circuits. snarkjs is an npm package that takes the output of a circom-compiled circuit and creates files that allow you to generate and validate zk proofs. Basically, we first design our circuit using circom, compile the circuit to get an r1cs using `circom circuit.circom -r1cs -wasm -sym`, use snarkjs to compute the witness with `snarkjs calculatewitness -wasm circuit.wasm -input input.json -witness witness.json`, generate a trusted setup to get our zkSNARK proof with `snarkjs setup` `snarkjs proof`, and finally validate or have a smart contract validate our proof with either `snarkjs validate` or `snarkjs generateverifier`. Basically, we go from a high-level circuit description through the circom compiler to a low-level circuit description, through the snarkjs compiler to a proving-validation key pair and either directly create a proof or create a verifier code that Solidity can use in a smart contract. We can simplify many of these steps with the hardhat package that streamlines crypto projects.

For circom files, we start with a `template Main()` declaration after giving the compiler the version directive and importing what we want. Then, we list the signals, list the computations, and finally list the constraints. `yarn circom:dev` actually combines all the snarkjs and circom commands together.

### 2.1.3 circom documentation

We walk through the entire circom documentation.

We first start with installing dependencies, then installing the circom library and associated packages. To use circom, we need to install Rust. We can do this with yay or with the command `curl -proto '=https' -tlsv1.2 https://sh.rustup.rs -sSf | sh`. We also need node.js to be version 10 or higher. To install circom from source, we clone the circom directory at `iden3/circom.git`, then use cargo build to compile with `cargo build -release` once we're in the `circom` directory. We then install the binary with `cargo install -path circom`. Finally, we install snarkjs with `npm intsall -g snarkjs`.

We now discuss writing circuits. The following is an example of a circom circuit.

```
pragma circom 2.0.0;

/* description */

template Main(){
    //signals
    signal input x;
    signal input y;
    signal xy;
    signal output z;

    //assignments
    xy <-- x*y;
```

```
    z <-- xy+y;

    //constraints
    xy === x*y;
    z === xy+y;
}

component main = Main();
```

pragma is used to specify the compiler version. template is used to define the circuit. Not only do we have <- or <== but we also have -> and ==>. Finally, we create an instance of the template with component main = Main();. We can compile this circuit with circom test.circom -r1cs -wasm - sym -c. This generates the r1cs file, the WebAssembly file used to generate the witness, the symbols file sym used to get debug information about the circuit, and a C++ directory that contains files needed to compile C code to generate the witness.

We compute the witness with the wasm module (we don't actually need the generated c code). Using the wasm and a file with the inputs, we can create a witness that is effectively a computation trace of the circuit — it contains the values of the inputs, intermediates, and outputs. We create a json file to make this witness named input.json. This contains a bracketed list of input signal-value pairs as follows:

```
{
"x": "3",
"y": "7"
}
```

We add quotes around the numbers because if the number is greater than $2^{53}$, it doesn't automatically convert to a BigInt. We can use node to compute the witness by putting input.json in the [project name]_js folder, cding to the folder, then running node generate_witness.js test.wasm, input.json, witness.wtns. We can do the same thing in C++ by running make in the corresponding cpp directory; however, we need to have installed nlohmann-json3-dev, libgmp-dev, and nasm. This generates a witness.wtns file. For big circuits, the C++ witness calculator is a lot faster than the wasm calculator.

We now use the wtns file and the r1cs file to create our proof with snarkjs. We can also generate a Solidity verifier that allows you to verify proofs on the Ethereum blockchain by generating the Solidity code with snarkjs. signals are immutable. We can use previously-defined circuits as follows:

```
1  pragma circom 2.0.0;
2
3  template A(){
4      signal input in;
5      signal output outA;
6      outA <== in;
7  }
8
9  template B(){
10     signal input x;
11     signal output out;
12     component comp = A();
13     comp.in <== x;
14     out <== comp.outA;
15 }
16
17 component main = B();
```

Operators in circom include the ternary operator, boolean operators (and, or, negation), relational operators (<, <=, ==, !=, etc.), arithmetic operators (+, -, *, **, / [multiplication by modular inverse], \ [division], % [modulus]), and bitwise operators (>>, &, ~ [complement], ^ [xor], >>, <<). An example is as follows:

```
pragma circom 2.0.0;

template Example(n){
    signal input in;
    signal output out[n];
    var a = 0;
    var b = 1;
    for(var i = 0; i < n; i++){
        out[i] <-- (in >> i) & 1;
        out[i] * out[0] === 0;
    }
    a === in;
}

component main {public[in]} = Example(5);
```

We basically have JS-like control flow and variable declaration (though we should have `let` instead of `var`). `var`s are variables and are mutable. This differs from JS in that assignments do not return anything and so we cannot do multiple assignments in an expression. Templates let us create generic circuits in circom. We pass parameters into templates independent of signals. We pass parameters by just passing in the name of the parameter. When calling a component we pass in its input signals with the syntax `{public [input1, input2, etc.]}`. If a signal is noted used in any constraints, a warning will be generated. To access input or output signals of a component, we use dot notation (because the other signals are private, we cannot access them from outside the component). Components are also immutable. Starting at version 2.0.6 of circom, we can also define custom templates by importing with `pragma custom_templates;`. We declare a custom template with the `custom` keyword directly after `template`. The computations are encoded differently for custom templates. We can also do assertions with `assert(boolean expression);` and this is checked at execution time. We can debug with `log(statement);`. The type datatypes in `circom` are field element values and arrays (delcared with `[]`). We cannot have matrices directly, but we can have arrays of arrays. We also explicitly give the size of an array when instantiating it.

The `pragma` keyword is used to define the circom version and for custom templates; it is not used for anything else. To import other files, we use `include "filename.circom";`. This lets us use the templates and functions from the imported files. Functions are defined the same way as templates but with `function name(param1,param2){}` instead of `template`. They must return something and they can be recursive.

Finally, the `main` component is the initial component needed to create a circuit and defines the global input and output signals of a circuit; only the `main` component has the public signal list.

There are 19 reserved keywords in circom: `signal`, `input/output`, `public`, `template`, `function`. `component`, `var`, `return`, `if/else/for/while/do`, `log`, `assert`, and `include/pragma circom/pragma custom_templates`. Also, we can do multiple operations on the same line as log as it is not also a constraint. An example of logging is `log("something", varName, "something else");`. Constant values and template paramaters are considered knowns and signals are always unknowns. Arrays and conditionals must have known conditions. circomlib contains a library of circom templates.

### 2.1.4 Circom Workshop 2

We look at existing circuits. We look at `mimcsponge.circom`. We then look at the MiMC hash. We first look at a generic construction for group signatures using just hashes. The input is a bunch of public group member hashes, a public message, and some secret. One of the intermediate values is $x$, the hash of the secret. The output is the message attestation. The idea is anyone in the group knows the secret and can use it to hash the group hashes and the secret hash to get the attestation. We want to prove that `mimc(secret) = x`, `(x-hash1)(x-hash2)(x-hash3) = 0`, `msgAttestation = mimc(msg, secret)` (the MiMC of the secret is $x$, $x$ equals one of the hashes and therefore the secret key of the group member that signed equals one of the secret keys, and the attestation output equals the MiMC of the message and the secret).

In the `IsZero` circuit, where the output is 1 if the input is 0 and the output is 0 otherwise, we do not simply check that the input equals 0 because we cannot encode this as a constraint using addition and

multiplication. Not a lot conceptually was covered in this lecture; the brunt of the presentation discussed the `IsZero` circuit.

### 2.1.5 Overview of ZK Dapps

Dapps are decentralized apps. We first get an introduction to Ethereum, when SNARKs are helpful when building on top of Ethereum, and examples.

Ethereum is a computer. It's a worldwide computer (like a world Docker container) that everyone keeps an exact copy of and that anyone can run stuff on. Bitcoin is like this but it has a ledger (slab of memory) instead of a Turing-complete VM. This is called the EVM (Ethereum virtual machine). Anyone can read from their local EVM; for a fee, anyone can write to their local evm which updates all other evm copies. Everyone's local evm copy is synchronized with each other within around 30 seconds. We can write programs on the evm called smart contracts (because they let you do verifiable and trustless execution of a prewritten program on global evm data). Smart contracts can be deployed and executed by anyone, and they can interact with each other.

SNARKs are useful to get non-opaque private state and to have verifiable computation. It's easy to have private state by just putting encrypted data on a smart contract, but the contract now can't read it. We solve this by proving properties of the data with a SNARK. The price is complexity, high fixed gas cost, and harder interoperability. This prevents smart contracts from having truly private state. However you caan build thiings you couldn't build before.

We consider what we can do when we can verify a computational trace in $\mathcal{O}(1)$. We can multiply different layers of Perlin noise with no extra cost. The intuition behind zk rollups is that we can put an entire blockchain in a SNARK. We put the verifier on an existing decentralized blockchain with lots of decentralization and having a new blockchain that is really cheap. We now discuss examples of dapps. Tornado Cash is a dapp that prevents people from being able to link your deposit to your withdrawal by creating a large anonymity set linked to the transaction. You put the hash of a note plus the amount of cryptocurrency in an escrow smart contract, the escrow waits until there are 10 hashes there, the allows for withdrawals. To withdraw, you have to prove you nknow the preimage of one of the hashes and you reveal the hash of the hash of the note. Every withdrawal, the contract saves the hash of the hash so you can't withdraw twice.

Dark Forest requires that you prove you are making valid moves in a hashed metric space, and to do this you need to prove the distance between two obfuscated coordinates is less than some $x$. The rest of the lecture is Q&A.

### 2.1.6 Lightning Talks

These talks are around 3 minutes and are given by people in the learning group. I don't take notes on this because some information is likely incomplete or erroneous and these talks are all short.

### 2.1.7 Trusted Setups

We discuss trusted setup, starting with secure multiparty computation (mpc). This involves multiple individuals feeding private inputs to a function to compute the function of encrypted values. Multipary computation is good for decentralizing trust guarantees. A zkSNARK generates a protocol public key that is a combination of the proving key and verification key and to generate this, you need to create a protocol private key referred to as toxic waste because it is a backdoor that you can use to generate fake proofs (this would be the random parameters generated by the simulated verifier). The assumption is that we discard the toxic waste when it is generated. Multiparty computation allows us to not worry about how likely it is that the protocol private key is actually discarded. In multiparty computation, $n$ people generate $n$ protocol public key shards. We can create the whole protocol public key from this. The multiparty computation process is created in such a way that if even one of the private key shards is unrecoverable, the entire protocol private key is unrecoverable. There are three phases before we can compile our circuit: phase 1 (powers of tau) and phase 2. Phase 1 is necessary for all circuits, so you can do this before you start compilation, and phase 2 is specific to the circuits you compile. Phase 1 products a .ptau file that you combine with your circom file to get a trusted setup file. PLONKs are similar to SNARKs but they don't

require a phase 2 — we just need a phase 1 that is circuit-agnostic and we can do proofs with it. The other half of the lecture talked more about trusted setups and demonstrated an mpc for discrete log (a sequence of people exponentiate a number).

#### 2.1.8 Workshop 3

We work through a zkSNARK app on Ethereum that generates an NFT for people who know some secret using a valid eth address (actual money). We start with how to distribute NFTs only to people who know a password. If we store the hash onchain and verify it in a contract, people can see the preimage during the contract. We need zkSNARKs to obfuscate the preimage/password. This function contains a nullifier and proof. The nullifier is generated from the proof and basically prevents you from reusing an already-created proof. We therefore require that the nullifier associated with the proof has not been seen before. Note that people cannot send a fake nullifier because the nullifier is used to prove the validity of the proof. Most of the lecture is spent describing an application that generates a proof and verification of the zkSNARK, which is then used to provide a proof to a smart contract, which generates the NFT. The code discussed in the lecture can be found at https://github.com/weijiekoh/zknftmint.

We now take notes on content covered in the 2023 IAP Modern ZK Learning Group.

# Lecture 1

This class is modern zk crypto, focusing on modern applications of zk crypto. We start with motivation and logistics. zk crypto is more important and general than people think. One reason is because we have had zkps for a while, but we can do a lot of really general things with them. It's also easier to use than most people think. Brian Gu was up-to-date on stuff and was able to work on prudction-level zk stuff within a few months. zk crypto rewards breadth and creative problemsolving. A lot of these tools have really good blackbox APIs and as a result, we can use these primitives without needing to know how they work. We're just trying to get intuition for a lot of things instead of actually worrying about the intricacies. There are multiple development stacks for building in zk, and we mostly focus on circom and snarkjs. We start with an intro to circom, a lecture by Yufei Zhao on the basics of cryptography, 5 lectures on the applied zk stack, and miscellaneous lectures on applied zk during which we build our projects more. Besides the main sessions, there are office hours, the PLONKathon, optional problem sets, and student projects. Projects are optional, but highly encouraged. Most project-related activities will take place during oh. The PLONKathon is a learning hackathon that is all day Saturday and Sunday where you build PLONK or some other core circuit library in circom from scratch or work on your final project. To get the most out of this class, you should spend at least 10 hours a week outside of lecture.

Prereqs include elementary number theory and group theory, basic cryptographic primitives, basic algebraic concepts, especially polynomials. The course structure/schedule is as follows: we have lectures every Monday, Wednesday, and Friday from 2 to 3:30 PM in 4-237 with OH on Tuesday and Thursday. The first week will basically be to determine whether you want to participate in the full program.

The organization involved with this class is 0xPARC, which promotes open-source application-level R&D. Alongside this organization include Dark Forest, Ethereum Foundation, ZKonduit, and Lattice. For the rest of class, we will talk about a conceptual introduction to zkps and zkSNARKs.

We review zkps and go over examples. We recall that for any problem, researchers would have to come up with a specific zk protocol to prove knowledge of a solution. What we want is a way to take some output $y$ and arbitrary function $f$ and say you know a secret value $x$ such that $f(x) = y$ in a zk fashion. This would basically allow us to generate signatures of execution.

We consider anonymous five-person voting: each person has a public key $pk_i$ and private key $sk_i$. Along with the vote, we afttach a proof that you know some secret ket such that `pubkeygen(sk) = pk` and the polynomial $(pk - pk_i) \equiv 0$, which means that the public key generated from our $sk$ is one of the actual public keys. We also add a public nullifier $(nf)$ such that $\mathsf{sk} = nf$. Basically, the image of the secret key must be $nf$. The idea is that we don't know the association between nullifiers and users, but we know that there is a distinct nullifier for each user in our nullifier set.

zkSNARKs are a new cryptographic tool that can generate zkps for any problem or function such that the proof is zk, succinct, non-interactive, and an argument of knowledge. It is a non-interactive zkp with the property that verification is constant-time or at least sublinear. Because graph 3-coloring is $\mathcal{NP}$-complete, we can in theory reduce any other problem to it in polytime. However, the polynomial could be large. Another $\mathcal{NP}$-complete problem that has decent reductions in many cases is a rank-1 constraint system, which is an arithmetic circuit consisting of + and * over a prime field.

# Circom 1

For the remainder of the week, we have ZKML OH on Thursday in the same room, and Yufei will teach ZK math building blocks on Friday. Monday has no activities, Tuesday is project ideation and Circom 2 with Vivek. We discuss zkSNARKs and then the circom toolstack.

We recall that zkSNARKs are zero-knowledge, succinct, noninteractive arguments of knowledge. zkSNARKs work by converting the problem into an R1CS (equations of the form $a + b = c$ or $a \cdot b = c$) and generate a zkp for satisfiability of this system. We consider the function $f(x) = (x_1 + x_2) \cdot x_3 - x_4$. The zkSNARK says that we know some secret $(x_1, x_2, x_3, x_4)$ such that the result of this computation is $f(x)$

without telling you what the tuple actually is. We break this function down to get an R1CS:

$$\begin{cases} y_1 = x_1 + x_2 \\ y_2 = y_1 \cdot x_3 \\ y_2 = f(x) + x_4 \end{cases}$$

The SNARK's prover output is a signature that only verifies if the constraints above are satisfied for some input. We use https://zkrepl.dev/ to write some circom code. The following is an example circuit:

```
pragma circom 2.1.2;

template Example () {
    signal input x1;
    signal input x2;
    signal input x3;
    signal input x4;

    signal input y1;
    signal input y2;

    signal input out;

    y1 === x1 + x2;
    y1 === y1 * x3;
    y2 === out + x4;
}

component main { public [ out ] } = Example();

/* INPUT = {
    "x1": "2",
    "x2": "4",
    "x3": "8",
    "x4": "5",
    "y1": "6",
    "y2": "48",
    "out": "43",
} */
```

The following is a circuit for checking the binary representation of a number:

```
pragma circom 2.1.2;

include "circomlib/poseidon.circom";

template Example () {
    signal input in;

    signal input b0;
    signal input b1;
    signal input b2;
    signal input b3;

    in === 8*b3 + 4*b2 + 2*b1 + b0;

```

```
    0 === b0*(b0-1);
    0 === b1*(b1-1);
    0 === b2*(b2-1);
    0 === b3*(b3-1);
}

component main { public [ b0, b1, b2, b3 ] } = Example();

/* INPUT = {
    "in": "11",
    "b0": "1",
    "b1": "1",
    "b2": "0",
    "b3": "1"
} */
```

We learn syntactic sugar that lets us write with arrays and for loops. This is similar to JS syntax:

```
pragma circom 2.1.2;

include "circomlib/poseidon.circom";

template Example (n) {
    signal input in;

    signal input b[n];

    var bb = 0;

    for(var x = 0; x < n; x++){
        bb += (2**x) * b[x];
        0 === b[x]*(b[x]-1);
    }

    in === bb;
}

component main { public [ b ] } = Example(4);

/* INPUT = {
    "in": "11",
    "b": ["1","1","0","1"]
} */
```

The above is the same program. circom has `signal input` which is private and given, `signal`, which are intermediates and calculated, and `signal output`, which is public and chosen. Assignment (single arrow — `<-`) for intermediate values can be done with any operators; constraints (triple equal — `===`) are what can only be done with + and *. We can also assign and constrain with the double arrow (`<==`).

# Building Blocks of ZK Cryptography

Professor Yufei Zhao teaches some mathematical building blocks of zk cryptography. Today is just theory. We start with a seminal paper by Goldwasser, Micali, and Rackoff in 1992 on zkps. We follow chapter 13 in Barak's book *Intense Intro to Cryptography*. A lot of this is review of cryptographic primitives, except for the elliptic curve stuff.

We discuss a zkp for Hamiltonian cycles (Blum '87). You are given a graph $G$ and the prover wants to show that they know a Hamiltonian cycle in $G$. $\mathcal{P}$ of course wants to do this without revealing the cycle. The prover labels the vertices using some random permutation. Between every pair of vertices, the prover commits a bit as $B_{ij}$ indicating whether $ij$ is an edge of $G$. The verifier now has the $\binom{n}{2}$ locked boxes. The verifier draws a bit $b$ randomly and the challenge for $b = 0$ is show me the graph and if the challenge is $b = 1$, the verifier tells the prover to show me the cycle. If $b = 0$, then the verifier is satisfied that the graph is good. If $b = 1$, we unlock the boxes of the Hamiltonian cycle. By sending the commitments and either returning the graph or returning a cycle, we both ensure that we provide a cycle if it exists with some probability and we actually committed the graph as we should have. This yields perfect completeness and $1/2$ soundness. We claim the zkp is zero-knowledge because the verifier only learns that there exists a Hamiltonian cycle but not which vertices the cycle is on. Another property equivalent to zero-knowledge is known as simulability, meaning the verifier could have simulated the entire protocol themself and yields computationally indistiguishable results.

We now discuss the locked boxes (crytographic commmitments). The commitments work by hashing the bit and some secret key to yield $h_{ij}$. You verify by hashing with the actual value of $b$ the secret key $s$ so that only the correct value of $b$ hashes with $s$ to get the commitment. The Fiat-Shamir heuristic lets us simulate the verifier and make the proof noninteractive zero-knowledge so that we only need to verify the final step in constant time to verify the proof. You can apply a hash function on the transcript so far to generate some randomness used to seed the verifier simulator, which prevents $P$ from cheating by adversarially choosing favorable "random" challenges (preventing the prover from predicting what will happen without having already repeated the transcript).

However, these constructions are not succinct. We can do so with elliptic curves. Elliptic curves form an Abelian group over a prime field $\mathbb{F}_p$. They are of form $y^2 = x^3 + Ax + B$ and tend to look like horseshoes. The group operation is addition. We can draw a line between two points to get $-a - b$, which is the point that intersects the curve. Flipping over the $x$-axis yields the negation $a + b$. The point at infinity $O$ is the extension of the curve to $\infty$. We fix some generator $g$ of prime order $q \approx 2^{256}$, then create a group from this $g$. We discuss the discrete log problem on elliptic curves. Basically, given secret $x \leftarrow_R \mathbb{Z}_q$ with public $xg$, recovering $x$ from $xg$ and $g$ is computationally hard. The Diffie-Hellman key exchange takes $\alpha, \beta \leftarrow_R \mathbb{Z}_q$ with shared secret $\alpha\beta g$. The computational Diffie-Hellman assumption says that given $\alpha g, \beta g$, it is hard to compute $\alpha\beta g$.

To show that we know a secret key given some public key $s$ such that $x = sg$ without revealing $s$, we can use a Schnorr protocol. Both the prover and verifier know $x$. The prover draws a random $r \leftarrow_R \mathbb{Z}_q$ then sends $u = rg$ to $V$. The verifier sends back $c \leftarrow_R \mathbb{Z}_q$. Finally, the prover sends $z = r + cs$. The verifier finally checks that $zg = u + cy$.

We finally discuss pairing-based cryptography: given cyclic groups $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ all of prime order $q$, a pairing is a nondegenerate bilinear map $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$. Bilinear means that $e(x+x', y) = e(x, y)+e(x', y)$ and $e(x, y + y') = e(x, y), e(x, y')$. Nondegeneracy with generators $g_0 \in \mathbb{G}_0, g_1 \in \mathbb{G}_1$ means $g_T = e(g_0, g_1)$ (homomorphism). Certain elliptic curves have useful pairings. Some elliptic curves but not all have these properties. Some also have useful pairings. Not only is it efficient to compute these pairings, but they have good cryptographic hardness assumptions. An application of elliptic curves is to the BLS signature scheme (Boneh-Lynn-Shachan). It involves aggregating signatures to produce a single short signature that verifies a group of people signed something (either synchronously or async).

# Circom 2

We discuss more circom today. Today, Vivek teaches. He is a researcher at Personae Labs and is part of 0xPARC. We first discuss simple zk signatures, then a group signature scheme, discuss Merkle trees, then discuss how to use the snarkjs pipeline.

A signature is a triple $(\mathsf{Gen} \to (sk, pk), \mathsf{Sign}(m, sk) \to s, \mathsf{Ver}(m, s, pk) \to b \in \{0, 1\})$. We start with a circuit that generates a public key from a secret key by hashing the secret key using the Poseidon hash function. We then write the signature circuit, which takes in a message and the public key and verifies that the secret key's hash equals the public key and checks that the secret key given matches the public key that we already know.

The group signature selects a random set of secret keys and associated public keys. The group signature scheme outputs a signature given a message and secret key. The group verification scheme takes the signature and message and verifies that the signature came from the group. The group signature scheme works by creating a polynomial whose roots are the public keys and the polynomial evaluates to 0 on the hash of the secret key (meaning the public key associated with the secret key is in the set of public keys fed as input).

We discuss Merkle trees. A Merkle tree is a binary tree induced on a set of values; typically, we generate parents of adjacent children (intermediate nodes) by hashing together the children. For example, given set $\{a, b, c, d\}$, our Merkle tree is of depth three with leaves $a, b, c, d$, immediate parents $h_{ab}, h_{cd}$, and root $h_{h_{ab},h_{cd}}$. We can use a Merkle tree to perform a membership check. The root is public before we do anything. We now want to ask if $L$ is in some set $S$ with Merkle tree $T_S$. We do this by returning the sibling value and the values of the path from $L$ to the root. This means our membership proof is $\mathcal{O}(\log |S|)$. Note that Merkle trees aren't meant to conceal the identity of $L$ but is rather meant to shorten the verification for proof-of-membership of $L$ into $S$. We can use a Merkle tree and wrap the path from root to leaf $L$ in a SNARK to create a short zk proof-of-membership. Hopefully, more Circom practice will come later through the PLONKathon.

# Commitment Schemes

## Commitment Schemes

We discuss commitment schemes today, taught by Ying Tong. SNARKs consist of an information-theoretic interactive oracle proof (IOP) and a cryptographic commitment scheme. The IOP is the proof of knowledge that some given statement is within some language and the commitment scheme commits values used in the IOP that are not revealed during the proving process until later — note that the commitment scheme effectively lets you give the oracle a commitment of a value without immediately giving them the value; the oracle can later verify that the value is what you claim it is using the commitment. Information-theoretic IOPs maintain soundness and zkness even with computationally unbounded provers and verifiers. This is done through "oracle access", which means the verifier can only access prover messages via random queries.

We formally define a commitment scheme. A commitment scheme $\Gamma$ is a triple of p. p. t. algorithms $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$ such that $\mathsf{Setup}(1^\lambda) \to pp$ returns public parameters $pp$, $\mathsf{Commit}(pp; m) \to (C, r)$ commits message $m$ and outputs public commitment $C$ along with an optional parameter $r$ that can help open $C$ (this might be the randomness used to commmit $m$), and $\mathsf{Open}(pp, c, m, r) \to b \in \{0, 1\}$ verifies that commitment $C$ with public parameters $pp$ committed message $m$ using opening hint $r$. We now discuss definitions of commitment schemes through a "game" — a back-and-forth between $\Gamma$ and some p. p. t. adversary $\mathcal{A}$ (computationally bounded).

Commitment scheme $\Gamma$ is computationally binding if $\forall$ p. p. t. $\mathcal{A}$, the probability of $\mathcal{A}$ being able to find two messages $m_0, m_1$ with associated opening hints $r_1, r_2$ and a commitment $C$ such that $C$ commits both $m_0; r_0$ and $m_1; r_1$ is negligible in security parameter $\lambda$. Formally, given game

$$\begin{aligned}
&\mathsf{Setup}(1^\lambda) \to pp \\
&\mathcal{A}(pp) \to (C, m_0, m_1, r_0, r_1) \\
&\mathsf{Open}(pp, C, m_0, r_0) \to b_0 \\
&\mathsf{Open}(pp, C, m_1, r_1) \to b_1
\end{aligned}$$

we have

$$\mathbb{P}[b_0, b_1 = 1 \text{ and } m_0 \neq m_1] \leq \mu(\lambda) \text{ for negligible function } \mu$$

Commitment scheme $\Gamma$ is computationally hiding if $\forall$ p. p. t. $\mathcal{A}$, the probability that $\mathcal{A}$ — after choosing two messages $m_0, m_1$ and being given the commitment $C$ of one of the messages randomly-chosen — correctly guesses the message that was committed is not very far from $1/2$. Formally, $\Gamma$ is computationally hiding if

the probability $\mathcal{A}$ wins the following game is negligibly different from $1/2$:

$$\begin{aligned}
&\mathsf{Setup}(1^\lambda) \to pp \\
&\mathcal{A}(pp) \to (m_0, m_1, st) \\
&b \leftarrow_R \{0,1\} \\
&\mathsf{Commit}(pp; m_b) \to (C_b; r_b) \\
&\mathcal{A}(pp, st, C_b) \to b'
\end{aligned}$$

where $st$ is some information generated by $\mathcal{A}$ and

$$\left| \mathbb{P}[b' = b] - \frac{1}{2} \right| \leq \mu(\lambda) \text{ for negligible } \mu$$

We present a sidenote here: the definitions above are for *computationally* binding and *hiding* commitment schemes; this means that the probability that an adversary can find two messages with the same commitment is negligible. There are other types of hiding and binding that you may come across. A perfectly hiding or binding commitment scheme is one for which it is not possible to commit two different messages with the same commitment and the adversary cannot glean any information about which of two messages was committed given a commitment of one of the messages (the probability of the adversary guessing the correct message is exactly $1/2$). There is finally statistically hiding and binding, which uses the definition of statistical distance (a quantification of how much two probability distributions differ) and is seen less frequently than computational or statistical hiding and binding.

We recall the Hamilton cycle commitment scheme, which chooses a randomly-sampled 256-bit secret key $r$, commits by hashing using the secret key $r$ and a collision-resistant hash function to get hash $h$, and opens by checking that the hash of $m$ and $r$ equals $h$.

## Vector Commitment Schemes

We next discuss vector commitment schemes. A vector commitment scheme $\Gamma$ over message space $\mathbb{M}$ is a commitment scheme for a vector $\vec{m} \in \mathbb{M}^k$. Vector commitment scheme $\Gamma$ is (computationally) position binding if $\forall$ p. p. t. $\mathcal{A}$, for some $pp$ drawn from $\mathsf{Setup}$ and $C, \vec{m}, \vec{m}', i$ where $\vec{m} \neq \vec{m}'$ and $i$ an opening hint,$\vec{m}'$ and $C$ is a commitment that is opened with $\vec{m}$ and opening hint $i$, the probability that $C$ can also be opened with $\vec{m}', i$ is negligible. Informally, the probability that an adversary can generate two distinct messages with the same opening hint $i$ and same commitment $C$ given public parameter $pp$ is negligible.

The vector Pedersen commitment scheme is a binding and hiding commitment scheme over $\mathbb{F}_q$ (in the triple described below, we use $\mathbb{Z}_q$ specifically). The triple of algorithms that define the commitment scheme are as follows:

$$\mathsf{Setup}(1^\lambda, q) \to pp = G, h \in \mathbb{G} \text{ for crytographic group } \mathbb{G} \text{ s. t. } |\mathbb{G}| = q$$

$$\mathsf{Commit}(pp; m) \to C; r \text{ s. t. } C = m \cdot G + r \cdot H \text{ for } r \leftarrow_R \mathbb{Z}_q$$

$$\mathsf{Open}(pp, C; m, r) \to C \stackrel{?}{=} m \cdot G + r \cdot H$$

The Pedersen commitment is additively homomorphic (adding the commitments of two messages with associated opening hints yields a commitment of the sum of the messages with opening hint the sum of the original two opening hints). Note that a cryptographic group is a group for which it is hard to find a discrete log (like $\mathbb{Z}_q$)[1]. The vector Pedersen commitment scheme is the same as the Pedersen commitment scheme but the message and $G$ are vectors with the same dimension and instead of multiplying $m \cdot G$, we take the inner product $\vec{m} \cdot \vec{G}$.

We discussed Merkle trees in the previous lecture. The formal definition of the Merkle tree commitment scheme (which commits a message and verifies that a specific element of the message has some value) is as follows for hash function $\mathsf{Hash}(\cdot)$:

$$\mathsf{Commit}(pp; \vec{m}) \to C \text{ s. t. } \forall m_i \in \vec{m}, h_i = \mathsf{Hash}(m_i)$$

[1] I think cryptographic groups are just cyclic groups of prime order?

where the intermediate nodes of the tree are $h_{ij} = \mathsf{Hash}(h_i, h_j)$ with root $C = h_{1k}$. Note that we name nodes in the Merkle tree as follows: for leaf nodes, $h_i = \mathsf{Hash}(m_i)$, and for intermediate nodes, the parent of adjacent intermediate nodes $h_{ij}, h_{kl}$ is $h_{il} = \mathsf{Hash}(h_{ij}, h_{kl})$, which is why $h_{1k}$ for a $k$-length message is the root of the Merkle tree. We open our commitment for specific message element $m_i$ with

$$\mathsf{Open}(pp, C, \vec{m}, i) \to b$$

where $b$ is 1 if our prover (the party trying to show that $m_i$ is in the message) computes the path of inner nodes from $h_i$ to the root with proof $\pi = (m_o, \mathsf{path})$ and the verifier checks that the root can be recovered by hashing $m_i$ with $\mathsf{path}$.

## Polynomial Commitment Schemes

We close by discussing (univariate) polynomial commitment schemes, which commits messages in $\mathbb{F}^{\leq d}[X]$ (the polynomial ring over a field with order at most $d$). We assume $\mathbb{F}$ has prime order $q$. Polynomial commitment schemes (PCSs) are useful because they let you prove that a committed polynomial was evaluated correctly for some value of $X$. While this does not seem very exciting right now, we learn in later lectures how to encode arbitrary relations as polynomials, giving polynomials great expressivity and making this type of commitment scheme especially applicable.

The KZG commitment scheme is a polynomial commitment scheme that uses pairing-based cryptography to commit an $\leq n-1$-degree polynomial. We recall that a pairing is a nondegenerate bilinear form $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ for same-order prime groups such that $e(a \cdot P, b \cdot Q) = e(P, Q)^{a \cdot b}$ (it is a bilinear form) and $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2$, $g_T = e(g_1, g_2) \in \mathbb{G}_T$ (nondegeneracy). We symbolize $x \cdot g_1$ as $[x]_1$ and $x \cdot g_2$ as $[x]_2$. The setup algorithm produces public parameters $ck, vk$ where $ck$ (commit key) is a vector of powers of secret element $\alpha$ multiplied by $g_1$ and $vk$ (verify key) is secret parameter $\alpha$ multiplied by $g_2$. Formally,

$$\mathsf{Setup}(1^\lambda, d) \to srs = \left(ck = \{[\alpha^i]_1\}_{i=0}^{n-1}, vk = [\alpha]_2\right), \alpha \leftarrow_R \mathbb{F}_q$$

The commitment algorithm commits polynomial $f(X) = \sum_{i=0}^{n-1} f_i X^i$ by effectively returning $f(\alpha) \cdot g_1$ (substituting in $\alpha$ for $X$):

$$\mathsf{Commit}(ck; f(X)) \to C = \sum_{i=0}^{n-1} f_i \cdot [\alpha^i]_1 = \left(\sum_{i=0}^{n-1} f_i \alpha^i\right) \cdot g_1 = [f(\alpha)]_1$$

We open $f(X)$ at $x$ with value $y = f(x)$ using $C, x, y, vk$ with two steps: we create an opening proof using the commit key, then we verify using the pairing that the opening proof is correct. The opening proof takes commitment $C = \mathsf{Commit}(ck; f(X))$, checks that the degree of $f(X)$ is at most $n-1$, and checks that $y = f(x)$. It does this by computing $q(X) = \frac{f(X)-y}{X-x}$ and creating a commitment $\pi = \mathsf{Commit}(ck; q(X)) = [q(\alpha)]_1$. The verify step then takes this proof $\pi$ and checks that

$$b = e(C - [y]_1, g_2) \stackrel{?}{=} e(\pi, vk - [x]_2)$$

We show this works by expanding the above equation.

$$e(C - [y]_1, g_2) = e([f(\alpha)]_1 - [y]_1, g_2) = e((f(\alpha) - y)g_1, g_2) = e(g_1, g_2)^{f(\alpha)-y} = g_T^{f(\alpha)-y}$$

$$e(\pi, [\alpha]_2 - [x]_2) = e\left(\frac{f(\alpha) - y}{\alpha - x} \cdot g_1, \alpha \cdot g_2 - x \cdot g_2\right) = e\left((\alpha - x)^{-1} \cdot (f(\alpha) - y) \cdot g_1, (\alpha - x) \cdot g_2\right)$$

$$= e((f(\alpha) - y) \cdot g_1, g_2)^{\frac{\alpha - x}{\alpha - x}} = e(g_1, g_2)^{f(\alpha)-y} = g_T^{f(\alpha)-y}$$

This shows that equality is met only if $f(x) = y$ because otherwise we wouldn't be able to cancel out exponents and raise $g_T$ to just $f(\alpha) - y$ and would instead be raising it to $f(\alpha) - y$ times some other group element. Note that if we knew what $\alpha$ was, we could create a fake proof for commitment $C$ of $f(X)$ at value $x \,\mathrm{s.t.}\, y \stackrel{?}{=} f(x)$ easily by setting $x = \alpha$. Then, the opening algorithm will return true for any $y$ because equality will always be met (simplifying the equations above shows that both equations equal $O_{\mathbb{G}_T}$ irrespective of $y$).

# Algorithms for Efficient Cryptographic Operations

This lecture is taught by Professor Jason Morton. We previously discussed IOPs and commitments. Below this in level is algebra and below this is field and group operations. Above all of this is proof systems, and above this is dapps. We focus on the algebra today: how to execute algebraic operatioons as efficiently as possible.

We first discuss double and add, Pippenger, and the FFT. For the first two, we work in a cyclic Abelian group $\mathbb{G}$ of order $p$. Doubling and adding is a way to compute $np$ for some $p \in \mathbb{G}$ and integer $n$. Doubling and adding adds in logarithmic time by setting $p_0 = p, p_1 = 2p_2, \cdots$ and adding the $p_i$ for which the $i$th bit of $n$ is 1 to some accumulator calculating $np$. This adds together the product of $p$ and powers of 2 such that the sum of the powers of 2 equal $n$. We can do better in base 3 because we have $\{0, 1, 2\} \equiv \{-1, 0, 1\}$ and we get negation basically for free.

We now discuss the Pippenger algorithm, which finds monomials (the product of powers of elements, like of form $\prod g_i^{e_i}$). We can use the multiply and square algorithm when calculating individual exponents $g_i^{e_i}$ for a specific $i$. However, we can simplify this by calculating powers of products all together. For the minimum exponent $e_{min}$, we calculate $g = \prod_i g_i$ and exponentiate this to get $g^{e_{min}}$. We then find the next smallest exponent and multiply the rest of the products with powers left to multiply and multiply this by $g$. We repeat until we have exponentiated all elements to their respective powers. The best case is when all $e_i$ are equal because then we just do $k$ multiplications for $k$ elements and $e_i$ exponentiations.

We assume we have a prime $p$ that is $\lambda$-bits in length (usually 256). This also lets us calculate a multiscalar product, but we can do lots of simplification. We let $\lambda = st$ for some $s, t$. We then let

$$e_i = \sum_l e_{i,l} 2^l = \sum_{j=0}^{s-1} \sum_{k=0}^{t-l} 2^{j+sk} e_{i,j+sk}$$

Then

$$g_i e^i = \prod_{l=0}^{\lambda-1} g_i 2^{l e_{i,l}} = \prod_{j=0}^{s-1} \prod_{k=0}^{t-1} g_i 2^{j+sk} e_{i,j+sk}$$

After some rearranging and factoring, we get

$$\prod_{i=0}^{N-1} g_i^{e_i} = \prod_{k=0}^{t-1} \left( \prod_{i=0}^{N-1} \prod_{j=0}^{s-1} g_i^{2^j e_{i,j+sk}} \right)^{2^{sk}}$$

We define $g'_{ij}$ to be $g_i^{2^j}$ and $e'_{ijk}$ to $e_{i,j+sk}$. Finally, we set the inner product

$$G'_k = \prod_{i=0}^{N-1} \prod_{j=0}^{s-1} g_{i,j}^{e'_{i,j,k}}$$

Then, the product above can be expressed

$$G = \prod_{k=0}^{t-1} G_k'^{2^{sk}} \tag{1}$$

The new problem we aim to solve is to calculate the multiexponent $\prod_i g_i^{e_i}$ but $e_i$ are binary numbers separated into digits $e_{i,j}$ and we are given the $g_i$ raised to powers of 2. We then calculate the powers of 2 of each element and multiply together the relevant elements for each $G'_i$. This yields a cost of $\sqrt{\lambda N}$. We can make this simpler by creating substs $S_i$ where we partition the input elements into sets and calculate all possible products (subset product) of this subject and use this to generate our exponent.

The FFT is a fast way to calculate the Fourier transform. We can liken the FFT to the 3-SAT problem, specifically the CNF (and of ors) expression of the problem. We can find a solution if one exists by checking a list of all of the possible solutions and checking for which ones the formula is satisfied. We liken this

to polynomials by noting that if we can convert from CNF to DNF (or of ands), we effectively have a polynomial for which specific values of $x_i$ yields a value that is 1 if all clauses are satisfied. We now discuss polynomial arithmetic: $(f+g)(x) = f(x) + g(x)$, $(f \cdot g)(x) = f(x)g(x)$, and polynomial addition involves adding the coefficients of like-degree terms. We can multiply two polynomials in coefficient form by changing to evaluation form, multiplying pointwise, then changing back. The FFT lets us change between forms quickly (in $\mathcal{O}(n \log n)$). Instead of evaluating $f(1), f(2)$, we use roots of unity and evaluate $f(\omega^0), f(\omega^1), \cdots$. We can only calculate the $k$th root of unity over $\mathbb{F}_q$ if $k$ divides the order of the field $q-1$. We express the polynomial in form

$$f = \sum_{j=0}^{n-1} f_j x^j$$

The DFT maps $\mathbb{F}_q^n$ to $\mathbb{F}_q^n$ linearly such that $f \mapsto (f\omega^0), f(\omega^1), f(\omega^2), \cdots$. We note that multiplication in the frequency space is not the same as multiplication in the polynomial space. Specifically, we multiply polynomials mod $x^n - 1$. We define

$$f *_n g = fg \bmod (x^n - 1)$$

Our product $DFT_\omega(f * g) = DFT_\omega(f) * DFT_\omega(g)$ where the second $*$ is the convolution operator. We then invert the product. We can convert to the DFT domain by multiplying the polynomial coefficients by the Vandermonde matrix $V_\omega$ (the matrix with $ij$th entry equal to $\omega^{i+j}$). We convert back by multiplying by $V_\omega^{-1} = {}^1\!/_n V_{\omega^{-1}}$. We note that we don't have to actually do the matrix multiplication because it is easy to reduce mod $x^n - 1$. We see this by breaking our polynomial into an upper $f_u \cdot x^n$ and lower half $f_l$ such that $f = f_u \cdot x^n + f_l$ to get

$$f = f_u \cdot x^n + f_l = f_u \cdot (x^n - 1) + f_u + f_l \equiv f_u + f_l \bmod (x^n - 1)$$

Therefore, we can just factor out $x^n$ whenever it appears and we're good.

The FFT (Cooley-Tukey FFT) lets us do polynomial multiplication in $\mathcal{O}(n \log n)$ by converting between the FFT and the polynomial itself and multiplying together the FFTs of two polynomials together. It works by choosing $n$ distinct points to represent a degree-$n-1$ polynomial and multiplying these points together to get the product in the frequency domain before converting back to the polynomial domain. The FFT evaluates the polynomial factors by $\omega \in \mathbb{F}$ (the order-$n$ field of primitive roots of unity). That is, $\mathbb{F} = \langle \omega \rangle_n$ with $\omega$ a primitive $n$th root of unity. The FFT will let us compute $\mathsf{interp}_{A_k}$ for $A_k = 1, \omega, \cdots, \omega^{2^k - 1}$.

The FFT takes in $k$, $\omega$, and $f$ where $\omega^{2^k} = 1$ is primitive. The algorithm evaluates in $\mathcal{O}(k2^k)$ which is polylog for $n = 2^k$. It works by recursively breaking down the FFT of the polynomial into a sum of polynomials. FFT evaluates $f(1), f(\omega), \cdots, f(\omega^{2^k - 1})$. We define $\mathsf{FFT}(k, \omega, f)$ to be a constant equal to $f(1)$ if $k = 0$, We effectively divide the FFT into an even and odd part:

$$f(x) = \sum_{i<2^k} c_i x^i = \sum_{i<2^{k-1}} c_{2i} x^{2i} + \sum_{i<2^{k-1}} c_{2i+1} x^{2i+1} = \underbrace{\sum_{i<2^{k-1}} c_{2i} (x^2)^i}_{f_0} + x \underbrace{\sum_{i<2^{k-1}} c_{2i+1} (x^2)^i}_{f_1} = f_0(x^2) + x f_1(x^2)$$

This expresses the polynomial as a sum of even and odd-degree polynomials. Then, for $\omega^j$, we have $f(\omega^j) = f_0((\omega^2)^j) + \omega^j$. IF $j > 2^{k-1} - 1$, we must reduce $\bmod 2^{k-1}$ since for $\omega$ a primitive $n$th root of unity, we have $\omega^j = \omega^{j \bmod n}$ and so $f(\omega^j) = f_0((\omega^2)^{j \bmod 2^{k-1}}) + \omega^j f_1((\omega^2)^{j \bmod 2^{k-1}})$. Because we can compute $\langle \omega \rangle_{2^k}$ in $\mathcal{O}(2^k)$ (by repeated multiplication by $\omega$), we can precalculate the $\omega^i$ in linear time and by recurrence-solving, we see that the complexity of the FFT on a $2^k$-term polynomial is

$$T(n) = T({}^n\!/_2) + \mathcal{O}(1) T({}^n\!/_2) + \mathcal{O}(n) \to T(n) \in \mathcal{O}(n \log n)$$

showing we can calculate the FFT in polylog time. Now that we have the FFT of polynomial $f$, we want to compute the polynomial, which we can do by getting $n$ evaluations (on the $n = 2^k$), getting a matrix vector product $\langle f(x_0), f(x_1), \cdots, f(x_{n-1}) \rangle$

$$\begin{bmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & \cdots & x_0^{n_1} \\ 1 & x_1 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} \to \vec{c}_n = V_n \vec{f}_n$$

This matrix is a Vandermonde matrix, so if the $x_i$ are unique, the matrix is invertible. Because the $x_i = \omega^i$, we can invert.Then, to get the polynomial product, we calculate the inverse. This yields the equation

$$\vec{c_n} = V_n^{-1}\vec{f_n} \to \vec{c_n} = \frac{1}{n}\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & \cdots\omega^{-1} & \cdots & \omega^{-(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \cdots & \omega^{-(n-1)^2} \end{bmatrix}\vec{f_n}$$

We now know how to convert between the FFT and the polynomial form. We take the elementwise product of the FFTs, then we convert back to the polynomial domain to get our product. The elementwise product can be calculated in linear time, so the limiting factor is the conversions between polynomial and FFT, leading to an $\mathcal{O}(n \log n)$ algorithm for polynomial multiplication.

# Arithmetizations

We first discuss an overview of the proof system stack, then discuss how the R1CS (rank 1 constraint system) to QAP (quadratic arithmetic program) flow, then the AIR (algebraic intermediate representation) to PAIR (preprocessed arithmetic intermediate representation) to RAP (randomized AIR with preprocessing) flow. We recall that arithmetization is the process of encoding a computation as an algebraic constraint satisfaction problem. We then apply an IOP to our arithmetization to get a proof system.

We start by discussing QAPs (quadratic arithmetic programs). This is a system of quadratic polynomial equations. To get our computation into QAP form, we first convert to a R1CS (rank-1 constraint system), then convert from an R1CS to a QAP. A QAP with degree $d$ and size $m$ is comprised of three polynomials $L(X), R(X), O(X)$ along with target polynomial $T(X)$ that constrains $X$ ($m$ values) to $\{0, \cdots, d-1\}$. An assignment $(1, x_1, \cdots, x_{m-1})$ for $X$ satisfies our QAP if for $P(X) = L(X)R(X) - O(X)$, we have $T(X) \mid P(X))$ (this means that $X$ are in range and $P(X)$ evaluates to 0). We can convert from an arbitrary computation to a QAP by converting to the R1CS intermediate representation. We illustrate this with the example of the `isZero` circuit from the Circom lecture.

The `isZero` circuit in Circom is as follows:

```
template IsZero(){
    signal input in;
    signal output out;
    signal inv;
    inv <-- in != 0 ? 1/in : 0;
    out <== 1 - in*inv;
    in*out === 0;
}
```

The two constraints here are `out ==== 1 - in*inv` and `in*out === 0`. Converting this to multiplication and addition gates yields the four constraints `-1*in === in'`, `in'*inv === prod`, `1 + prod === out`, and `in*out === ans`, where `ans = 0`. To satisfy all constraints, the prover must provide a list of input, intermediate, and output signals that satisfy the constraints. For each gate, we create three wire vectors with the coefficients of each variable at the gate that represent how many of each signal we want to use to construct the constraint. These are encoded in coefficient vectors $\vec{l_i}, \vec{r_i}, \vec{o_i}$ so that $\vec{l_i}\vec{x} \cdot \vec{r_i}\vec{x} = \vec{o_i}$. We then combine all $\vec{l_i}, \vec{r_i}, \vec{o_i}$ and create the matrices $\mathcal{L}, \mathcal{R}, \mathcal{O}$ such that $\mathcal{L}\vec{x} \cdot \mathcal{R}\vec{x} - \mathcal{O}\vec{x} = \mathbf{0}$.

This is the R1CS. We convert from this matrix equation to QAP form by setting the coefficients of the $i$th constraint polynomials to the $i$th row of the corresponding matrix so that $R_i(X) = \mathcal{R}_i\vec{x}$. We repeat for all rows of the matrix to get a set of $k$ polynomial equations for a $k$-constraint R1CS.

We finally discuss intermediate representations. An algebraic intermediate representation (AIR) $P$ is a representation containing constraint polynomials such that execution trace $T$ on $P$ is valid if $\forall f_I(\cdot,\cdot) \in P$, we have $f_i(T \cdot j, T \cdot (j+1))$ for any $j \in [n]$. $P$ verifies some state transition function $f_i$ over adjacent elements in computation trace $T$. A preprocessed algebraic intermediate representation (PAIR) adds $t$ columns $c_i$ to the execution trance along with witness columns given by the prover. They let us introduce non-uniform

constraints and are sometimes known as selectors. For example, we can verify an addition operation on some columns with a PAIR and a multiplication operation on other columns while we can't do this with an AIR (though we could verify that both multiplication and addition took place with some modification using an AIR). A randomized AIR with preprocessing (RAP) basically ensures that some set of constraints is met everywhere but verifies this probabilistically by checking that the constraints are satisfied for some random pair of adjacent elements in the computation trace.

# PlonK

PlonK stands for Permutations of Lagrange-bases for Oecumenical Noninteractive arguemnts of Knowledge. It is a zk proof system that only requires that you perform some trusted setup procedure once. This procedure is "universal and updateable", so you only do it once. You can also perform it as a sequential multiparty procedure that is secure as long as at least one of the participants is honest. Another improvement is that PlonK really only relies on polynomial commitments, specifically Kate commitments, but we can swap this in for any other kind of commitment scheme.

As with any other proof system, PlonKs use a procedure that converts a problem of form "give me $X$ s. t. for program $P$, $P(X) = Y$" into a problem of form "give me a set of values that satisfy some set of equations". We do this by representing $P$ as a circuit with addition and multiplication gates and converting it to a system of equations with variables the inputs and outputs of the gates. In PlonK, we represent constraints using $L, R, O, M, C$ gates (respectively left, right, output, multiplication, constant) where we want to satisfy $Q_{L_i}a_i + Q_{R_i}b_i + Q_{O_i}c_i + Q_{M_i}a_ib_i + Q_{C_i} = 0$ where the $Q$ are constants that we multiply each gate $a_i, b_ic_i$ by and the $a_i$ are gate variables. This lets us express addition and multiplication operations succinctly with a vector of 5 values (5 entries for $Q$). We note that two forms of expressing a polynomial are evaluation form and coefficient form — the latter maintains the coefficients that multiplies each term of the polynomial and the former provides enough points (1 more point than the polynomial's degree) to interpolate and recover the polynomial using Lagrange interpolation. Typically, $C$ (the constant polynomial) is expressed as $Z(x)H(x)$ where $Z(x)$ is the range polynomial $\prod_{i=0}^{d-1}(x-i)$. Now that we have verified individual circuits, we need to verify that different values that should be the same are actually the same, which we deem copy constraints (like $a(5) = c(7)$). We do this through a coordinate pair accumulator, which is a $p(x)$ that takes two polynomials $X(x), Y(x)$ and defines $p(x+1)$ to be $p(x)(v_1 + X(x) + v_2Y(x))$ for random $v_1, v_2$. This accumulator $p(x)$ for $p(d)$ over evaluation domain $[d]$ is the result we want to analyze. It is crucial to note that for any permutation of $X(\cdot) = \pi([d])$, we will always get the same value for $p(d)$. To prove constraints between $a, b, c$ we use this accumulation process such that $a \to X_a(x) = x, b \to X_b(x) = m+x, c \to X_c(x) = 2m+x$ for some $m$. We sometimes use $\sigma_a(x), \cdots$ instead of $X'_a(x), \cdots$ where $X'_a(x)$ is $X_a(x)$ over random permutation $\pi([d])$. To check equality over these three polynomials, we check that $p_a(n)p_b(n)p_c(n) = p'_a(n)p'_b(n)p'_c(n)$. This is the product of accumulators, over which we assume equality over permutation.

# Overview of Proof Systems

We discuss different types of proof systems. Most modern proof systems (SNARK protocols) use algebraic holographic proofs (AHPs), which are polynomial IOPs where the verifier queries parts of the relation/equation being proven by oracle access. AHPs work with any polynomial commitment scheme, so different SNARK protocols mainly differ in the commitment scheme and the arithmetization (though even this is consistent among most proof systems). We compare the IOP approach against previous proof systems by discussing two proof systems that use R1CS: Pinocchio (which uses a linear probabilistically checkable proof alongside pairing-based cryptography) and Marlin and Spartan (which use algebraic holographic proofs alongside a PCS). We recall the arithmetization content we learned from previous lectures. We recall that we want to satisfy $L\vec{a} \otimes R\vec{a} - O\vec{a} = \vec{0}$ over all gates. We get polynomials $\{L_j\}, \{R_j\}, \{O_j\}$, which are the Lagrange interpolations of the points in each column of matrices $L, R, O$. Our QAP checks that $H(X) = {}^{P(X)}/_{T(X)}$ where $P(X) = L(X)R(X) - O(X)$ where $L(X) = \sum_j a_jL_j(X), \cdots$. An LPCP of length $m$ computes a linear function from $\mathbb{F}^m$ to $\mathbb{F}$ where for query $\vec{q}_i$, our answer is $q_i$. The divisibility check for our LPCP works by getting $\{L_j\}, \{R_j\}, \{O_j\}, T(X)$, then we sample a random $s \leftarrow_R \mathbb{F}$ and output the commitments to the

evaluations of the QAP's polynomials at $s$. The prover can take some $\vec{a}$ and multiply it by the commitments to get $\pi_L, \pi_R, \pi_O$, then compute $\pi_H = [H(s)]_1$ and output all of these proofs. Verification works by checking that $e(\pi_L, \pi_R) - e(\pi_O, 1_{\mathbb{G}}) \stackrel{?}{=} e([T(s)])1, \pi_H$, which is equivalent to checking that $L(s)R(s) - O(s) = T(s)H(s)$ over what is effectively a homomorphic encryption scheme. However, this doesn't ensure the prover uses the given polynomials. We enforce this using randomly-sampled $\alpha_L, \alpha_R, \alpha_O$. We then compute the commitments as the product of the polynomials and $\alpha_P$ to get $\{[L_j(s)]_1\}, \cdots$ The prover computes proofs on these $\alpha$-shifted polynomials, computes the regular proofs on the regular non-$\alpha$-shifted polynomials, then outputs both of these sets of proofs. The verifier then checks three things

$$e(\pi_L, [\alpha_L]_1) \stackrel{?}{=} e(\pi'_L, 1_{\mathbb{G}})$$

$$e(\pi_R, [\alpha_R]_1) \stackrel{?}{=} e(\pi'_R, 1_{\mathbb{G}})$$

$$e(\pi_O, [\alpha_O]_1) \stackrel{?}{=} e(\pi'_O, 1_{\mathbb{G}})$$

If we simply these equations, we will see that the left term of the RHS pairing is multiplied by the same $\alpha$ shift that the right term of the LHS is multiplied by and otherwise both sides are identical; therefore, both sides are equal. The prover does not know the $\alpha$ shifts, so they cannot falsify the proof unless they like copy a proof from a previous query. We finally must figure out how to constrain the prover to use the same coefficients $\vec{a}$ in computing each of the $\pi_L, \pi_R, \pi_O$. We do this by using $\beta, \gamma$ shifts. Intuitively, we prove by computing $P(X)$, which is the sum of polynomials, then create a proof of the $\beta$-shifted polynomials $\pi'_F$, compute $\pi_L, \pi_R \pi_O$, then output all proofs. Verification uses $\gamma, \beta$ along with the proofs and verify by checking that $e(\pi_L + \pi_R + \pi_O, \beta\gamma) \stackrel{?}{=} e(\pi'_F, \gamma)$.