Intro to Programmable Cryptography

Notes from a spring 2024 reading group

0xPARC

29 May 2024

Evan, I can now prove to you that I have a message M such that $\mathrm{sha}(M) = \mathrm{0xa91af3ac}\ldots$*, without revealing M. But not just for SHA. I can do this for any function you want.*

— gubsheep introducing progcrypto to Evan for the first time

Table of contents

§1 Frontmatter

These are compiled lecture notes from a reading group hosted by the 0xPARC Foundation. It's not meant to be a exhaustive textbook or mathematically complete reference, but rather a introduction to the general landscape and ideas for newcomers.

We assume a bit of general undergraduate math background, but not too much. (For example, I'll just use the word "abelian group" freely, and the reader is assumed to know modular arithmetic.) We don't assume specialized knowledge like elliptic curve magic.

§1.1 Characters

- **Alice** and **Bob** reprise their usual roles as generic characters.
- **Peggy** and **Victor** play the roles of *Prover* and *Verifier* for protocols in which Peggy wishes to prove something to Victor.
- **Trent** is a trusted administrator or arbiter, for protocols in which a trusted setup is required. (In real life, Trent is often a group of people performing a multi-party computation, such that as long as at least one of them is honest, the trusted setup will work.)

§1.2 Notation and conventions

- Throughout these notes, E will always denote an elliptic curve over some finite field $\mathbb{F}$ (whose order is known for calculation but otherwise irrelevant).
- If we were being pedantic, we might be careful to distinguish the elliptic curve E from its set of $\mathbb{F}$-points $E(\mathbb{F})$. But to ease notation, we simply use E interchangeably with $E(\mathbb{F})$.
- Hence, the notation "$g \in E$" means "g is a point of $E(\mathbb{F})$". Elements of the curve E are denoted by lowercase Roman letters; $g \in E$ and $h \in E$ are especially common.
- We always use additive notation for the group law on E: given $g \in E$ and $h \in E$ we have $g + h \in E$.
- $\mathbb{F}_q$ denotes the finite field of order q. In these notes, q is usually a globally known large prime, often $q \approx 2^{256}$.
- $\mathbb{F}_q[X]$ denotes the ring of univariate polynomials with coefficients in $\mathbb{F}_q$ in a single formal variable X. More generally, $\mathbb{F}_q[T_1, \ldots, T_n]$ denotes the ring of polynomials in the n formal variables T_1, ..., T_n. We'll prefer capital Roman letters for both polynomials and formal variables.
- $\mathbb{N} = \{1, 2, \ldots, \}$ denotes the set of *positive* integers, while $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$ is the set of all integers. We prefer the Roman letters m, n, N for integers.
- We let $\text{hash}()$ denote your favorite one-way hash function, such as SHA-256. For us, it'll take in any number of arguments of any type (you should imagine they are coerced into strings) and output a single number in $\mathbb{F}_q$. That is,

$$\text{hash} : \text{any number of inputs} \to \mathbb{F}_q.$$

§1.3 Acknowledgments

Authors

TODO: Vitalik, Darken

§2 Introduction

§2.1 What is programmable cryptography?

Cryptography is everywhere now and needs no introduction. "Programmable cryptography" is a term coined by 0xPARC for a second generation of cryptographic primitives that have arisen in the last 15 or so years.

To be concrete, let's consider two examples of what protocols designed by classical cryptography can achieve:

- *Proofs*. An example of this is digital signature algorithms like RSA, where Alice can do some protocol to prove to Bob that a message was sent by her. A more complicated example might be a group signature scheme, allowing one member of a group to sign a message on behalf of a group.
- *Hiding inputs*: for example, consider Yao's millionaire problem, where Alice and Bob wants to know which of them has more money without learning the actual incomes.

Classically, first-generation cryptography relied on coming up for a protocol for solving given problems or computing certain functions. The goal of the second-generation "programmable cryptography" can then be described as:

> *We want to devise cryptographic primitives that could be programmed to work on **arbitrary** problems and functions, rather than designing protocols on a per-problem or per-function basis.*

To draw an analogy, it's sort of like going from older single-purpose hardware, like a digital alarm clock or thermostat, to having a general-purpose device like a smartphone which can do any computation so long as someone writes code for it.

The quote on the title page ("I have a message M such that $\text{sha}(M) = \text{0x91af3ac...}$") is a concrete example; the hash function SHA is a particular set of arbitrary instructions, yet programmable cryptography promises that such a proof can be made using a general compiler rather than inventing an algorithm specific to SHA256.

TODO: Brian's image of an alarm clock and a computer chip

§2.2 Ideas in programmable cryptography

These notes focus on the following specific topics.

§2.2.1 The zkSNARK: proofs of general problems

The **zkSNARK**, first described in 2012, was the first type of primitive that arguably falls into the "programmable cryptography" umbrella. It provides a way to produce proofs of *arbitrary* problem statements, at least once encoded as a system of equations in a certain way. The name stands for:

- **Zero-knowledge**: a person reading the proof doesn't learn anything about the solution besides that it's correct.
- **Succinct**: the proof length is short (actually constant length).
- **Non-interactive**: the protocol is not interactive.
- **Argument**: technically not a "proof," but we won't worry about the difference.
- **of Knowledge**: the proof doesn't just show the system of equations has a solution; it also shows that the prover knows one.

So, you can think of these as generalizing something like a group signature scheme to authenticating any sort of transaction:

- A normal signature scheme is a (zero-knowledge, succinct, non-interactive) proof that "I know Alice's private key".
- A group signature scheme can be construed as a succinct proof that "I know one of Alice, Bob, or Charlie's private keys".
- But you could also use a zkSNARK to prove a statement like "I know a message M such that $\text{hash}(M) = 0\text{x}91\text{af}3\text{ac}\ldots$", of course without revealing M or anything about M.
- ... Or really any arbitrarily complicated statement.

TODO: gubsheep's slide had a funny example with emoji, link it

These notes focus on two constructions, PLONK (Section 7) and Groth16 (Section 8).

§2.2.2 Multi-party computation (MPC)

A **multi-party computation**, in which $n \geq 2$ people want to jointly compute some known function

$$F(x_1, ..., x_n)$$

where the ith person only knows the input x_i and does not learn the other inputs.

For example, we saw earlier Yao's millionaire problem — Alice and Bob want to know who has a higher income without revealing the incomes themselves. This is the case where $n = 2$, $F = \max$, and x_i is the i'th person's income.

Multi-party computation makes a promise that we'll be able to do this for *any* function F as long as we implement it in code.

§2.2.3 Fully homomorphic encryption (FHE)

In **fully homomorphic encryption**, one person encrypts some data x, and then anybody can perform arbitrary operations on the encrypted data x without being able to read x.

For example, imagine you have some private text that you want to translate into another language. You encrypt the text and feed it to your favorite FHE machine translation server. You decrypt the server's output and get the translation. The server only ever sees encrypted text, so the server learns nothing about the text you translated.

§2.3 Where these fit together

ZkSNARKS, MPC, and FHE are just some of a huge zoo of cryptographic primitives, from the elementary (public-key cryptography) to the impossibly powerful (indistinguishability obfuscation). There are protocols for zkSNARKS, MPC and FHE; they are very slow, but they can be implemented and used in practice.

This whole field is an active area of research. On the one hand: Can we make existing tools (zkSNARKS, etc.) more efficient? For example, the cost of doing a computation in zero knowledge is currently about 10^6 times the cost of doing the computation directly. Can we bring that number down? On the other hand: What other cryptographic games can we play to develop new sorts of programmable cryptography functionality?

At 0xPARC, we see this as a door to a new world. What sort of systems can we build on top of programmable cryptography?

TODO: Import Brian's tree. Talk about reduction? Evan, take a look at the flavor text, idk if I like it - Aard

§2.4 What's all the fuss about zero-knowledge anyhow?

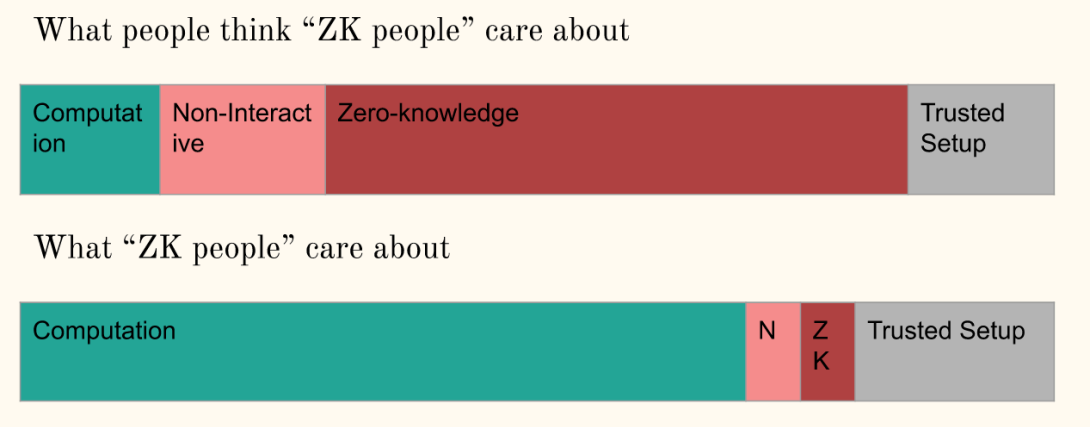

Figure 1: Expectations vs. reality.

TODO: Aard suggests deleting the figure, it's cute but Aard isn't sure about the message

When we think about how to use programmable cryptography we need to be creative. As an example, what can you do with a zkSNARK?

One answer: You can prove that you have a solution to a system of equations. Sounds pretty boring, unless you're an algebra student.

Slightly better answer: You can prove that you have executed a program correctly, revealing some or all of the inputs and outputs, as you please. For example: You know a messame M such that $\text{sha}(M) = 0\text{xa91af3ac...}$, but you don't want to reveal M. Or: You only want to reveal the first 30 bytes of M. Or: You know a message M, and a digital signature proving that M was signed by [trusted authority], such that a certain neural network, run on the input M, outputs "Good."

One recent application along these lines is TLSNotary. TLSNotary lets you certify a transcript of communications with a server in a privacy-preserving way: you only reveal the parts you want to.

zkSNARK constructions

This part covers two constructions of the zkSNARK, the **PLONK** and **Groth16** constructions. Despite being fairly modern constructions, these are arguably simpler and more informative to learn about than the PCP construction that preceded them (which is covered in Section 15).

The dependency chart of this chapter goes as follows:

- Section 3 describes the discrete logarithm problem on an elliptic curve, which provides a basis for everything afterwards.
- Section 5 and Section 6 give two different **polynomial commitment schemes**, which allow a prover Peggy to
  - commit to some polynomial $P(X) \in \mathbb{F}_q[X]$ ahead of time,
  - and then **open the commitment** at any input $z \in \mathbb{F}_q$ while not revealing P itself.

  The KZG scheme from Section 5 is quite simple and elegant but requires a trusted setup. In contrast, IPA from Section 6 has fewer assumptions and is more versatile, but it's slower and more complicated.
- Regardless of whether KZG/IPA scheme is used, we then show two constructions of a zkSNARK. In Section 7 we construct PLONK; in Section 8 we construct Groth16.

§3 Elliptic curves

In the public-key cryptography system RSA, one key assumption is that there is no efficient method to factor large semiprimes. RSA can be thought of as working with the abelian group $(\mathbb{Z}/N\mathbb{Z})^\times$, where N is the product of two large primes. This setup, while it does work, is brittle in some ways (for example, every person needs to pick a different N for RSA).

In many modern protocols, one replaces $(\mathbb{Z}/N\mathbb{Z})^\times$ with a so-called *elliptic curve* E. The assumption "factoring is hard" is then replaced by a new one, that the discrete logarithm problem is hard.

This chapter sets up the math behind this elliptic curve E. The roadmap goes roughly as follows:

- In Section 3.1 we will describe one standard elliptic curves E, the BN254 curve, that will be used in these notes.
- In Section 3.2 we describe the discrete logarithm problem: that for $g \in E$ and $n \in \mathbb{F}_q$, one cannot recover n from $n \cdot g$. This is labeled as Assumption 3.5 in these notes.
- As an example, in Section 3.4 we describe how Assumption 3.5 can be used to construct a signature scheme, namely EdDSA. This idea will later grow up to be the KZG commitment scheme in Section 5.
- As another example, in Section 3.5 we describe how Assumption 3.5 can be used to create a commitment scheme for vectors: the Pedersen commitment. This idea will later grow up to be the IPA commitment scheme in Section 6.

§3.1 The BN254 curve

Rather than set up a general definition of elliptic curve, for these notes we will be satisfied to describe one specific elliptic curve that could be used for all the protocols we describe later. The curve we choose for these notes is the **BN254 curve**.

§3.1.1 The set of points

The BN254 specification fixes a specific[1] large prime $p \approx 2^{254}$ (and a second large prime $q \approx 2^{254}$ that we define later) which has been specifically engineered to have certain properties (see e.g. https://hackmd.io/@jpw/bn254). The name BN stands for Barreto-Naehrig, two mathematicians who proposed a family of such curves in 2006.

Definition 3.1: The **BN254 curve** is the elliptic curve

$$E(\mathbb{F}_p) : Y^2 = X^3 + 3 \tag{1}$$

defined over $\mathbb{F}_p$, where $p \approx 2^{254}$ is the prime from the BN254 specification.

So each point on $E(\mathbb{F}_p)$ is an ordered pair $(X, Y) \in \mathbb{F}_p^2$ satisfying Equation 1. Okay, actually, that's a white lie: conventionally, there is one additional point $O = (0, \infty)$ called the "point at infinity" added in (whose purpose we describe in the next section).

The constants p and q are contrived so that the following holds:

[1] If you must know, the values in the specification are given exactly by

$$\begin{aligned} x &:= 4965661367192848881 \\ p &:= 36x^4 + 36x^3 + 24x^2 + 6x + 1 \\ &= 21888242871839275222246405745257275088696311157297823662689037894645226208583 \\ q &:= 36x^4 + 3x^3 + 18x^2 + 6x + 1 \\ &= 21888242871839275222246405745257275088548364400416034343698204186575808495617. \end{aligned}$$

Theorem 3.2 (BN254 has prime order): Let E be the BN254 curve. The number of points in $E(\mathbb{F}_p)$, including the point at infinity O, is a prime $q \approx 2^{254}$.

Definition 3.3: This prime $q \approx 2^{254}$ is affectionately called the **Baby Jubjub prime** (a reference to The Hunting of the Snark). It will usually be denoted by q in these notes.

§3.1.2 The group law

So at this point, we have a bag of q points denoted $E(\mathbb{F}_p)$. However, right now it only has the structure of a set.

The beauty of elliptic curves is that it's possible to define an *addition* operation on the curve; this is called the group law on elliptic curve. This addition will make $E(\mathbb{F}_p)$ into an abelian group whose identity element is O.

This group law involves some kind of heavy algebra. It's not important to understand exactly how it works. All you really need to take away from this section is that there is some group law, and we can program a computer to compute it.

So, let's get started. The equation of E is cubic – the highest-degree terms have degree 3. This means that (in general) if you take a line $y = mx + b$ and intersect it with E, the line will meet E in exactly three points. The basic idea behind the group law is: If P, Q, R are the three intersection points of a line (any line) with the curve E, then the group-law addition of the three points is

$$P + Q + R = O.$$

(You might be wondering how we can do geometry when the coordinates x and y are in a finite field. It turns out that all the geometric operations we're describing – like finding the intersection of a curve with a line – can be translated into algebra. And then you just do the algebra in your finite field. But we'll come back to this.)

Why three points? Algebraically, if you take the equations $Y^2 = X^3 + 3$ and $Y = mX + b$ and try to solve them, you get

$$(mX + b)^2 = X^3 + 3,$$

which is a degree-3 polynomial in X, so it has (at most) 3 roots. And in fact if it has 2 roots, it's guaranteed to have a third (because you can factor out the first two roots, and then you're left with a linear factor).

OK, so given two points P and Q, how do we find their sum $P + Q$? Well, we can draw the line through the two points. That line – like any line – will intersect E in three points: P, Q, and a third point R. Now since $P + Q + R = 0$, we know that

$$-R = P + Q.$$

So now the question is just: how to find $-R$? Well, it turns out that if $R = (x_R, y_R)$, then

$$-R = (x_R, -y_R).$$

Why is this? Well, if you take the vertical line $X = x_R$, and try to intersect it with the curve, it looks like there are only two intersection points. After all, we're solving

$$Y^2 = x_R^3 + 3,$$

and since x_R is fixed now, this equation is quadratic. The two roots are $Y = pmy_R$.

OK, there are only two intersection points, but we say that the third intersection point is "the point at infinity" O. (The reason for this lies in projective geometry, but we won't get into it.) So the group law here tells us

$$(x_R, y_R) + (x_R, -y_R) + O = O.$$

And since O is the identity, we get

$$-R = (x_R, -y_R).$$

So:

- Given a point $P = (x_P, y_P)$, its negative is just $-P = (x, -y)$.
- To add two points P and Q, compute the line through the two points, let R be the third intersection of that line with E, and set

$$P + Q = -R.$$

I just described the group law as a geometric thing, but there are algebraic formulas to compute it as well. They are kind of a mess, but here goes.

If $P = (x_P, y_P)$ and $Q = \left(x_Q, y_Q\right)$, then the line between the two points is $Y = mX + b$, where

$$m = \frac{y_Q - y_P}{x_Q - x_P}$$

and

$$b = y_P - mx_P.$$

The third intersection is $R = (x_R, y_R)$, where

$$x_R = m^2 - x_P - x_Q$$

and

$$y_R = mx_R + b.$$

There are separate formulas to deal with various special cases (if $P = Q$, you need to compute the tangent line to E at P, for example), but we won't get into it.

TODO: Is there a way to... put the above into a box environment or something, so the reader can skip it easily?

In summary we have endowed the set of points $E\left(\mathbb{F}_p\right)$ with the additional structure of an abelian group, which happens to have exactly q elements. However, an abelian group with prime order is necessarily cyclic. In other words:

Theorem 3.4 (The group BN254 is isomorphic to $\mathbb{F}_q$)**:** Let E be the BN254 curve. We have the isomorphism of abelian groups

$$E(\mathbb{F}_p) \cong \mathbb{Z}/q\mathbb{Z}.$$

In these notes, this isomorphism will basically be a standing assumption; so moving forward, as described in Section 1.2 we'll abuse notation slightly and just write E instead of $E(\mathbb{F}_p)$.

§3.1.3 We treat $\mathbb{F}_q$ as the field of scalars henceforth

Consequently — and this is important — **one should actually think of $\mathbb{F}_q$ as the base field for all our cryptographic primitives** (despite the fact that the coordinates of our points are in $\mathbb{F}_p$).

Whenever we talk about protocols, and there are any sorts of "numbers" or "scalars" in the protocol, **these scalars are always going to be elements of $\mathbb{F}_q$**. Since $q \approx 2^{254}$, that means we are doing something like 256-bit integer arithmetic. This is why the baby Jubjub prime q gets a special name, while the prime p is unnamed and doesn't get any screen-time later.

§3.2 Discrete logarithm is hard

For our systems to be useful, rather than relying on factoring, we will rely on the so-called **discrete logarithm** problem.

Assumption 3.5 (Discrete logarithm problem)**:** Let E be the BN254 curve (or another standardized curve). Given arbitrary nonzero $g, g' \in E$, it's hard to find an integer n such that $n \cdot g = g'$.

In other words, if one only sees $g \in E$ and $n \cdot g \in E$, one cannot find n. For cryptography, we generally assume g has order q, so we will talk about $n \in \mathbb{N}$ and $n \in \mathbb{F}_q$ interchangeably. In other words, n will generally be thought of as being up to 2^{254} in size.

Remark 3.6 (The name "discrete log")**:** This problem is called discrete log because if one used multiplicative notation like in RSA, it looks like solving $g^n = g'$ instead. We will never use this multiplicative notation in these notes.

On the other hand, given $g \in E$, one can compute $n \cdot g$ in just $O(\log n)$ operations, by repeated squaring. For example, to compute $400g$, one only needs to do 10 additions, rather than 400: one starts with

$$\begin{aligned} 2g &= g + g \\ 4g &= 2g + 2g \\ 8g &= 4g + 4g \\ 16g &= 8g + 8g \\ 32g &= 16g + 16g \\ 64g &= 32g + 32g \\ 128g &= 64g + 64g \\ 256g &= 128g + 128g \end{aligned}$$

and then computes

$$400g = 256g + 128g + 16g.$$

Because we think of n as up to $q \approx 2^{254}$-ish in size, we consider $O(\log n)$ operations like this to be quite tolerable.

§3.3 Curves other than BN254

We comment briefly on how the previous two sections adapt to other curves, although readers could get away with always assuming E is BN254 if they prefer.

In general, we could have chosen for E any equation of the form $Y^2 = X^3 + aX + b$ and chosen any prime $p \geq 5$ such that a nondegeneracy constraint $4a^3 + 27b^2 \not\equiv 0 \bmod p$ holds. In such a situation, $E(\mathbb{F}_p)$ will indeed be an abelian group once the identity element $O = (0, \infty)$ is added in.

How large is $E(\mathbb{F}_p)$? There is a theorem called Hasse's theorem that states the number of points in $E(\mathbb{F}_p)$ is between $p + 1 - 2\sqrt{p}$ and $p + 1 + 2\sqrt{p}$. But there is no promise that $E(\mathbb{F}_p)$ will be *prime*; consequently, it may not be a cyclic group either. So among many other considerations, the choice of constants in BN254 is engineered to get a prime order.

There are other curves used in practice for which $E(\mathbb{F}_p)$ is not a prime, but rather a small multiple of a prime. The popular Curve25519 is such a curve that is also believed to satisfy Assumption 3.5. Curve25519 is defined as $Y^2 = X^3 + 486662X^2 + X$ over $\mathbb{F}_p$ for the prime $p := 2^{255} - 19$. Its order is actually 8 times a large prime $q' := 2^{252} + 27742317777372353535851937790883648493$. In that case, to generate a random point on Curve25519 with order q', one will usually take a random point in it and multiply it by 8.

BN254 is also engineered to have a property called *pairing-friendly*, which is defined in Section 4.2 when we need it later. (In contrast, Curve25519 does not have this property.)

§3.4 Example application: EdDSA signature scheme

We'll show how Assumption 3.5 can be used to construct a signature scheme that replaces RSA. This scheme is called EdDSA, and it's used quite frequently (e.g. in OpenSSH and GnuPG). One advantage it has over RSA is that its key size is much smaller: both the public and private key are 256 bits. (In contrast, RSA needs 2048-4096 bit keys for comparable security.)

§3.4.1 The notation $[n]$

Let E be an elliptic curve and let $g \in E$ be a fixed point on it of prime order $q \approx 2^{254}$. For $n \in \mathbb{Z}$ (equivalently $n \in \mathbb{F}_q$) we define

$$[n] := n \cdot g \in E.$$

The hardness of discrete logarithm means that, given $[n]$, we cannot get n. You can almost think of the notation as an "armor" on the integer n: it conceals the integer, but still allows us to perform (armored) addition:

$$[a+b] = [a] + [b].$$

In other words, $n \mapsto [n]$ viewed as a map $\mathbb{F}_q \to E$ is $\mathbb{F}_q$-linear.

§3.4.2 Signature scheme

So now suppose Alice wants to set up a signature scheme.

Algorithm 3.7 (EdDSA public and secret key):
1. Alice picks a random integer $d \in \mathbb{F}_q$ as her **secret key**.
2. Alice publishes $[d] \in E$ as her **public key**.

Now suppose Alice wants to prove to Bob that she approves the message msg, given her published public key $[d]$.

Algorithm 3.8 (EdDSA signature generation): Suppose Alice wants to sign a message msg.

1. Alice picks a random scalar $r \in \mathbb{F}_q$ (keeping this secret) and publishes $[r] \in E$.
2. Alice generates a number $n \in \mathbb{F}_q$ by hashing msg with all public information, say
$$n := \text{hash}([r], \mathsf{msg}, [d]).$$
3. Alice publishes the integer
$$s := (r + dn) \bmod q.$$

In other words, the signature is the ordered pair $([r], s)$.

Algorithm 3.9 (EdDSA signature generation): For Bob to verify a signature $([r], s)$ for msg:

1. Bob recomputes n (by also performing the hash) and computes $[s] \in E$.
2. Bob verifies that $[r] + n \cdot [d] = [s]$.

An adversary cannot forge the signature even if they know r and n. Indeed, such an adversary can compute what the point $[s] = [r] + n[d]$ should be, but without knowledge of d they cannot get the integer s, due to Assumption 3.5.

The number r is called a **blinding factor** because its use prevents Bob from stealing Alice's secret key d from the published s. It's therefore imperative that r isn't known to Bob nor reused between signatures, and so on. One way to do this would be to pick $r = \text{hash}(d, \mathsf{msg})$; this has the bonus that it's deterministic as a function of the message and signer.

In Section 5 we will use ideas quite similar to this to build the KZG commitment scheme.

§3.5 Example application: Pedersen commitments

A multivariable corollary of Assumption 3.5 is that if $g_1, ..., g_n \in E$ are a bunch of randomly chosen points of E with order q, then it's computationally infeasible to find $(a_1, ..., a_n) \neq (b_1, ..., b_n) \in \mathbb{F}_q^n$ such that

$$a_1 g_1 + ... + a_n g_n = b_1 g_1 + ... + b_n g_n.$$

Indeed, even if one fixes any choice of $2n - 1$ of the $2n$ coefficients above, one cannot find the last coefficient.

Definition 3.10: In these notes, if there's a globally known elliptic curve E and points $g_1, ..., g_n$ have order q and no known nontrivial linear dependencies between them, we'll say they're a **computational basis over** $\mathbb{F}_q$.

Remark 3.11: This may horrify pure mathematicians because we're pretending the map

$$\mathbb{F}_q^n \to \mathbb{F}_q \quad \text{by } (a_1, ..., a_n) \mapsto \sum_1^n a_i g_i$$

is injective, even though the domain is an n-dimensional $\mathbb{F}_q$-vector space and the codomain is one-dimensional. This can feel weird because our instincts from linear algebra in pure math are wrong now. This map, while not injective in theory, ends up being injective *in practice* (because we can't find collisions). And this is a critical standing assumption for this entire framework!

This injectivity gives us a sort of hash function on vectors (with "linearly independent" now being phrased as "we can't find a collision"). To spell this out:

Definition 3.12: Let $g_1, ..., g_n \in E$ be a computational basis over $\mathbb{F}_q$. Given a vector

$$\vec{a} = \langle a_1, ..., a_n \rangle \in \mathbb{F}_q^n$$

of scalars, the group element

$$\sum a_i g_i = a_1 g_1 + ... + a_n g_n \in E$$

is called the **Pedersen commitment** of our vector $\vec{a}$.

The Pedersen commitment is thus a sort of hash function: given the group element above, one cannot recover any of the a_i (even when $n = 1$); but given the entire vector $\vec{a}$ one can compute the Pedersen commitment easily.

Pedersen commitments aren't used in the KZG scheme covered in Section 5, but they feature extensively in the IPA scheme covered in Section 6.

§4 Bilinear pairings on elliptic curves

The map $[\bullet] : \mathbb{F}_q \to E$ is linear, meaning that $[a+b] = [a] + [b]$, and $[na] = n[a]$. But as written we can't do "armored multiplication":

Claim 4.1: As far as we know, given $[a]$ and $[b]$, one cannot compute $[ab]$.

On the other hand, it *does* turn out that we know a way to *verify* a claimed answer on certain curves. That is:

Proposition 4.2: On the curve BN254, given three points $[a]$, $[b]$, and $[c]$ on the curve, one can verify whether $ab = c$.

The technique needed is that one wants to construct a nondegenerate bilinear function

$$\text{pair} : E \times E \to \mathbb{Z}/N\mathbb{Z}$$

for some large integer N. I think this should be called a **bilinear pairing**, but for some reason everyone justs says **pairing** instead. A curve is called **pairing-friendly** if this pairing can be computed reasonably (e.g. BN254 is pairing-friendly, but Curve25519 is not).

This construction actually uses some really deep graduate-level number theory (in contrast, all the math in Section 3 is within an undergraduate curriculum) that is well beyond the scope of these lecture notes. Fortunately, we won't need the details of how it works; but we'll comment briefly in Section 4.2 on what curves it can be done on. And this pairing algorithm needs to be worked out just once for the curve E; and then anyone in the world can use the published curve for their protocol.

Going a little more generally, the four-number equation

$$\text{pair}([m], [n]) = \text{pair}([m'], [n'])$$

will be true whenever $mn = m'n'$, because both sides will equal $mn\,\text{pair}([1], [1])$. So this gives us a way to *verify* two-by-two multiplication.

Remark 4.3: The last sentence is worth bearing in mind: in all the protocols we'll see, the pairing is only used by the *verifier* Victor, never by the prover Peggy.

Remark 4.4 (We don't know how to do multilinear pairings): On the other hand, we currently don't seem to know a good way to do *multilinear* pairings. For example, we don't know a good trilinear map $E \times E \times E \to \mathbb{Z}/N\mathbb{Z}$ that would allow us to compare $[abc]$, $[a]$, $[b]$, $[c]$ (without already knowing one of $[ab]$, $[bc]$, $[ca]$).

§4.1 Verifying more complicated claims

Example 4.5: Suppose Peggy wants to convince Victor that $y = x^3 + 2$, where Peggy has sent Victor elliptic curve points [x] and [y]. To do this, Peggy additionally sends to Victor $[x^2]$ and $[x^3]$.

Given $[x]$, $[x^2]$, $[x^3]$, and $[y]$, Victor verifies that:
- $\text{pair}([x^2], 1) = \text{pair}([x], [x])$
- $\text{pair}([x^3], 1) = \text{pair}([x^2], [x])$
- $[y] = [x^3] + 2[1]$.

The process of verifing this sort of identity is quite general: The prover sends intermediate values as needed so that the verifier can verify the claim using only pairings and linearity.

§4.2 So which curves are pairing-friendly?

If we chose E to be BN254, the following property holds:

Proposition 4.6: For (p, q) as in BN254, the smallest integer k such that q divides $p^k - 1$ is $k = 12$.

This integer k is called the **embedding degree**. This section is an aside explaining how the embedding degree affects pairing.

The pairing function $\text{pair}(a, b)$ takes as input two points $a, b \in E$ on the elliptic curve, and spits out a value $\text{pair}(a, b) \in \mathbb{F}^*_{\{p^k\}}$ – in other words, a nonzero element of the finite field of order p^k (where k is the embedding degree we just defined). In fact, this element will always be a qth root of unity in $\mathbb{F}_{\{p^k\}}$, and it will satisfy $\text{pair}([m], [n]) = \zeta^{\{mn\}}$, where $z\eta$ is some fixed qth root of unity. The construction of the pairing is based on the Weil pairing. in algebraic geometry. How to compute these pairings is well beyond the scope of these notes; the raw definition is quite abstract, and a lot of work has gone into computing the pairings efficiently. (For more details, see these notes.)

The difficulty of computing these pairings is determined by the size of k: the values $\text{pair}(a, b)$ will be elements of a field of size p^k, so they will require 256 bits even to store. For a curve to be "pairing-friendly" – in order to be able to do pairing-based cryptography on it – we need the value of k to be pretty small.

§5 Kate-Zaverucha-Goldberg (KZG) commitments

§5.1 Pitch: KZG lets you commit a polynomial and reveal individual values

The goal of the KZG commitment schemes is to have the following API:

- Peggy has a secret polynomial $P(X) \in \mathbb{F}_q[X]$.
- Peggy sends a short "commitment" to the polynomial (like a hash).
- This commitment should have the additional property that Peggy should be able to "open" the commitment at any $z \in \mathbb{F}_q$. Specifically:
  - Victor has an input $z \in \mathbb{F}_q$ and wants to know $P(z)$.
  - Peggy knows P so she can compute $P(z)$; she sends the resulting number $y = P(z)$ to Victor.
  - Peggy can then send a short "proof" convincing Victor that y is the correct value, without having to reveal P.

The KZG commitment scheme is amazingly efficient because both the commitment and proof lengths are a single point on E, encodable in 256 bits.

§5.2 Elliptic curve setup done once

The good news is that this can be done just once, period. After that, anyone in the world can use the published data to run this protocol.

For concreteness, E will be the BN256 curve and g a fixed generator.

§5.2.1 The notation $[n]$

We retain the notation $[n] := n \cdot g \in E$ defined in Section 3.4.1.

§5.2.2 Trusted calculation

To set up the KZG commitment scheme, a trusted party needs to pick a secret scalar $s \in \mathbb{F}_q$ and publishes

$$[s^0], [s^1], ..., [s^M]$$

for some large M, the maximum degree of a polynomial the scheme needs to support. This means anyone can evaluate $[P(s)]$ for any given polynomial P of degree up to M. (For example, $[s^2 + 8s + 6] = [s^2] + 8[s] + 6[1]$.) Meanwhile, the secret scalar s is never revealed to anyone.

This only needs to be done by a trusted party once for the curve E. Then anyone in the world can use the resulting sequence for KZG commitments.

Remark 5.1: The trusted party has to delete s after the calculation. If anybody knows the value of s, the protocol will be insecure. The trusted party will only publish $[s^0] = [1], [s^1], ..., [s^M]$. Given these published values, it is (probably) extremely hard to recover s – this is a case of the discrete logarithm problem.

You can make the protocol somewhat more secure by involving several different trusted parties. The first party chooses a random s_1, computes $[s_1^0], ..., [s_1^M]$, and then discards s_1. The second party chooses s_2 and computes $\left[(s_1 s_2)^0\right], ..., \left[(s_1 s_2)^M\right]$. And so forth. In the end, the value s will be the product of the secrets s_i chosen by the i parties... so the only way they can break secrecy is if all the "trusted parties" collaborate.

§5.3 The KZG commitment scheme

Peggy has a polynomial $P(X) \in \mathbb{F}_p[X]$. To commit to it:

Algorithm 5.2 (Creating a KZG commitment):
1. Peggy computes and publishes $[P(s)]$.

This computation is possible as $[s^i]$ are globally known.

Now consider an input $z \in \mathbb{F}_p$; Victor wants to know the value of $P(z)$. If Peggy wishes to convince Victor that $P(z) = y$, then:

Algorithm 5.3 (Opening a KZG commitment):
1. Peggy does polynomial division to compute $Q(X) \in \mathbb{F}_q[X]$ such that
$$P(X) - y = (X - z)Q(X).$$
2. Peggy computes and sends Victor $[Q(s)]$, which again she can compute from the globally known $[s^i]$.
3. Victor verifies by checking
$$\text{pair}([Q(s)], [s] - [z]) = \text{pair}([P(s)] - [y], [1]) \tag{2}$$
and accepts if and only if Equation 2 is true.

If Peggy is truthful, then Equation 2 will certainly check out.

If $y \neq P(z)$, then Peggy can't do the polynomial long division described above. So to cheat Victor, she needs to otherwise find an element

$$\frac{1}{s - x}([P(s)] - [y]) \in E.$$

Since s is a secret nobody knows, there isn't any known way to do this.

§5.4 Multi-openings

To reveal P at a single value z, we did polynomial division to divide $P(X)$ by $X - z$. But there's no reason we have to restrict ourselves to linear polynomials; this would work equally well with higher-degree polynomials, while still using only a single 256-bit for the proof.

For example, suppose Peggy wanted to prove that $P(1) = 100$, $P(2) = 400$, ..., $P(9) = 8100$. Then she could do polynomial long division to get a polynomial Q of degree $\deg(P) - 9$ such that

$$P(X) - 100X^2 = (T - 1)(T - 2)...(T - 9) \cdot Q(T).$$

Then Peggy sends $[Q(s)]$ as her proof, and the verification equation is that

$$\text{pair}([Q(s)], [(s - 1)(s - 2)...(s - 9)]) = \text{pair}([P(s)] - 100[s^2], [1]).$$

The full generality just replaces the $100T^2$ with the polynomial obtained from Lagrange interpolation (there is a unique such polynomial f of degree $n - 1$). To spell this out, suppose Peggy wishes to prove to Victor that $P(z_i) = y_i$ for $1 \leq i \leq n$.

Algorithm 5.4 (Opening a KZG commitment at n values):

1. By Lagrange interpolation, both parties agree on a polynomial $f(X)$ such that $f(z_i) = y_i$.
2. Peggy does polynomial long division to get $Q(X)$ such that
$$P(X) - f(X) = (X - z_1)(X - z_2)...(X - z_n) \cdot Q(X).$$
3. Peggy sends the single element $[Q(s)]$ as her proof.
4. Victor verifies
$$\text{pair}([Q(s)], [(s - z_1)(s - z_2)...(s - z_n)]) = \text{pair}([P(s)] - [f(s)], [1]).$$

So one can even open the polynomial P at 1000 points with a single 256-bit proof. The verification runtime is a single pairing plus however long it takes to compute the Lagrange interpolation f.

§6 The inner product argument (IPA)

IPA is actually a lot more general than polynomial commitment. So the roadmap is as follows:

- In Section 6.1, describe the API of the inner product argument.
- In Section 6.2 and Section 6.3, describe the protocol.
- In Section 6.4, show two other applications of IPA, as a side demo.
- In Section 6.5, show how IPA can be used as a polynomial commitment scheme.

Throughout this section, E is defined as in Section 1.2, and there are fixed globally known generators $g_1, ..., g_n, h_1, ..., h_n, u \in E$ which are a computational basis (Definition 3.10).

§6.1 Pitch: IPA allows verifying $c = \sum a_i b_i$ without revealing a_i, b_i, c

Consider Pedersen commitments of the form

$$a_1 g_1 + ... + a_n g_n + b_1 h_1 + ... + b_n h_n + cu \in E$$

where $a_1, ..., a_n, b_1, ..., b_n, c \in \mathbb{F}_p$.

Definition 6.1: Let's say that an element

$$v = a_1 g_1 + ... + a_n g_n + b_1 h_1 + ... + b_n h_n + cu \in E$$

is **good** (with respect to the basis $\langle g_1, ..., g_n, h_1, ..., h_n, u \rangle$) if $\sum_1^n a_i b_i = c$.

The Inner Product Argument (IPA) is a protocol that kind of resembles the older Sum-Check (described in Section 14) in spirit: Peggy and Victor will do a series of interactions which allow Peggy to prove to Victor that v is good. And Peggy will be able to do this without having to reveal all of the a_i's, b_i's, and c.

(I think we missed a chance to call this "Inner Product Interactive Proof Inductive Protocol" or something cute like this, but I'm late to the party.)

§6.2 The interactive induction of IPA

The way IPA is done is by induction: one reduces verifying a vector for n is good (hence $2n + 1$ length) by verifying a vector for $\frac{n}{2}$ is good (of length $n + 1$).

To see how you might think of the idea on your own, check out this 0xPARC blog post.

To illustrate the induction, we'll first show how to get from $n = 2$ to $n = 1$. So the given input to the protocol is

$$v = a_1 g_1 + a_2 g_2 + b_1 h_1 + b_2 h_2 + cu$$

which has the basis $\langle g_1, g_2, h_1, h_2, u \rangle$. The idea is that we want to construct a new (good) vector w whose basis is

$$\langle (g_1 + \lambda^{-1} g_2), (h_1 + \lambda h_2), u \rangle$$

for a randomly chosen challenge $\lambda \in \mathbb{F}_p$.

The construction is the following vector:

$$w(\lambda) := (a_1 + \lambda a_2) \cdot \underbrace{(g_1 + \lambda^{-1} g_2)}_{\text{basis elm}} + (b_1 + \lambda^{-1} b_2) \cdot \underbrace{(h_1 + \lambda h_2)}_{\text{basis elm}} + (a_1 + \lambda a_2)(b_1 + \lambda^{-1} b_2) \underbrace{u}_{\text{basis elm}}.$$

Expanding and isolating the parts with λ and λ^{-1} gives

$$\begin{aligned} w(\lambda) &= (a_1 g_1 + a_2 g_2 + b_1 h_1 + b_2 h_2 + cu) \\ &\quad + \lambda \cdot \underbrace{(a_2 g_1 + b_1 h_2 + a_2 b_1 u)}_{=:w_L} + \lambda^{-1} \cdot \underbrace{(a_1 g_2 + b_2 h_1 + a_1 b_2 u)}_{=:w_R} \\ &= v + \lambda \cdot w_L + \lambda^{-1} \cdot w_R. \end{aligned}$$

Note that, importantly, w_L and w_R don't depend on λ. So this gives a way to provide a construction of a good vector w of half the length (in the new basis) given a good vector v.

This suggests the following protocol:

Algorithm 6.2 (Reducing IPA for $n = 2$ to $n = 1$):

1. Peggy, who knows the a_i's, computes
$$w_L := a_2 g_1 + b_1 h_2 + a_2 b_1 u \in E \quad \text{and} \quad w_R := a_1 g_2 + b_2 h_1 + a_1 b_2 u \in E,$$
and sends those values to Victor. (Note there is no dependence on λ.)
2. Victor picks a random challenge $\lambda \in \mathbb{F}_q$ and sends it.
3. Both Peggy and Victor calculate the point
$$w(\lambda) = v + \lambda \cdot w_L + \lambda^{-1} \cdot w_R \in E.$$
4. Peggy and Victor run the $n = 1$ case of IPA to verify whether $w(\lambda)$ is good with respect the smaller 3-element basis
$$\langle (g_1 + \lambda^{-1} g_2), (h_1 + \lambda h_2), u \rangle.$$
Victor accepts if and only if this IPA is accepted.

Assuming Peggy was truthful and v was indeed good with respect to the original 5-element basis for $n = 2$, the resulting $w(\lambda)$ is good with respect to the new basis. So the interesting part is soundness:

Claim 6.3: Suppose $v = a_1 g_1 + a_2 g_2 + b_1 h_1 + b_2 h_2 + cu$ is given. Assume further that Peggy can provide some $w_L, w_R \in E$ such that

$$w(\lambda) := v + \lambda \cdot w_L + \lambda^{-1} \cdot w_R$$

lies in the span of the shorter basis, and is good for at least four values of λ.

Then all of the following statements must be true:

- $w_L = a_2 g_1 + b_1 h_2 + a_2 b_1 u$,
- $w_R = a_1 g_2 + b_2 h_1 + a_1 b_2 u$,
- $c = a_1 b_1 + a_2 b_2$, i.e., v is good.

Proof: At first, it might seem like a cheating prover has too many parameters they could play with to satisfy too few conditions. The trick is that λ is really like a formal variable, and even the requirement that $w(\lambda)$ lies in the span of

$$\langle (g_1 + \lambda^{-1} g_2), (h_1 + \lambda h_2), u \rangle$$

is going to determine almost all the coefficients of w_L and w_R.

To be explicit, suppose a cheating prover tried to provide

$$\begin{aligned} w_L &= \ell_1 g_1 + \ell_2 g_2 + \ell_3 h_1 + \ell_4 h_2 + \ell_5 \\ w_R &= r_1 g_1 + r_2 g_2 + r_3 h_1 + r_4 h_2 + r_5. \end{aligned}$$

Then we can compute

$$\begin{aligned} w(\lambda) &= v + \lambda \cdot w_L + \lambda^{-1} \cdot w_R \\ &= (a_1 + \lambda \ell_1 + \lambda^{-1} r_1) g_1 + (a_2 + \lambda \ell_2 + \lambda^{-1} r_2) g_2 \\ &+ (b_1 + \lambda \ell_3 + \lambda^{-1} r_3) h_1 + (b_2 + \lambda \ell_4 + \lambda^{-1} r_4) h_1 \\ &+ (c + \lambda \ell_5 + \lambda^{-1} r_5) u. \end{aligned}$$

In order to lie in the span we described, one needs the coefficient of g_1 to be λ times the coefficient of g_2, that is

$$\lambda^{-1} r_1 + a_1 + \lambda \ell_1 = r_2 + \lambda a_2 + \lambda^2 \ell_2.$$

Since this holds for more than three values of λ, the two sides must actually be equal coefficient by coefficient. This means that $\ell_1 = a_2, r_2 = a_1$, and $r_1 = \ell_2 = 0$. In the same way, we get $\ell_4 = b_1, r_3 = b_2$, and $\ell_3 = r_4 = 0$.

So just to lie inside the span, the cheating prover's hand is already forced for all the coefficients other than the ℓ_5 and r_5 in front of u. Then indeed the condition that $w(\lambda)$ is good is that

$$(a_1 + \lambda a_2)(b_1 + \lambda^{-1} b_2) = c + \lambda \ell_5 + \lambda^{-1} r_5.$$

Comparing the constant coefficients we see that $c = a_1 b_1 + a_2 b_2$ as desired. (One also can recover $\ell_5 = a_2 b_1$ and $r_5 = a_1 b_2$, but we never actually use this.) □

So we've shown completeness and soundness for our protocol reducing $n = 2$ to $n = 1$. The general situation is basically the same with more notation. To prevent drowning in notation, we write this out for $n = 6$, with the general case of even n being analogous. So suppose Peggy wishes to prove $v = a_1 g_1 + \ldots + a_6 g_6 + b_1 h_1 + \ldots + b_6 h_6 + cu$ is good with respect to the length-thirteen basis $\langle g_1, \ldots, h_6, u \rangle$.

Algorithm 6.4 (Reducing IPA for $n = 6$ to $n = 3$):

1. Peggy computes

$$\begin{aligned} w_L &= (a_4 g_1 + a_5 g_2 + a_6 g_3) + (b_1 h_4 + b_2 h_5 + b_3 h_6) + (a_1 b_4 + a_2 b_5 + a_3 b_6) u \\ w_R &= (a_1 g_4 + a_2 g_5 + a_3 g_6) + (b_4 h_1 + b_5 h_2 + b_6 h_3) + (a_4 b_1 + a_5 b_2 + a_6 b_3) u \end{aligned}$$

   and sends these to Victor.
2. Victor picks a random challenge $\lambda \in \mathbb{F}_q$.
3. Both parties compute $w(\lambda) = v + \lambda \cdot w_L + \lambda^{-1} \cdot w_R$.
4. Peggy runs IPA for $n = 3$ on $w(\lambda)$ to convince Victor it's good with respect to the length-seven basis

$$\langle g_1 + \lambda^{-1} g_4, g_2 + \lambda^{-1} g_5, g_3 + \lambda^{-1} g_6, h_1 + \lambda h_4, h_2 + \lambda h_5, h_3 + \lambda h_6, u \rangle.$$

§6.3 The base case

If we're in the $n = 1$ case, meaning we have a Pedersen commitment

$$v = ag + bh + cu$$

for $a, b, c \in \mathbb{F}_q$, how can Peggy convince Victor that v is good?

Well, one easy way to do that would be to just reveal all of a, b, c. However, this isn't good enough in situations in which Peggy really cares about the zero-knowledge part. Is there a way to proceed without revealing anything about a, b, c?

The answer is yes, we just need more blinding factors.

Algorithm 6.5 (The $n = 1$ case of IPA):
1. Peggy picks random blinding factors $a', b' \in \mathbb{F}_q$.
2. Peggy sends the following Pedersen commitments:
$$w_1 := a'g + a'bu$$
$$w_2 := b'h + ab'u$$
$$w_3 := a'b'u.$$
3. Victor picks a random challenge $\lambda \in \mathbb{F}_q$.
4. Both parties compute
$$\begin{aligned} w &= v + \lambda \cdot w_1 + \lambda^{-1} \cdot w_2 + \cdot w_3 \\ &= (a + \lambda a')g + (b + \lambda^{-1}b')h + (a + \lambda a')(b + \lambda^{-1}b')u. \end{aligned}$$
5. Victor asks Peggy to reveal all three coefficients of w.
6. Victor verifies that the third coefficient is the product of the first two.

This is really the naive protocol we described except that a and b have each been offset by a blinding factors that prevents Victor from learning anything about a and b: he gets $a + \lambda a'$ and $b + \lambda^{-1}b'$, and knows λ, but since a' and b' are randomly chosen, this reveals no information about a and b themselves.

§6.4 Two simple applications

As we mentioned before, IPA can actually do a lot more than just polynomial commitments.

§6.4.1 Application: revealing an element of a Pedersen commitment

Suppose Peggy has a vector $\vec{a} = \langle a_1, ..., a_n \rangle$ and a Pedersen commitment $v = \sum a_i g_i$ to it. Suppose Peggy wishes to reveal a_1. The right way to think of this is as the dot product $\vec{a} \cdot \vec{b}$, where

$$\vec{b} = \langle 1, 0, ..., 0 \rangle$$

has a 1 in the 1st position and 0's elsewhere. To spell this out:

Algorithm 6.6 (Revealing a_1 in a Pedersen commitment):
1. Both parties compute $w = v + h_1 + a_1 u$.
2. Peggy runs IPA on w to convince Victor that w is good.

§6.4.2 Application: showing two Pedersen commitments are to the same vector

Suppose there are two Pedersen commitments $v = \sum a_i g_i$ and $v' = \sum a'_i g'_i$ in different bases; Peggy wants to prove that $a_i = a'_i$ for all i (i.e. the vectors $\vec{a}$ and $\vec{a'}$ coincide) without revealing anything else about the two vectors.

This can also be done straightforwardly: show that the dot products of $\vec{a}$ and $\vec{a}'$ with a random other vector $\vec{\lambda}$ are equal.

Algorithm 6.7 (Matching Pedersen commitments):
1. Victor picks a random challenge vector $\vec{\lambda} = \langle \lambda_1, ..., \lambda_n \rangle \in \mathbb{F}_q^n$.
2. Both parties compute its Pedersen commitment $w = \lambda_1 h_1 + ... + \lambda_n h_n$.
3. Peggy also privately computes the dot product $c := \vec{a} \cdot \vec{\lambda} = \vec{a}' \cdot \vec{\lambda} = a_1\lambda_1 + ... + a_n\lambda_n$.
4. Peggy sends a Pedersen commitment cu to the number c.
5. Peggy runs IPA to convince Victor both $v + w + cu$ and $v' + w + cu$ are good.

This protocol provides a proof to Victor that $\vec{a}$ and $\vec{a}'$ have the same dot product with his random challenge vector $\vec{\lambda}$, without having to actually reveal this dot product. Since Victor chose the random vector $\vec{\lambda}$, this check passes with vanishingly small probability of at most $\frac{1}{q}$ if $\vec{a} \neq \vec{a}'$.

§6.5 Using IPA for polynomial commitments

Suppose now $P(X) = \sum a_i X^{i-1}$ is a given polynomial. Then Peggy can use IPA to commit the polynomial P as follows:

- Peggy publishes Pedersen commitment of the coefficients of P; that is, Peggy publishes
$$g := \sum a_i g_i \in E.$$
- Suppose Victor wants to open the commitment at a value z, and Peggy asserts that $P(z) = y$.
- Victor picks a random constant $\lambda \in \mathbb{F}_p$.
- Both parties compute
$$v := \underbrace{(a_1 g_1 + ... + a_n g_n)}_{C} + (\lambda z^0 h_1 + ... + \lambda z^{n-1} h_n) + \lambda y u$$
and run IPA on it.

(When Peggy does a vanilla IPA protocol, she can keep all $2n + 1$ coefficients secret. In this context, Peggy has published the first part g and still gets to keep her coefficients a_n private from Victor. The other $n + 1$ coefficients are globally known because they're inputs to the protocol for opening the commitment at z.)

The introduction of the hacked constant λ might be a bit of a surprise. The reason is that without it, there is an amusing loophole that Peggy can exploit. Peggy can pick the vector v, so imagine she tries to swindle Victor by reporting $v = a_1 g_1 + ... + a_n g_n - 10u$ instead of the honest $v = a_1 g_1 + ... + a_n g_n$. Then, Peggy inflates all the values of y she claims to Victor by 10. This would allow Peggy to cheat Victor into committing the polynomial P but for each input z giving Victor the value of $P(z) + 10$ rather than $P(z)$ (though the cheating offset would be the same at every value she opened). The offset λ prevents this attack.

§7 PLONK, a zkSNARK protocol

For this section, one can use any polynomial commitment scheme one prefers. So we'll introduce the notation $\mathrm{Com}(P)$ for the commitment of a polynomial $P(X) \in \mathbb{F}_q[X]$, with the understanding that either KZG, IPA, or something else could be in use here.

§7.1 Root check (using long division with commitment schemes)

Both commitments schemes allow for a technique I privately call *root-check* here. Here's the problem statement:

> *Problem 7.1*: Suppose one had two polynomials P_1 and P_2, and Peggy has given commitments $\mathrm{Com}(P_1)$ and $\mathrm{Com}(P_2)$. Peggy would like to prove to Victor that, say, the equation $P_1(z) = P_2(z)$ for all z in some large finite set S.

Of course, Peggy could open $\mathrm{Com}(P_1)$ and $\mathrm{Com}(P_2)$ at every point in S. But there are some situations in which Peggy still wants to prove this without actually revealing the common values of the polynomial for any $z \in S$. Even when S is a single number (i.e. Peggy wants to show P_1 and P_2 agree on a single value without revealing the common value), it's not obvious how to do this.

Well, it turns out we can basically employ the same technique as in Section 5. Peggy just needs to show is that $P_1 - P_2$ is divisible by $Z(X) := \prod_{z \in S}(X - z)$. This can be done by committing the quotient $H(X) := \frac{P_1(X) - P_2(X)}{Z(X)}$. Victor then gives a random challenge $\lambda \in \mathbb{F}_q$, and then Peggy opens $\mathrm{Com}(P_1)$, $\mathrm{Com}(P_2)$, and $\mathrm{Com}(H)$ at λ.

But we can actually do this more generally with *any* polynomial expression F in place of $P_1 - P_2$, as long as Peggy has a way to prove the values of F are correct. As an artificial example, if Peggy has sent Victor $\mathrm{Com}(P_1)$ through $\mathrm{Com}(P_6)$, and wants to show that

$$P_1(42) + P_2(42)P_3(42)^4 + P_4(42)P_5(42)P_6(42) = 1337,$$

she could define $F(X) = P_1(X) + P_2(X)P_3(X)^4 + P_4(X)P_5(X) + P_6(X)$ and run the same protocol with this F. This means she doesn't have to reveal any $P_i(42)$, which is great!

To be fully explicit, here is the algorithm:

Algorithm 7.2 (Root-check): Assume that F is a polynomial for which Peggy can establish the value of F at any point in $\mathbb{F}_q$. Peggy wants to convince Victor that F vanishes on a given finite set $S \subseteq \mathbb{F}_q$.

1. Both parties compute the polynomial
$$Z(X) := \prod_{z \in S}(X - z) \in \mathbb{F}_q[X].$$
2. Peggy does polynomial long division to compute $H(X) = \frac{F(X)}{Z(X)}$.
3. Peggy sends $\mathrm{Com}(H)$.
4. Victor picks a random challenge $\lambda \in \mathbb{F}_q$ and asks Peggy to open $\mathrm{Com}(H)$ at λ, as well as the value of F at λ.
5. Victor verifies $F(\lambda) = Z(\lambda)H(\lambda)$.

§7.2 Arithmetization

The promise of programmable cryptography is that we should be able to perform zero-knowledge proofs for arbitrary functions. That means we need a "programming language" that we'll write our function in.

For PLONK (and Groth16 in the next section), the choice that's used is: **systems of quadratic equations over $\mathbb{F}_q$**.

This leads to the natural question of how a function like SHA256 can be encoded into a system of quadratic equations. Well, quadratic equations over $\mathbb{F}_q$, viewed as an NP-problem called Quad-SAT, is pretty clearly NP-complete, as the following example shows:

Remark 7.3 (Quad-SAT is pretty obviously NP-complete): If you can't see right away that Quad-SAT is NP-complete, the following example instance can help, showing how to convert any instance of 3-SAT into a Quad-SAT problem:

$$\begin{aligned}
x_i^2 &= x_i \quad \forall 1 \leq i \leq 1000 \\
y_1 &= (1 - x_{42}) \cdot x_{17}, & 0 &= y_1 \cdot x_{53} \\
y_2 &= (1 - x_{19}) \cdot (1 - x_{52}) & 0 &= y_2 \cdot (1 - x_{75}) \\
y_3 &= x_{25} \cdot x_{64}, & 0 &= y_3 \cdot x_{81} \\
&\vdots
\end{aligned}$$

(imagine many more such pairs of equations). The x_i's are variables which are seen to either be 0 or 1. And then each pair of equations with y_i corresponds to a clause of 3-SAT.

So for example, any NP decision problem should be encodable. Still, such a theoretical reduction might not be usable in practice: polynomial factors might not matter in complexity theory, but they do matter a lot to engineers and end users. Just having a galactic algorithm isn't enough.

But it turns out that Quad-SAT is actually reasonably code-able. This is the goal of projects like Circom, which gives a high-level language that compiles a function like SHA-256 into a system of equations over $\mathbb{F}_q$ that can actually be used in practice. Systems like this are called **arithmetic circuits**, and Circom is appropriately short for "circuit compiler". If you're curious, you can see how SHA256 is implemented in Circom on GitHub.

To preserve continuity of the mathematics, we'll defer further discussion of coding in quadratic equations to later.

§7.3 An instance of PLONK

For PLONK, the equations are standardized further to a certain form:

Definition 7.4: An instance of PLONK consists of two pieces, the **gate constraints** and the **copy constraints**.

The **gate constraints** are a system of n equations,

$$q_{L,i}a_i + q_{R,i}b_i + q_{O,i}c_i + q_{M,i}a_ib_i + q_{C,i} = 0$$

for $i = 1, \ldots, n$, in the $3n$ variables a_i, b_i, c_i. while the $q_{*,i}$ are coefficients in $\mathbb{F}_q$, which are globally known. The confusing choice of subscripts stands for "Left", "Right", "Output", "Multiplication", and "Constant", respectively.

The **copy constraints** are a bunch of assertions that some of the $3n$ variables should be equal to each other, so e.g. "$a_1 = c_7$", "$b_{17} = b_{42}$", and so on.

So the PLONK protocol purports to do the following: Peggy and Victor have a PLONK instance given to them. Peggy has a solution to the system of equations, i.e. an assignment of values to each a_i, b_i, c_i such that all the gate constraints and all the copy constraints are satisfied. Peggy wants to prove this to Victor succinctly and without revealing the solution itself. The protocol then proceeds by having:

1. Peggy sends a polynomial commitment corresponding to a_i, b_i, and c_i (the details of what polynomial are described below).
2. Peggy proves to Victor that the commitment from Step 1 satisfies the gate constraints.
3. Peggy proves to Victor that the commitment from Step 1 also satisfies the copy constraints.

Let's now explain how each step works.

§7.4 Step 1: The commitment

In PLONK, we'll assume that $q \equiv 1 \bmod n$, which means that we can fix $\omega \in \mathbb{F}_q$ to be a primitive nth root of unity.

Then, by polynomial interpolation, Peggy chooses polynomials $A(X)$, $B(X)$, and $C(X)$ in $\mathbb{F}_q[X]$ such that

$$A(\omega^i) = a_i, \quad B(\omega^i) = b_i, \quad C(\omega^i) = c_i \quad \text{for all } i = 1, 2, \ldots, n. \tag{3}$$

We specifically choose ω^i because that way, if we use Algorithm 7.2 on the set $\{\omega, \omega^1, \ldots, \omega^n\}$, then the polynomial called Z is just $Z(X) = (X - \omega)\ldots(X - \omega^n) = X^n - 1$, which is really nice. In fact, often n is chosen to be a power of 2 so that A, B, and C are really easy to compute, using a fast Fourier transform. (Note: When you're working in a finite field, the fast Fourier transform is sometimes called the "number theoretic transform" (NTT) even though it's exactly the same as the usual FFT.)

Then:

Algorithm 7.5 (Commitment step of PLONK):
1. Peggy interpolates A, B, C as in Equation 3.
2. Peggy sends $\mathrm{Com}(A)$, $\mathrm{Com}(B)$, $\mathrm{Com}(C)$ to Victor.

To reiterate, each commitment is a 256-bit that can later be "opened" at any value $x \in \mathbb{F}_q$.

§7.5 Step 2: Gate-check

Both Peggy and Victor knows the PLONK instance, so they can interpolate a polynomial $Q_{L(X)} \in \mathbb{F}_q[X]$ of degree $n-1$ such that

$$Q_L(\omega^i) = q_{L,i} \quad \text{for } i = 1, ..., n.$$

Then the analogous polynomials Q_R, Q_O, Q_M, Q_C are defined in the same way.

Now, what do the gate constraints amount to? Peggy is trying to convince Victor that the equation

$$Q_L(x)A(x) + Q_R(x)B(x) + Q_O(x)C(x) + Q_M(x)A(x)B(x) + Q_C(x) = 0 \tag{4}$$

is true for the n numbers $x = 1, \omega, \omega^2, ..., \omega^{n-1}$.

However, Peggy has committed A, B, C already, while all the Q_* polynomials are globally known. So this is a direct application of Algorithm 7.2:

Algorithm 7.6 (Gate-check):
1. Both parties interpolate five polynomials $Q_* \in \mathbb{F}_q[X]$ from the $15n$ coefficients q_* (globally known from the PLONK instance).
2. Peggy uses Algorithm 7.2 to convince Victor that Equation 4 holds for $X = \omega^i$ (that is, the left-hand side is is indeed divisible by $Z(X) := X^n - 1$).

§7.6 Step 3: Proving the copy constraints

The copy constraints are the trickier step. There are a few moving parts to this idea, so to ease into it slightly, we provide a solution to a "simpler" problem called "permutation-check". Then we explain how to deal with the full copy check.

§7.6.1 Easier case: permutation-check

So let's suppose we have polynomials $P, Q \in \mathbb{F}_q[X]$ which are encoding two vectors of values

$$\vec{p} = \langle P(\omega^1), P(\omega^2), ..., P(\omega^n) \rangle$$
$$\vec{q} = \langle Q(\omega^1), Q(\omega^2), ..., Q(\omega^n) \rangle.$$

Is there a way that one can quickly verify $\vec{p}$ and $\vec{q}$ are the same up to permutation of the n entries?

Well, actually, it would be necessary and sufficient for the identity

$$(T + P(\omega^1))(T + P(\omega^2))...(T + P(\omega^n)) = (T + Q(\omega^1))(T + Q(\omega^2))...(T + Q(\omega^n)) \tag{5}$$

to be true, in the sense both sides are the same polynomial in $\mathbb{F}_q[T]$ in a single formal variable T. And for that, it actually is sufficient that a single random challenge $T = \lambda$ passes Equation 5; as if the two sides of Equation 5 aren't the same polynomial, then the two sides can have at most $n-1$ common values.

We can then get a proof of Equation 5 using the technique of adding an *accumulator polynomial*. The idea is this: Victor picks a random challenge $\lambda \in \mathbb{F}_q$. Peggy then interpolates the polynomial $F_P \in \mathbb{F}_q[T]$ such that

$$
\begin{aligned}
F_P(\omega^1) &= \lambda + P(\omega^1) \\
F_P(\omega^2) &= (\lambda + P(\omega^1))(\lambda + P(\omega^2)) \\
&\vdots \\
F_P(\omega^n) &= (\lambda + P(\omega^1))(\lambda + P(\omega^2))\cdots(\lambda + P(\omega^n)).
\end{aligned}
$$

Then the accumulator $F_Q \in \mathbb{F}_{q[T]}$ is defined analogously.

So to proev Equation 5, the following algorithm works:

Algorithm 7.7 (Permutation-check): Suppose Peggy has committed $\mathrm{Com}(P)$ and $\mathrm{Com}(Q)$.

1. Victor sends a random challenge $\lambda \in \mathbb{F}_q$.
2. Peggy interpolates polynomials $F_P[T]$ and $F_Q[T]$ such that $F_P(\omega^k) = \prod_{i \le k}(\lambda + P(\omega^i))$. Define F_Q similarly. Peggy sends $\mathrm{Com}(F_P)$ and $\mathrm{Com}(F_Q)$.
3. Peggy uses Algorithm 7.2 to prove all of the following statements:
   - $F_P(X) - (\lambda + P(X))$ vanishes at $X = \omega$;
   - $F_P(\omega X) - (\lambda + P(X))F_P(X)$ vanishes at $X \in \{\omega, \ldots, \omega^{n-1}\}$;
   - The previous two statements also hold with F_P replaced by F_Q;
   - $F_P(X) - F_Q(X)$ vanishes at $X = 1$.

§7.6.2 Copy check

Moving on to copy-check, let's look at a concrete example where $n = 4$. Suppose that our copy constraints were

$$\textcircled{a_1} = \textcircled{a_4} = \textcircled{c_3} \quad \text{and} \quad \boxed{b_2} = \boxed{c_1}.$$

(We've colored and circled the variables that will move around for readability.) So, the copy constraint means we want the following equality of matrices:

$$
\begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \end{pmatrix} = \begin{pmatrix} \textcircled{a_4} & a_2 & a_3 & \textcircled{c_3} \\ b_1 & \boxed{c_1} & b_3 & b_4 \\ \boxed{b_2} & c_2 & c_3 & \textcircled{a_1} \end{pmatrix}. \tag{6}
$$

Again, our goal is to make this into a *single* equation. There's a really clever way to do this by tagging each entry with $+\eta^j\omega^k\mu$ in reading order for $j = 0, 1, 2$ and $k = 1, \ldots, n$; here $\eta \in \mathbb{F}_q$ is any number such that η^2 doesn't happen to be a power of ω, so all the tags are distinct. Specifically, if Equation 6 is true, then for any $\mu \in \mathbb{F}_q$, we also have

$$
\begin{aligned}
&\begin{pmatrix} a_1 + \omega^1\mu & a_2 + \omega^2\mu & a_3 + \omega^3\mu & a_4 + \omega^4\mu \\ b_1 + \eta\omega^1\mu & b_2 + \eta\omega^2\mu & b_3 + \eta\omega^3\mu & b_4 + \eta\omega^4\mu \\ c_1 + \eta^2\omega^1\mu & c_2 + \eta^2\omega^2\mu & c_3 + \eta^2\omega^3\mu & c_4 + \eta^2\omega^4\mu \end{pmatrix} \\
&= \begin{pmatrix} \textcircled{a_4} + \omega^1\mu & a_2 + \omega^2\mu & a_3 + \omega^3\mu & \textcircled{c_3} + \omega^4\mu \\ b_1 + \eta\omega^1\mu & \boxed{c_1} + \eta\omega^2\mu & b_3 + \eta\omega^3\mu & b_4 + \eta\omega^4\mu \\ \boxed{b_2} + \eta^2\omega^1\mu & c_2 + \eta^2\omega^2\mu & c_3 + \eta^2\omega^3\mu & \textcircled{a_1} + \eta^2\omega^4\mu \end{pmatrix}.
\end{aligned} \tag{7}
$$

Now how can the prover establish Equation 7 succinctly? The answer is to run a permutation-check on the $3n$ entries of Equation 7! Because μ was a random challenge, one can really think of each binomial above more like an ordered pair: for almost all challenges μ, there are no "unexpected" equalities. In other words, up to a negligible number of μ, Equation 7 will be true if and only if the right-hand side is just a permutation of the left-hand side.

To clean things up, shuffle the 12 terms on the right-hand side of Equation 7 so that each variable is in the cell it started at:

$$\begin{aligned}\text{Want to prove } &\begin{pmatrix} a_1+\omega^1\mu & a_2+\omega^2\mu & a_3+\omega^3\mu & a_4+\omega^4\mu \\ b_1+\eta\omega^1\mu & b_2+\eta\omega^2\mu & b_3+\eta\omega^3\mu & b_4+\eta\omega^4\mu \\ c_1+\eta^2\omega^1\mu & c_2+\eta^2\omega^2\mu & c_3+\eta^2\omega^3\mu & c_4+\eta^2\omega^4\mu \end{pmatrix} \\ \text{is a permutation of } &\begin{pmatrix} a_1+\eta^2\omega^4\mu & a_2+\omega^2\mu & a_3+\omega^3\mu & a_4+\omega^1\mu \\ b_1+\eta\omega^1\mu & b_2+\eta^2\omega^1\mu & b_3+\eta\omega^3\mu & b_4+\eta\omega^4\mu \\ b_1+\eta\omega^2\mu & c_2+\eta^2\omega^2\mu & c_3+\eta^2\omega^3\mu & c_4+\omega^4\mu \end{pmatrix}.\end{aligned} \tag{8}$$

The permutations needed are part of the problem statement, hence globally known. So in this example, both parties are going to interpolate cubic polynomials $\sigma_a, \sigma_b, \sigma_c$ that encode the weird coefficients row-by-row:

$$\begin{aligned} \sigma_a(\omega^1) &= \eta^2\omega^4 & \sigma_a(\omega^2) &= \omega^2 & \sigma_a(\omega^3) &= \omega^3 & \sigma_a(\omega^4) &= \omega^1 \\ \sigma_b(\omega^1) &= \eta\omega^1 & \sigma_b(\omega^2) &= \eta^2\omega^1 & \sigma_b(\omega^3) &= \eta\omega^3 & \sigma_b(\omega^4) &= \eta\omega^4 \\ \sigma_c(\omega^1) &= \eta\omega^2 & \sigma_c(\omega^2) &= \eta^2\omega^2 & \sigma_c(\omega^3) &= \eta^2\omega^3 & \sigma_c(\omega^4) &= \omega^4. \end{aligned}$$

Then one can start defining accumulator polynomials, after re-introducing the random challenge λ from permutation-check. We're going to need six in all, three for each side of Equation 8: we call them $F_a, F_b, F_c, F_{a'}, F_{b'}, F_{c'}$. The ones on the left-hand side are interpolated so that

$$\begin{aligned} F_a(\omega^k) &= \prod_{i\le k}(a_i+\omega^i\mu+\lambda) \\ F_b(\omega^k) &= \prod_{i\le k}(b_i+\eta\omega^i\mu+\lambda) \\ F_c(\omega^k) &= \prod_{i\le k}(c_i+\eta^2\omega^i\mu+\lambda) \end{aligned} \tag{9}$$

whilst the ones on the right have the extra permutation polynomials

$$\begin{aligned} F'_a(\omega^k) &= \prod_{i\le k}(a_i+\sigma_a(\omega^i)\mu+\lambda) \\ F'_b(\omega^k) &= \prod_{i\le k}(b_i+\sigma_b(\omega^i)\mu+\lambda) \\ F'_c(\omega^k) &= \prod_{i\le k}(c_i+\sigma_c(\omega^i)\mu+\lambda). \end{aligned} \tag{10}$$

And then we can run essentially the algorithm from before. There are six initialization conditions

$$
\begin{aligned}
F_a(\omega^1) &= A(\omega^1) + \omega^1\mu + \lambda \\
F_b(\omega^1) &= B(\omega^1) + \eta\omega^1\mu + \lambda \\
F_c(\omega^1) &= C(\omega^1) + \eta^2\omega^1\mu + \lambda \\
F_a(\omega^1) &= A(\omega^1) + \sigma_a(\omega^1)\mu + \lambda \\
F_b(\omega^1) &= B(\omega^1) + \sigma_b(\omega^1)\mu + \lambda \\
F_c(\omega^1) &= C(\omega^1) + \sigma_c(\omega^1)\mu + \lambda.
\end{aligned} \tag{11}
$$

and six accumulation conditions

$$
\begin{aligned}
F_a(\omega X) &= F_a(X) \cdot (A(X) + X\mu + \lambda) \\
F_b(\omega X) &= F_b(X) \cdot (B(X) + 2X\mu + \lambda) \\
F_c(\omega X) &= F_c(X) \cdot (C(X) + 3X\mu + \lambda) \\
F'_a(\omega X) &= F'_a(X) \cdot (A(X) + \sigma_a(X)\mu + \lambda) \\
F'_b(\omega X) &= F'_b(X) \cdot (B(X) + \sigma_b(X)\mu + \lambda) \\
F'_c(\omega X) &= F'_c(X) \cdot (C(X) + \sigma_c(X)\mu + \lambda)
\end{aligned} \tag{12}
$$

before the final product condition

$$
F_a(1)F_b(1)F_c(1) = F'_a(1)F'_b(1)F'_c(1) \tag{13}
$$

To summarize, the copy-check goes as follows:

Algorithm 7.8 (Copy-check):

1. Both parties compute the degree $n-1$ polynomials $\sigma_a, \sigma_b, \sigma_c \in \mathbb{F}_q[X]$ described above, based on the copy constraints in the problem statement.
2. Victor chooses random challenges $\mu, \lambda \in \mathbb{F}_q$ and sends them to Peggy.
3. Peggy interpolates the six accumulator polynomials F_a, ..., F'_c defined in Equation 9 and Equation 10.
4. Peggy uses Algorithm 7.2 to prove Equation 11 holds.
5. Peggy uses Algorithm 7.2 to prove Equation 12 holds for $X \in \{\omega, \omega^2, ..., \omega^{n-1}\}$.
6. Peggy uses Algorithm 7.2 to prove Equation 13 holds.

§7.7 Public and private witnesses

TODO: warning: A, B, C should not be the lowest degree interpolations, imo AV: why not? I think it's fine if they are

The last thing to be done is to reveal the value of public witnesses, so the prover can convince the verifier that those values are correct. This is simply an application of Algorithm 7.2. Let's say the public witnesses are the values a_i, for all i in some set S. (If some of the b's and c's are also public, we'll just do the same thing for them.) The prover can interpolate another polynomial, A^{public}, such that $A^{\text{public}(\omega^i)} = a_i$ if $i \in S$, and $A^{\text{public}(\omega^i)} = 0$ if $i \notin S$. Actually, both the prover and the verifier can compute A^{public}, since all the values a_i are publicly known!

Now the prover runs Algorithm 7.2 to prove that $A - A^{\text{public}}$ vanishes on S. (And similarly for B and C, if needed.) And we're done.

§8 Groth16, another zkSNARK protocol

Like PLONK, Groth16 is a protocol for quadratic equations, as we described in Section 7.2. For Groth16, the format of the equations is in so-called *R1CS format*.

§8.1 Input format

To describe the technical specification of the input format, we call the variables $a_0 = 1, a_1, ..., a_n$. The q'th equation takes the form

$$\left(\sum_{i=0}^{n} u_{i,q} a_i\right)\left(\sum_{i=0}^{n} v_{i,q} a_i\right) = \left(\sum_{i=0}^{n} a_i w_{i,q}\right)$$

for $1 \leq q \leq m$, where m is the number of equations. (In other words, we require the quadratic part of each equation to factor. The $a_0 = 1$ term is a dummy variable that simplifies notation so that we don't need to write the constant terms separately.)

The inputs are divided into two categories:

- $(a_0), a_1, ..., a_\ell$ are *public inputs*; and
- $a_{\ell+1}, a_1, ..., a_n$ are *private inputs*.

§8.2 Interpolation

The basic idea of Groth16 is to interpolate polynomials through the coefficients of the m equations, then work with KZG commitments to these polynomials. (This is sort of the opposite of the interpolation in PLONK (Section 7), where we interpolate a polynomial through Peggy's solution $(a_0, ..., a_n)$. Philosophically, you might also think of this as verifying a random linear combination of the m equations – where the coefficients of the random linear combination are determined by the unknown secret s from the KZG protocol.)

Interpolate polynomials U_i, V_i, W_i such that

$$\begin{aligned} U_i(q) &= u_{i,q} \\ V_i(q) &= v_{i,q} \\ W_i(q) &= w_{i,q} \end{aligned}$$

for $1 \leq q \leq m$. In this notation, we want to show that

$$\left(\sum_{i=0}^{n} a_i U_i(q)\right)\left(\sum_{i=0}^{n} a_i V_{i(q)}\right) = \left(\sum_{i=0}^{n} a_i W_{i(q)}\right),$$

for $q = 1, ..., m$. This is the same as showing that there exists some polynomial H such that

$$\left(\sum_{i=0}^{n} a_i U_i(X)\right)\left(\sum_{i=0}^{n} a_i V_i(X)\right) = \left(\sum_{i=0}^{n} a_i W_i(X)\right) + H(X)T(X),$$

where

$$T(X) = (X-1)(X-2)...(X-m).$$

The proof that Peggy sends to Victor will take the form of a handful of KZG commitments. As a first idea (we'll have to build on this afterwards), let's have Peggy send KZG commitments

$$\text{Com}\left(\sum_{i=0}^{n} a_i U_i\right), \quad \text{Com}\left(\sum_{i=0}^{n} a_i V_i\right), \quad \text{Com}\left(\sum_{i=0}^{n} a_i W_i\right), \quad \text{Com}(HT).$$

Recall from Section 5 that the Kate commitment $\text{Com}(F)$ to a polynomial F is just the elliptic curve point $[F(s)]$. Here s is some field element whose value nobody knows, but a handful of small powers $[1]$, $[s]$, $[s^2]$, ..., are known from the trusted setup.

The problem here is for Peggy to convince Victor that these four group elements, supposedly

$$\text{Com}\left(\sum_{i=0}^{n} a_i U_i\right)$$

and so forth, are well-formed. For example, Peggy needs to show that

$$\text{Com}\left(\sum_{i=0}^{n} a_i U_i\right)$$

is a linear combination of the KZG commitments $\text{Com}(U_i)$, that

$$\text{Com}\left(\sum_{i=0}^{n} a_i V_i\right)$$

is a linear combination of the KZG commitments $\text{Com}(V_i)$, and that the two linear combinations use the same coefficients a_i. How can Peggy prove this sort of claim?

§8.3 Proving claims about linear combinations

We've already come across this sort of challenge in the setting of IPA (Section 6), but Groth16 uses a different approach, so let's get back to a simple toy example.

Example 8.1: Suppose there are publicly known group elements

$$\mathrm{Com}(U_1), \mathrm{Com}(U_2), ..., \mathrm{Com}(U_n).$$

Suppose Peggy has another group element g, and she wants to show that g has the form

$$g = a_0 + \sum_{i=1}^{n} a_i \, \mathrm{Com}(U_i),$$

where the a_i's are constants Peggy knows.

Groth's solution to this problem uses a *trusted setup* phase, as follows.

Before the protocol runs, Trent (our trusted setup agent) chooses a nonzero field element δ at random and publishes:

$$[\delta], \mathrm{Com}(U_1), \mathrm{Com}(\delta U_1), \mathrm{Com}(U_2), \mathrm{Com}(\delta U_2), ..., \mathrm{Com}(U_n), \mathrm{Com}(\delta U_n).$$

Trent then throws away δ.

Peggy now sends to Victor

$$g = [a_0] + \sum_{i=1}^{n} a_i \, \mathrm{Com}(U_i)$$

and

$$h = \delta g = a_0[\delta] + \sum_{i=1}^{n} a_i \, \mathrm{Com}(\delta U_i).$$

Victor can verify that

$$\mathrm{pair}(g, [\delta]) = \mathrm{pair}(h, [1]),$$

which shows that the element Peggy said was δg is in fact δ times the element g.

But Peggy does not know δ! So (assuming, as usual, that the discrete logarithm problem is hard), the only way Peggy can find elements g and h such that $h = \delta g$ is to use the commitments Trent released in trusted setup. In other words, g must be a linear combination of the elements $[1], \mathrm{Com}(U_1), ..., \mathrm{Com}(U_n)$, which Peggy knows how to multiply by δ, and h must be the same linear combination of $[\delta], \mathrm{Com}(\delta U_1), ..., \mathrm{Com}(\delta U_n)$.

Example 8.2: Here's a more complicated challenge, on the way to building up the Groth16 protocol.

Suppose there are publicly known group elements

$$\mathrm{Com}(U_1), \mathrm{Com}(U_2), ..., \mathrm{Com}(U_n)$$

and

$$\mathrm{Com}(V_1), \mathrm{Com}(V_2), ..., \mathrm{Com}(V_n).$$

Peggy wants to publish

$$g_1 = \sum_{i=1}^{n} a_i \,\mathrm{Com}(U_i)$$

and

$$g_2 = \sum_{i=1}^{n} a_i \,\mathrm{Com}(V_i),$$

and prove to Victor that these two group elements have the desired form (in particular, with the same coefficients a_i used for both).

To do this, Trent does the same trusted setup thing. Trent chooses two constants α and β and publishes $[\alpha]$, $[\beta]$, and

$$\alpha\,\mathrm{Com}(U_i) + \beta\,\mathrm{Com}(V_i),$$

for $1 \leq i \leq n$.

In addition to g_1 and g_2, Peggy now also publishes

$$h = \sum_{i=1}^{n} a_i(\alpha\,\mathrm{Com}(U_i) + \beta\,\mathrm{Com}(V_i)).$$

Victor needs to verify that

$$h = \alpha g_1 + \beta g_2;$$

if this equality holds, then g_1 and g_2 must have the correct form, just like in Example 8.1.

So Victor checks the equality of pairings

$$\mathrm{pair}(h, [1]) = \mathrm{pair}(g_1, [\alpha]) + \mathrm{pair}(g_2, [\beta]),$$

and the proof is complete.

§8.4 The protocol

Armed with Example 8.1 and Example 8.2, it's not hard to turn our vague idea from earlier into a full protocol. This protocol won't be zero-knowledge – to make it zero-knowledge, we would have to throw in an extra "blinding" term, which just adds an additional layer of complication on top of the whole thing. If you want to see the full ZK version, check out Groth's original paper.

§8.4.1 Trusted setup

We start with the same secret setup as the KZG commitment scheme. That is, we have a fixed pairing pair : $E \times E \to \mathbb{Z}/N\mathbb{Z}$ and a secret scalar $s \in \mathbb{F}_p$. The trusted setup is as before: s is kept secret from everyone, but the numbers $[1]$, $[s]$, ..., up to $[s^{2m}]$ are published.

However, this protocol requires additional setup that actually depends on the system of equations (unlike in the KZG situation in Section 5, in which trusted setup is done for the curve E itself and afterwards can be freely reused for any situation needing a KZG commitment.)

Specifically, let's interpolate polynomials U, V, W through the coefficients of our R1CS system; that is we have $U_i(X), V_i(X), W_i(X) \in \mathbb{F}_p[X]$ such that

$$U_i(q) = u_{i,q}, \quad V_i(q) = v_{i,q}, \quad W_i(q) = w_{i,q}.$$

So far we have ignored the issue of public inputs. The values $a_0, a_1, ..., a_\ell$ will be public inputs to the circuit, so both Peggy and Victor know their values, and Victor has to be able to verify that they were assigned correctly. The remaining values $a_{\ell+1}, ..., a_n$ will be private.

Trent (who is doing the trusted setup) then selects secrets $\alpha, \beta, \delta, \varepsilon \in \mathbb{F}_p$ and publishes all of the following points on E:

$$[\alpha], [\beta], [\delta], [\varepsilon],$$
$$\left[\frac{\beta U_i(s) + \alpha V_i(s) + W_i(s)}{\delta}\right] \text{ for } \ell < i \le m, \quad \left[\frac{x^i T(s)}{\varepsilon}\right] \text{ for } 0 \le i \le n-2.$$

Note that this means this setup needs to be done *for each system of equations*. That is, if you are running Groth16 and you change the system, the trusted setup with δ needs to be redone.

This might make the protocol seem limited. On the other hand, for practical purposes, one can imagine that Peggy has a really general system of equations that she wants to prove many solutions for. In this case, Trent can run the trusted setup just once, and once the setup is done there is no additional cost.

Example 8.3: In practice, one often wants to prove a computation of a hash function:

$$\text{sha}(M) = H.$$

When you convert this into a system of quadratic equations for PLONK or Groth16, both M and H will be public inputs to the system. The equations themselves will depend only on the details of the hash function sha.

In this case, a single trusted setup can be used to prove the hash of any message.

§8.4.2 The protocol (not optimized)

1. Peggy now sends to Victor:

$$A = \left[\sum_{i=0}^{n} a_i U_i(s)\right], \quad B = \left[\sum_{i=0}^{n} a_i V_i(s)\right], \quad C = \left[\sum_{i=0}^{n} a_i W_i(s)\right],$$
$$D = \left[\sum_{i=\ell+1}^{n} a_i \frac{\beta U_i(s) + \alpha V_i(s) + W_i(s)}{\delta}\right],$$
$$E = [H(s)], \quad F = \left[H(s)\frac{T(s)}{\varepsilon}\right].$$

2. Victor additionally computes

$$D_0 = \left[\sum_{i=1}^{\ell} (\beta U_i(s) + \alpha V_i(s) + W_i(s))\right]$$

and

$$G = [T(s)]$$

based on publicly known information.

3. Victor verifies the pairings

$$\text{pair}([\delta], D) + \text{pair}([1], D_0) = \text{pair}([\beta], A) + \text{pair}([\alpha], B) + \text{pair}([1], C).$$

This pairing shows that

$$\delta D + D_0 = \beta A + \alpha B + C.$$

Now just like in Example 8.1, the only way that Peggy could possibly find two group elements g and h such that $\delta g = h$ is if g is a linear combination of terms $\left[\frac{\beta U_i(s) + \alpha V_i(s) + W_i(s)}{\delta}\right]$. So we have verified that

$$D = \left[\sum_{i=\ell+1}^{n} a_i \frac{\beta U_i(s) + \alpha V_i(s) + W_i(s)}{\delta}\right]$$

for some constants a_i, which implies

$$\beta A + \alpha B + C = \left[\sum_{i=0}^{n} a_i (\beta U_i(s) + \alpha V_i(s) + W_i(s))\right].$$

And just like in Example 8.2, since α and β are unknown, the only way an equality like this can hold is if

$$A = \left[\sum_{i=0}^{n} a_i U_i(s)\right], \quad B = \left[\sum_{i=0}^{n} a_i V_i(s)\right], \quad C = \left[\sum_{i=0}^{n} a_i W_i(s)\right],$$

where a_i is equal to the public input for $i \leq \ell$ (because Victor computed D_0 himself!) and a_i is equal to some fixed unknown value for $i > \ell$.

4. Victor verifies that

$$\text{pair}([\varepsilon], F) = \text{pair}(E, G).$$

Again like in Example 8.1, since ε is unknown, this shows that F has the form

$$\left[\frac{H(s)T(s)}{\varepsilon}\right],$$

where H is a polynomial of degree at most $n-2$. Since $G = [T(s)]$ (Victor knows this because he computed it himself), we learn that $E = [H(s)]$ is a KZG commitment to a polynomial whose coefficients Peggy knows.

5. Finally, Victor verifies that

$$\text{pair}(A, B) = \text{pair}([1], C) + \text{pair}(E, G).$$

At this point, Victor already knows that A, B, C, E, H have the correct form, so this last pairing check convinces Victor of the key equality,

$$\left(\sum_{i=0}^{n} a_i U_i(X)\right)\left(\sum_{i=0}^{n} a_i V_{i(X)}\right) = \left(\sum_{i=0}^{n} a_i W_{i(X)}\right) + H(x)T(x).$$

The proof is complete.

§8.4.3 Optimizing the protocol

The protocol above can be optimized further. We didn't optimize it because we wanted it to be easier to understand.

In our protocol, the proof length is 6 group elements (Peggy sends Victor A, B, C, D, E, F), and Victor has to compute 8 elliptic curve pairings to verify the proof. Additionally, Victor has to do $O(\ell)$ group operations to compute D_0 depending on the public input.

It turns out that, by cleverly combining multiple verifications into one, you can get away with a proof length of just 3 group elements, and verifier work of just 3 elliptic curve pairings (plus the same $O(\ell)$ group operations).

Additionally, we didn't make the protocol zero-knowledge. This requires the addition of a blinding factor. Incredibly, Groth manages to take care of the blinding factor in the 3-element proof as well.

The fully optimized protocol is in Groth's paper.

§9 Lookups and cq

This is a summary of cq, a recent paper that tells how to do "lookup arguments" in a succint (or ZK) setting.

§9.1 What are lookups

Suppose you have two lists of items, $f_1, ..., f_n$ and $t_1, ..., t_N$. A "lookup argument" is an interactive proof protocol that lets Peggy prove the following claim, with a short proof:

Claim 9.1: Each of the f_i's is equal to (at least) one of the t_j's.

§9.1.1 Wait a second, who cares?

Here are a couple of applications of lookups which are very very important as part of practical ZK proof systems.

Example 9.2: Range checks.

Suppose you have some field element x, and you want to prove some claim like $0 \leqslant x < 2^{\{64\}}$. This sort of thing is surprisingly difficult in ZK, because we're working in a finite field, and there's no notion of "less than" or "greater than" in a finite field.

Kind of incredibly, one of the most competitive ways to prove this claim turns out to be a lookup argument: You simply write out a list of all the numbers $0, 1, ..., 2^{\{64\}} - 1$, and check that x is in this list.

There are lots of variants on this idea. For example, you could write x in base $2^{\{16\}}$ as x = x_0 + 2^{16} x_1 + 2^{32} x_2 + 2^{48} x_3, and run a lookup argument on each x_i to show that $0 \leqslant x_i < 2^{\{16\}}$.

Example 9.3: Function lookups.

Suppose you have some function f that is hard to compute in a circuit, and you want to prove that $y = f(x)$, where x and y are two numbers in your proof.

One way to do it is to precompute a table of all the pairs $(x, f(x))$, and then look up your specific value (x, y) in the table.

§9.1.2 Back to lookups

Let's notice a couple of features of these applications of the lookup problem.

First of all, let's call the values f_i the *sought* values, and the values t_j the *table* or the *index*. So the lookup problem is to prove that all the sought values can be found in the table.

- The values t_j are typically known to the verifier.
- The table $t_1, ..., t_N$ might be very long (i.e. N might be very big). In practice it's not uncommon to have a value of N in the millions.
- A single, fixed table of values t_j might be used in many, many proofs.

Conversely:

- The values f_i are usually secret, at least if the prover is concerned about zero knowledge.
- There may or may not be many values f_i (i.e. n might be big or small). If n is small, we'd like the prover runtime to be relatively small (roughly $O(n)$), even if N is big.
- The values f_i will be different in every proof.

As it turns out, lookups are a huge bottleneck in ZK proof systems, so an incredible amount of work has been done to optimize them. We're just going to talk about one system.

Remark 9.4: We will assume that the values f_i and t_j are all elements of some large finite field $\mathbb{F}_q$, and we will do algebra over this finite field.

In general, you might want to work with other values of other types than field elements. In the "function lookups" example (Example 9.3), we want to work with ordered pairs (of, say, field elements). In other contexts we might want to work with (suitably encoded) strings, or...

You can always solve this by hashing whatever type it is into a field element.

A second option (the "random linear combination" trick, we will see a lot of it) is to use a verifier challenge. In place of the pair (x_i, y_i), we will work with the single field element $x_i + ry_i$, where r is randomness provided by the verifier.

In more detail, imagine you have a list of pairs (x_i, y_i) that you want to run a lookup argument on. The prover sends two commitments, one to $x_1, ..., x_n$, and one to $y_1, ..., y_n$. The verifier responds with a challenge r, and then you run the lookup argument on the elements $x_i + ry_i$. (This is secure because the prover committed to both vectors before r was chosen.)

A similar trick works for tuples of arbitrary length: just use powers of r as the coefficients of your random linear combination.

In any case, we will only consider looking up field elements from here on out.

§9.2 cq

The name of the system "cq" stands for "cached quotients." For extra punniness, the authors of the paper note that "cq" is pronounced like "seek you."

Cq lookup arguments are based on the following bit of algebra:

Theorem 9.5: The lookup condition is satisfied (every f_i is equal to some t_j) if and only if there are field elements m_j such that

$$\sum_{i=1}^{n} \frac{1}{X - f_i} = \sum_{j=1}^{N} \frac{m_j}{X - t_j}$$

(as a formal equality of rational functions of the formal variable X).

Proof: If each f_i is equal to some t_j, it's easy. For each j, just take m_j to be the number of times t_j occurs among the f_i's, and the equality is obvious.

Conversely, if the equality of rational functions holds, then the rational function on the left will have a pole at each f_i. The rational function on the right can only have poles at the values t_j (of course, it may or may not have a pole at any given t_j, depending whether m_j is zero or not), so every f_i must be equal to some t_j. □

§9.2.1 Polynomial commitments for cq

Cq is going to rely on polynomial commitments. For concreteness, we'll work with KZG commitments (Section 5).

Cq will let the prover prove the lookup claim to the verifier...

- The proof (data sent from prover to verifier) will consist of $O(1)$ group elements. This includes a KZG commitment to the vector of sought values f_i.
- Given the index t_j, both prover and verifier will run a "setup" algorithm with $O(N \log N)$ runtime.
- Once the setup algorithm has been run, each lookup proof will require a three-round interactive protocol.
- The verifier work will be $O(1)$ calculations.
- Given the sought values f_i and the output of the setup algorithm, the prover will require $O(n)$ time to generate a proof.

Let's start with a brief outline of the protocol, and then we'll flesh it out.

- We're assuming both prover and verifier know the "index" values t_j. So they can both compute the KZG commitment to these values, and we'll assume this has been done once and for all before the protocol starts.
- The prover sends $\mathrm{Com}(F)$ and $\mathrm{Com}(M)$, KZG commitments to the polynomials F and M such that $F(\omega^i) = f_i$ for each $i = 1, ..., n$ and $M(\zeta^j) = m_j$ for each $j = 1, ..., N$. (Here ω is an nth root of unity, and ζ an Nth root of unity.)
- The verifier sends a random challenge β, which will substitute for X in the equality of rational functions (Theorem 9.5).
- The verifier chooses a random challenge β.
- The prover sends two more KZG commitments: a commitment $\mathrm{Com}(L)$ to the polynomial L such that

$$L(\omega^i) = \frac{1}{\beta - f_i},$$

  and another $\mathrm{Com}(R)$ to the polynomial R such that

$$R(\zeta^j) = \frac{m_j}{\beta - t_j}.$$

- The prover sends the value

$$s = \sum_{i=1}^{n} \frac{1}{X - f_i} = \sum_{j=1}^{N} \frac{m_j}{X - t_j}.$$

- Now the prover has to prove the following:
  - The polynomials L and R are well-formed. That is,

$$L(\omega^i)(\beta - F(\omega^i)) = 1$$

    for all $i = 1, ..., n$, and

$$R(\zeta^j)(\beta - T(\zeta^j)) = M(\zeta^j)$$

for all $j = 1, \ldots, N$.

- The value s is equal to both the sum of the l_i's and the sum of the r_j's.

The first claim is proven by a standard polynomial division trick: Asking that two polynomials agree on all powers of ω is the same as asking that they are congruent modulo $Z_{n(X)} = X^n - 1$. So the prover simply produces a KZG commitment to the quotient polynomial Q_L satisfying

$$L(x)(\beta - F(x)) = 1 + Q_L(X)Z_n(X).$$

And similarly for the claim involving R: the prover produces a KZG commitment to the polynomial Q_R such that

$$R(X)(\beta - T(X)) = M(X) + Q_R(X)Z_N(X).$$

Remark 9.6: The verifier can check the claim that $L(x)(\beta - F(x)) = 1 + Q_{L(X)}Z_{n(X)}$, and others like it, using the pairing trick.

(This is an example of the method explained in Example 4.5.)

The verifier already has access to KZG commitments $\mathrm{Com}(L)$, $\mathrm{Com}(F)$, $\mathrm{Com}(Q_L)$, and $\mathrm{Com}(Z_n)$, either because he can compute them himself ($\mathrm{Com}(Z_n)$), or because the prover sent them as part of the protocol ($\mathrm{Com}(L), \mathrm{Com}(F), \mathrm{Com}(Q_L)$). Additionally, the prover will need to send the intermediate value $\mathrm{Com}(Q_L Z_n)$, a KZG commitment to the product.

The verifier then checks the pairings

$$\mathrm{pair}(\mathrm{Com}(Q_L Z_n), [1]) = \mathrm{pair}(\mathrm{Com}(Q_L), \mathrm{Com}(Z_n))$$

(which verifies that the commitment $\mathrm{Com}(Q_L Z_n)$ to the product polynomial was computed honestly) and

$$\mathrm{pair}(\mathrm{Com}(L), [\beta] - \mathrm{Com}(F)) = \mathrm{pair}([1] + \mathrm{Com}(Q_L Z_n), [1])$$

(which verifies the claimed equality).

The process of verifing this sort of identity is quite general: The prover sends intermediate values as needed so that the verifier can verify the claim using only pairings and linearity.

The second claim is most easily verified by means of the following trick. If L is a polynomial of degree less than n, then

$$\sum_{i=0}^{n} L(\omega^i) = nL(0).$$

So the prover simply has to open the KZG commitment $\mathrm{Com}(L)$ at 0, showing that $nL(0) = s$ (and similarly for R).

§9.2.2 Cached quotients: improving the prover complexity

The protocol above works, and it does everything we want it to, except it's not clear how quickly the prover can generate the proof. To recall what we want:

- We're assuming $n \ll N$.
- Prover and verifier can both do a one-time $O(N)$ setup, depending on the lookup table T but not on the sought values F.

- After the one-time setup, the prover runtime (given the sought values F) should be only $O(n \log n)$.

The polynomial L has degree less than n – it is defined by Lagrange interpolation from its values at the nth roots of unity. So L can be computed quickly by a fast Fourier transform, and none of the identities involving L will give the prover any trouble.

But R is a bigger problem: it has degree N. So any calculation involving the coefficients of R – or of M, or the quotient Q_R – is a no-go.

So what saves us?
- The prover only ever needs to compute KZG commitments, not actual polynomials – and KZG commitments are linear.
- M, R and Q_R can be written as sums of only n terms (which can be precomputed once and for all).

Let's take R for example. R is the polynomial determined by Lagrange interpolation and the condition

$$R(\zeta^j)(\beta - T(\zeta^j)) = M(\zeta^j)$$

Let R_j be the polynomial (a mutiple of a Lagrange basis polynomial) such that

$$R_{j(\zeta^j)} = \frac{1}{\beta - t_j}$$

but

$$R_{j(\zeta^k)} = 0$$

for $k \neq j$. Then

$$R = \sum_j m_j R_j,$$

and the sum has at most n nonzero terms, one for each item on the sought list t_i. So the prover simply computes each commitment $\mathrm{Com}(R_j)$ in advance, and then given t_i, computes

$$\mathrm{Com}(R) = \sum_j m_j \,\mathrm{Com}(R_j).$$

A similar trick works for Q_R, which is the origin of the name "cached quotients." Recall that Q_R is defined by

$$R(X)(\beta - T(X)) = M(X) + Q_R(X)Z_N(X).$$

In other words, Q_R is the result of "division with remainder":

$$R(X)\frac{\beta - T(X)}{Z_N}(X) = Q_R(X) + \frac{M(X)}{Z_N}(X).$$

So the prover simply precomputes quotients Q_{R_j} and remainders M_j such that

$$R_j(X)\frac{\beta - T(X)}{Z_N}(X) = Q_{R_j}(X) + M_j\frac{X}{Z_N}(X),$$

and then computes Q_R and M as linear combinations of them.

So, in summary: The prover precomputes KZG commitments to R_j, Q_{R_j}, and M_j. Then prover and verifier run the protocol described above, and all the prover messages can be computed in $O(n \log n)$ time, using linear combinations of the cached precomputes.

Multi-party computation and garbled circuits

TODO: write this

§10 Oblivious transfer and multi-party computations

§10.1 Pitch

TODO: this whole section needs to be written

§10.2 How to do oblivious transfer

Suppose Alice has n keys, corresponding to elements $g_1, ..., g_n \in E$. Alice wants to send exactly one to Bob, and Bob can pick which one, but doesn't want Alice to know which one he picked. Here's how you do it:

1. Alice picks a secret scalar $a \in \mathbb{F}_q$ and sends $a \cdot g_1$, ..., $a \cdot g_n$.
2. Bob picks the index i corresponding to the key he wants and reads the value of $a \cdot g_i$, throwing away the other $n - 1$ values.
3. Bob picks a secret scalar $b \in \mathbb{F}_q$ and sends $b \cdot a \cdot g_i$ back.
4. Alice sends $\frac{1}{a} \cdot (b \cdot a \cdot g_i) = b \cdot g_i$ back to Bob.
5. Bob computes $\frac{1}{b} \cdot (b \cdot g_i) = g_i$.

§10.3 How to do 2-party AND computation

Suppose Alice and Bob have bits $a, b \in \{0, 1\}$. They'd like to compute $a \wedge b$ in such a way that if someone's bit was 0, they don't learn anything about the other person's bit. (Of course, if $a = 1$, then once Alice knows $a \wedge b$ then Alice knows b too, and this is inevitable.)

This is actually surprisingly easy. Alice knows there are only two cases for Bob, so she puts the value of $a \wedge 0$ into one envelope labeled "For Bob if $b = 0$", and the value of $a \wedge 1$ into another envelope labeled "For Bob if $b = 1$". Then she uses oblivious transfer to send one of them. Then Bob opens the envelope corresponding to the desired output. Repeat in the other direction.

Remark 10.1: Stupid use case I made up: Alice and Bob want to determine whether they have a mutual crush on each other. (Specifically, let $a = 1$ if Alice likes Bob and 0 otherwise; define b similarly.) Now we have a secure way to compute $a \wedge b$.

§10.4 Chaining circuits

Suppose now that instead of a single bit, Alice and Bob each have 1000 bits. They'd like to run a 2PC for function $f : \{0, 1\}^{2000} \to \{0, 1\}$ together.

The above protocol would work, but it would be really inefficient: it involves sending 2^{1000} envelopes each way.

However, in many real-life situations involving bits, the function f is actually given by a *circuit* with several AND, XOR, NOT gates or similar. So we'll try to improve the 2^{1000} down to something that grows only linearly in the number of gates in the circuit, rather than exponential in the input size.

Let $\oplus$ be binary XOR.

The idea is the following. A normal circuit has a bunch of registers, where the ith register just has a single bit x_i. We'd like to instead end up in a situation where we get a pair of bits (a_i, b_i) such that

$x_i = a_i \oplus b_i$, where Alice can see a_i but not b_i and vice-versa. This would let us do a 2PC for an arbitrarily complicated circuit.

It suffices to implement a single gate.

Lemma 10.2: Suppose $\diamondsuit : \{0,1\}^2 \to \{0,1\}$ is some fixed Boolean operator. Alice has two secret bits a_1 and a_2, while Bob has two secret bits b_1 and b_2. Then Alice and Bob can do use oblivious transfer to get a_3 and b_3 such that $a_3 \oplus b_3 = (a_1 \oplus b_1) \diamondsuit (a_2 \oplus b_2)$ without revealing a_3 or b_3 to each other.

Proof: Alice picks $a_3 \in \{0,1\}$ at random and prepares four envelopes for the four cases of (b_1, b_2) describing what Bob should set b_3 as. □

TODO: Don't do an OT every gate

Fully homomorphic encryption

WARNING: THESE ARE NOT POLISHED AT ALL AND MAY BE NONSENSE.

§11 FHE intro raw notes (April 16 raw notes from lecture)

TODO: raw notes based on Aard's lecture

Fully homomorphic encryption (FHE) refers to the idea that one can have some encrypted data and do circuit operations on this encrypted data (NAND gates, etc.).

§11.1 Outline

There are six pieces that go into this:

1. LWE (learning with errors): this was the red/blue tables from the retreat. Given a secret $\vec{a}$ and a bunch of vectors $\vec{v}$, and the values of $\langle \vec{a}, \vec{v} \rangle + \varepsilon(\vec{v})$ where there are errors $\varepsilon(v) \in \{0, 1\}$, determine $\vec{a}$.
2. How to public key cryptosystem out of LWE
3. The *approximate eignevector trick*
4. The *flatten trick*.

After the first four items, we get **somewhat homomorphic encryption**, which works but only up to a certain depth. (LWE always give small errors, and the errors compound as you do more operations.) To get from somewhat homomorphic encryption to FHE, we need:

5. Bootstrapping trick.

It turns out that LWE problems don't really care about the values of (q, n). So you can actually try to translate into a problem with smaller (q, n), which is called:

6. Dimension and modulus reduction

§11.2 Learning with errors (LWE)

Pick parameters q and n, and a "secret key" $\vec{s} \in \mathbb{F}_q^n$. I put secret key in quotes because we're not going to try to do cryptography just yet; instead, I'll invite you to guess $\vec{s}$.

Here's the puzzle. For lots of vectors $\vec{x} \in \mathbb{F}_q^n$, I will share with you

$$(\vec{x}, \vec{x} \cdot \vec{s} + \varepsilon)$$

where the "errors" ε are being sampled from some random distribution (e.g. Gaussian, doesn't really matter) and bounded with $|\varepsilon| \leq r$. At the retreat, we had $r = 1$.

I'm happy to give you as many of these perturbed dot products as you like. The challenge is to determine $\vec{s}$.[2]

Assume LWE is a "hard" problem, i.e. we can't easily recover $\vec{s}$ in this challenge. (LWE might be hard even when $q = 2$, if ε is say 90% zero and 10% one.)

§11.2.1 Attacks on LWE

[2] Technically, at the retreat we did a decision problem, but it's no different. Yan has an write-up at https://hackmd.io/F1vjMWzhTk-J7u-ctWQIIQ of the red and blue example.

- Brute force ε: once you have n vectors, guess all $(2r+1)^n$ possibilities of ε and invert the matrices, and see if they work with other vectors.
- Find linear dependencies among the $\vec{x}_i$.

Turns out all the naive algorithms still end up being exponential in n.

Remark 11.1: Lattice-based cryptography uses "find the shortest vector in a lattice" as a hard problem, and it turns out LWE can be reduced to this problem. So this provides reasons to believe LWE is hard.

§11.3 Building a public-key system out of LWE

Here's how to build a public-key system.

The public key will consist of m ordered pairs

$$\vec{x}_i, \quad y_i := \vec{x}_i \cdot \vec{s} + \varepsilon_i$$

where $m \approx 2n \log q = 2\log(q^n)$ and ε_i are small errors. We'll assume roughly the size constraints $n^2 \le q \le 2^{\sqrt{n}}$.

Now, how can I submit a single bit? Suppose Bob wants to send a single bit to Alice. The idea is to take $\vec{x}$ as a "not-too-big" random linear combination, like say

$$\vec{x} := 2\vec{x}_1 + \vec{x}_7 + \vec{x}_9$$

so that Bob can compute

$$y := 2y_1 + y_7 + y_9 = \vec{x} \cdot \vec{s} + \text{up to 4 errors}.$$

Then Bob sends $\vec{x}$ and then *either* $y + \left\lfloor \frac{q}{2} \right\rfloor$ or y.

Since Alice knows the true value of $\vec{x} \cdot \vec{s}$, she can figure out whether y or $y + \left\lfloor \frac{q}{2} \right\rfloor$ was sent.

So m needs to be big enough that $\vec{x}$ could be almost anything even with "not-too-big" coefficients, but small enough LWE is still hard.

§11.4 Building homomorphic encryption on top of this: approximate

Addition is actually just addition (as long as you don't add too many things and cause the errors to overflow). But multiplication needs a new idea, the approximate eigenvalue trick.

Suppose Bob has a message $\mu \in \{0,1\} \subset \mathbb{F}_q$. Our goal is to construct a matrix C such that $C\vec{s} \approx \mu\vec{s}$, which we can send as a ciphertext.

Algorithm 11.2:
1. Use the public key to find n "almost orthogonal" vectors $\vec{c}_i$, meaning $\vec{c}_i \cdot \vec{s}$ is small.
2. If $\mu = 0$, use C as the matrix whose rows are $\vec{c}_i$.
3. If $\mu = 1$, take the matrix C from Step 2 and then increase its diagonal by $\left\lfloor \frac{q}{2} \right\rfloor$.

The NOT gate is $1 - C$.

TODO: something about subset sum?

TODO: I think we can commit to $\vec{s}$ having first component 1 for convenience because the augmentation thing is terrible notation-wise

If we can do this, then multiplication can be done by multiplying the matrices.

TODO: Use NOT, AND gates

§12 FHE intro raw notes continued (April 23 raw notes from lecture)

§12.1 Recap

To paraphrase last week's setup:

- Our secret key is a vector $v = (1, \dots)$ of length $n + 1$.
- Our public key is about $2\log(q^n) = 2n \log q$ vectors a such that $a \cdot v \approx 0$.

Remark 12.1: I think we don't even really care whether q is prime or not, if we work in $\mathbb{Z}/q\mathbb{Z}$.

Our idea was that given a message $\mu \in \{0, 1\}$, we construct a matrix C such that $Cv \approx \mu v$; the matrix C is the ciphertext. And we can apply NOT by just looking at $\mathrm{id} - C$.

Then given the equations

$$C_1 v = \mu_1 v + \varepsilon_1$$
$$C_2 v = \mu_2 v + \varepsilon_2$$

the multiplication goes like

$$C_1 C_2 v = \mu_1 \mu_2 v + (\mu_2 \varepsilon_1 + C_1 \varepsilon_2).$$

In the new error term, $\mu_2 \varepsilon_1$ is still small, but $C_1 \varepsilon_2$ is not obviously bounded, because we don't have constraints on the entries of C_1.

§12.2 Flatten

So our goal is to modify our scheme so that:

Goal 12.2: We want to change our protocol so that C only has zero-one entries.

To do this, we are going to demand our secret key v has a specific form: it must be

$$\begin{aligned} v = \big(&1, 2, 4, ..., 2^\ell, \\ &a_1, 2a_1, 4a_1, ..., 2^\ell a_1, \\ &a_2, 2a_2, 4a_2, ..., 2^\ell a_2, \\ &..., \\ &a_p, 2a_p, 4a_p, ..., 2^\ell a_p\big) \end{aligned}$$

where $\ell \approx \log(q)$, where $p := \frac{n}{\ell} - 1$. (This means we really have security parameters (p, q) rather than (n, q).)

TODO: I think it's better to use n and N instead of p and n. Also, maybe we should just always use $\vec{s}$ for the secret key vector? v feels too generic for something that never changes lol.

Proposition 12.3: There exists a map $\text{Flatten} : \mathbb{F}_q^n \to \mathbb{F}_q^n$ taking row vectors of length n such that:

- $\text{Flatten}(r)$ has all entries either 0 or 1
- $\text{Flatten}(r) \cdot v = r \cdot v$ for any v in nice binary form.

We can then extend Flatten to work on $n \times n$ matrices, by just flattening each of the rows.

When we add this additional assumption, we have a natural map

$$C \mapsto \text{Flatten}(C)$$

on matrices that forces them to have 0/1.

Example 12.4: For concreteness, let's write out an example $\ell = 3$ and $p = 2$, so

$$v = (1, 2, 4, 8, a_1, 2a_1, 4a_1, 8a_1, a_2, 2a_2, 4a_2, 8a_2).$$

Let $q = 13$ for concreteness. Suppose the first row of C is

$$r := (3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8).$$

Then, we compute the dot product

$$r \cdot v = 29 + 79a_1 + 95a_2 = 3 + 11a_1 + 4a_2 \pmod{13}.$$

Now, how can we get a 0-1 vector? The answer is to just use binary: we can use

$$\text{Flatten}(r) = (1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0)$$

because

$$\text{Flatten}(r) \cdot v = (1 + 2) + (1 + 2 + 8)a_1 + 4a_2.$$

Repeating this for however many rows of the matrix you need

So, we can flatten any ciphertexts C we get. Now if C_1 and C_2 are flattened already, if we multiply C_1 and C_2, we don't get a zero-one matrix; but we can just use $\text{Flatten}(C_1 C_2)$ instead.

§12.3 Going from somewhat homomorphic encryption to fully homomorphic encryption

After a while, our errors are getting big enough that we have trouble. So we show how to go from somewhat homomorphic encryption to FHE.

This is the jankiest thing ever:

- So, suppose we're stuck at $\text{Enc}_{\mathsf{pk}}(x)$.
- Generate another pair $(\mathsf{pk}', \mathsf{sk}')$.
- Send $\text{Enc}_{\mathsf{pk}'}(\mathsf{sk})$.
- We could compute $\text{Enc}_{\mathsf{pk}'}\left(\text{Enc}_{\mathsf{pk}}(x)\right)$.
- The decryption is itself a circuit of absolute constant length. If we picked $(\mathsf{pk}, \mathsf{pk}')$ well, we should be able to get $\text{Enc}_{\mathsf{pk}'}(x)$.

So we need a way to convert $\text{Enc}_{\mathsf{pk}'}(x)$ back to $\text{Enc}_{\mathsf{pk}}(x)$.

Let's assume for now $q = q'$. So let's say we have a secret key $v \in \mathbb{F}_q^n$ and $v' \in \mathbb{F}_q^{n'}$. Our goal is to provide an almost linear map

$$F = F_{v,v'} : \mathbb{F}_q^n \to \mathbb{F}_q^{n'}$$

that given an arbitrary vector $a \in \mathbb{F}_q^n$, gives $F(a) \in \mathbb{F}_q^{n'}$ such that

$$F(a) \cdot v' \approx a \cdot v.$$

TODO: Something seems fishy here: we were about to provide F at all values of $(0, \ldots, 02^i, 0 \ldots, 0)$, but this seems like too much information. Defer to next meeting.

In the case where $q' \neq q$, modulus reduction is done by

$$\frac{F(a) \cdot v'}{q'} \approx \frac{a \cdot v}{q}.$$

§13 Ring SHE raw notes (May 2 raw notes from lecture)

This is how to set up a somewhat-homomorphic-encryption scheme using rings.

§13.1 A non-working scheme over $\mathbb{Z}$

Here is a non-working example (in the sense that it is obviously insecure).

We're going to have a small modulus, say $t = 2$, and a big odd modulus q coprime to t. The "ciphertexts" will be elements of $\mathbb{Z}$, but we'll think of them as elements of $\mathbb{Z}/q\mathbb{Z}$. Messages will be elements of a $\frac{\mathbb{Z}}{t}\mathbb{Z}$.

So to encrypt a bit $m \in \{0, 1\}$, we'll choose a small-ish integer $c_0 \equiv m \bmod t$ with $|c_0| \ll q$. Then the ciphertext will be an integer c with $c \equiv c_0 \bmod q$.

There's no security here yet: if you see c, you can get c_0 by just computing $c \bmod q$. But that's just because it's easy to take integers modulo q.

Now imagine q is a secret key and the public key is integers q_1 and q_2 with $\gcd(q_1, q_2) = q$. So again, to encrypt c_0, then we just add on random multiples of q_1 and q_2. This still isn't secure because we have Euclidean algorithm.

But the idea is that instead of using $\mathbb{Z}$, we can use a number field. In a general number field, we might *not* have the Euclidean algorithm anymore. That's the trick that makes this scheme go from trivially breakable to hard-to-break.

§13.2 Gaussian integers

For concreteness, for example, let's take the Gaussian integers $\mathbb{Z}[i]$ and the ideal $(6+i)$. The ideal $(6+i)$ is this lattice generated by $6+i$ and $i(6+i) = -1+6i$.

We'll have $6+i$ take the role of q; we could imagine taking a fundamental domain of this, so there is a canonical representation of each Gaussian integer. To encrypt a message, you pick an element of the fundamental domain that's even or odd (say, by taking modulo $1+i$), and then add a multiple of q.

Like before, q is the secret key; the public key is instead two Gaussian integers q_1 and q_2. We'll actually require

$$\begin{aligned} q_1 &\in \mathbb{Z} \\ q_2 &\in \mathbb{Z} + i \end{aligned}$$

which you can do using elimination: starting with the basis vectors $10+7i$ and $-7+10i$, you take some linear combination of these that gets you the part you want, say

$$\begin{aligned} 10(10+7i) - 7(-7+10i) &= 149 \\ 2(10+7i) - 3(-7+10i) &= 41 + i. \end{aligned}$$

Well, computing the GCD of two Gaussian integers is still pretty easy. But it gets harder quickly in the degree of the number field, *really quickly*.

§13.3 General procedure

To show how to do this for larger degree number fields, let's take ζ to be a primitive 8th root of unity. Hence we're working in the ring

$$\frac{\mathbb{Z}[\zeta]}{\zeta^4 + 1}$$

and the resulting ring of integers is a rank four $\mathbb{Z}$-module.

We choose something like $q = 1 + 3\zeta + 3\zeta^2 + 7\zeta^3$, for example. And then we find the smallest integers n_0, n_1, n_2, n_3 such that the four numbers

$$\langle n_0, n_1 + \zeta, n_2 + \zeta^2, n_3 + \zeta^3 \rangle$$

are multiples of q, hence with gcd q.

You don't need all four actually. Just two is enough. e.g. if you have 179 and $\zeta - 82$, then n_2 will be $82^2 \bmod 179$ or something like this.

§13.4 Relation to LWE

$$\langle 179, \zeta - 82, \zeta^2 - 17, \zeta^3 - 33 \rangle$$

looks a lot like LWE:

$$82a + 17b + 33c = d \bmod 179$$

with a, b, c, d small. Kind of? Need to think more about how to make this fleshed out.

§13.5 Somewhat homomorphic

Just add and multiply directly, lol.

Appendix: Classical PCP

Historically, the construction of the first PCP, or **Probabilistically Checkable Proof**, was sort of an ancestor to the zkSNARK. There are a few nice ideas in here, but they're actually more complicated than the zkSNARK and hence included here mostly for historical reference.

Pedagogically, we think it makes sense to just jump straight into PLONK and Groth16 even though the PCP construction came first. The more modern zkSNARK protocols are both better (according to metrics like message length or verifier complexity) and simpler (fewer moving parts).

This part is divided into two sections.

- Section 14 describes the sum-check protocol, which is actually useful a bit more generally and shows up in some other SNARK constructions besides the PLONK and Groth16 that we covered.
- Section 15 gives an overview of the first PCP constructions, but it's quite involved and much less enlightening. It's mostly here for historical reasons and not otherwise worth reading.

Elliptic curves will not be used in this part at all; in fact, these two chapters are self-contained and don't depend on earlier parts of these lecture notes.

§14 The sum-check protocol

§14.1 Pitch: Sum-check lets you prove a calculation without having the verifier redo it

Let's imagine we have a *single* equation

$$Z_1 + Z_2 + \cdots + Z_{\text{big}} = H$$

for some variables Z_i and constant $H \in \mathbb{F}_q$, all over $\mathbb{F}_q$, and assume further that q is not too small.

Imagine a prover Peggy has a value assigned to each Z_i, and is asserting to the verifier Victor they Z_i's sum to H. Victor wants to know that Peggy computed the sum H correctly, but Victor doesn't want to actually read all the values of Z_i.

Well, at face value, this is an obviously impossible task. Even if Victor knew all but one of Peggy's Z_i's, that wouldn't be good enough. Nevertheless, the goal of Sum-Check is to show that, with only a little bit of extra structure, this is actually possible.

§14.1.1 An oracle to a polynomial

Assume for convenience that the number of Z's happens to be 2^n and change notation to a function $f : \{0,1\}^n \to \mathbb{F}_q$, so our equation becomes

$$\sum_{\vec{v} \in \{0,1\}^n} f(\vec{v}) = H$$

. In other words, we have changed notation so that our variables are indexed over a hypercube: from $f(0, ..., 0)$ to $f(1, ..., 1)$.

But suppose that the values of f coincide with a polynomial $P \in \mathbb{F}_q[X_1, ..., X_n]$ of degree at most d in each variable.

Theorem 14.1 (Sum-check): There's an interactive protocol that allows Peggy to convince Victor that the value H above is the sum, which takes

- n rounds of communication, where each message from Peggy is a single-polynomial of degree at most d;
- For Victor, only a single evaluation of the polynomial P at some point (not necessarily in $\{0,1\}^n$).

In other words, Sum-Check becomes possible if Victor can make *one* call to an oracle that can tell Victor the value of $P(r_1, ..., r_n)$, for one random choice of $(r_1, ..., r_n) \in \mathbb{F}_q^n$. Note importantly that the r_i's do not have to 0 or 1; Victor chooses them randomly from the much larger $\mathbb{F}_q$. But he can only ask the oracle for that single value of P, and otherwise has no idea what any of the Z_i's are.

This is a vast improvement from the case where Victor had to evaluate P at 2^n points and add them all together.

§14.1.2 Comment on polynomial interpolation

The assumption that f coincides with a low-degree polynomial might seem stringent *a priori*. However, the truth is that *every* function $f : \{0,1\}^n \to \mathbb{F}_q$ can be expressed as a *multilinear* polynomial! (In other words, we can take $d = 1$ in the above theorem.)

The reason is just polynomial interpolation. For example, suppose $n = 3$ and the eight (arbitrary) variable values were given

$$\begin{aligned} f(0,0,0) &= 8 \\ f(0,0,1) &= 15 \\ f(0,1,0) &= 8 \\ f(0,1,1) &= 15 \\ f(1,0,0) &= 8 \\ f(1,0,1) &= 15 \\ f(1,1,0) &= 17 \\ f(1,1,1) &= 29. \end{aligned}$$

(So $H = 8 + 15 + 8 + 15 + 8 + 15 + 17 + 29 = 115$.) Then we'd be trying to fill in the blanks in the equation

$$P(x,y,z) = \square + \square x + \square y + \square z + \square xy + \square yz + \square zx + \square xyz$$

so that P agrees with f on the cube. This comes down to solving a system of linear equations; in this case it turns out that $P(x,y,z) = 5xyz + 9xy + 7z + 8$ works, and I've cherry-picked the numbers so a lot of the coefficients work out to 0 for convenience, but math majors should be able to verify that P exists and is unique no matter what eight initial numbers I would have picked (by induction on n).

Earlier, we commented that Sum-Check was an "obviously impossible task" if the values of f's were unrelated random numbers. The reason this doesn't contradict the above paragraph is that, if Peggy just sends Victor the table of 2^n values, it would be just as much work for Victor to manually compute P. However, in a lot of contexts, the values that are being summed can be construed as a polynomial in some way, and then the sum-check protocol will give us an advantage. We'll show two example applications at the end of the section.

§14.2 Description of the sum-check protocol

The author's opinion is that it's actually easier to see a specific example for $n = 3$ before specifying the pseudocode in general, rather than vice-versa.

§14.2.1 A playthrough of the sum-check protocol

Let's use the example above with $n = 3$: Peggy has chosen the eight values above with $H = 115$, and wants to convince Victor without actually sending all eight values. Peggy has done her homework and computed the coefficients of P as well (after all, she chose the values of f), so Peggy can evaluate P anywhere she wants. Victor is given oracle access to a single value of the polynomial P on a point (probably) outside the hypercube.

Here's how they do it. (All the information sent by Peggy to Victor is $\boxed{\text{boxed}}$.)

1. Peggy announces her claim $H = \boxed{115}$.
2. They now discuss the first coordinate:
   - Victor asks Peggy to evaluate the linear one-variable polynomial

     $$g_1(T) := P(T,0,0) + P(T,0,1) + P(T,1,0) + P(T,1,1)$$

     and send the result. In our example, it equals

$$g_1(T) = 8 + 15 + (9T + 8) + (14T + 15) = \boxed{23T + 46}.$$

- Victor then checks that this g_1 is consistent with the claim $H = 115$; it should satisfy $H = g_1(0) + g_1(1)$ by definition. Indeed, $g_1(0) + g_1(1) = 46 + 69 = 115 = H$.
- Finally, Victor commits to a random choice of $r_1 \in \mathbb{F}_q$; let's say $r_1 = 7$. From now on, he'll always use 7 for the first argument to P.

3. With the first coordinate fixed at $r_1 = 7$, they talk about the second coordinate:
   - Victor asks Peggy to evaluate the linear polynomial
   $$g_2(U) := P(7, U, 0) + P(7, U, 1).$$
   and send the result. In our example, it equals
   $$g_2(U) = (63U + 8) + (98U + 15) = \boxed{161U + 23}.$$
   - Victor makes sure the claimed g_2 is consistent with g_1; it should satisfy $g_1(r_1) = g_2(0) + g_2(1)$. Indeed, it does $g_1(7) = 23 \cdot 7 + 46 = 23 + 184 = g_2(0) + g_2(1)$.
   - Finally, Victor commits to a random choice of $r_2 \in \mathbb{F}_q$; let's say $r_1 = 3$. From now on, he'll always use 3 for the second argument to P.
4. They now settle the last coordinate:
   - Victor asks Peggy to evaluate the linear polynomial
   $$g_3(V) := P(7, 3, V)$$
   and send the result. In our example, it equals
   $$g_3(V) = \boxed{112V + 197}.$$
   - Victor makes sure the claimed g_3 is consistent with g_2; it should satisfy $g_2(r_2) = g_3(0) + g_3(1)$. Indeed, it does $g_2(3) = 161 \cdot 3 + 23 = 197 + 309 = g_3(0) + g_3(1)$.
   - Finally, Victor commits to a random choice of $r_3 \in \mathbb{F}_q$; let's say $r_3 = -1$.
5. Victor has picked all three coordinates, and is ready consults the oracle. He gets $P(7, 3, -1) = 85$. This matches $g_3(-1) = 85$, and the protocol ends.

§14.2.2 General procedure

The previous transcript should generalize obviously to any $n > 3$, but we spell it out anyways. Peggy has already announced H and pre-computed P. Now for $i = 1, ..., n$,

- Victor asks Peggy to compute the univariate polynomial g_i corresponding to partial sum, where the ith parameter is a free parameter while all the $r_1, ..., r_{i-1}$ have been fixed already.
- Victor sanity-checks each of Peggy's answer by making sure g_i is consistent with (that is, $g_{i-1}(r_{i-1}) = g_i(0) + g_i(1)$, or for the edge case $i = 1$ that $H = g_1(0) + g_1(1)$).
- Then Victor commits to a random $r_i \in \mathbb{F}_q$ and moves on to the next coordinate.

Once Victor has decided on every r_i, he asks the oracle for $P(r_1, ..., r_n)$ and makes sure that it matches the value of $g_n(r_n)$. If so, Victor believes Peggy.

Up until now, we wrote the sum-check protocol as a sum over $\{0, 1\}^n$. However, actually there is nothing in particular special about $\{0, 1\}^n$ and it would work equally well with $\mathbb{H}^n$ for any small fi-

nite set $\mathbb{H}$; the only change is that the polynomial P would now have degree at most $|\mathbb{H}| - 1$ in each variable, rather than being multilinear. Accordingly, the g_i's change from being linear to up to degree $|\mathbb{H}| - 1$. Everything else stays the same.

§14.2.3 Soundness

TODO: Can Peggy cheat?

§14.3 Two simple applications of sum-check

If you're trying to sum-check a bunch of truly arbitrary unrelated numbers, and you don't have an oracle, then naturally it's a lost cause. You can't just interpolate P through your 2^n numbers as a "manual oracle", because the work of interpolating the polynomial is just as expensive.

However, in real life, sum-check gives you leverage because of the ambient context giving us a way to rope in polynomials. We'll give two examples below.

§14.3.1 Verifying a triangle count

This example was communicated to us by Justin Thaler. Suppose Peggy and Victor have a finite simple graph $G = (V, E)$ on n vertices and want to count the number of triangles in it. Peggy has done the count, and wants to convince a lazy verifier Victor who doesn't want to spend the $O(n^3)$ time it would take to count it himself.

Proposition 14.2: It's possible for Peggy to prove her count of the number of triangles is correct with only $O(n^2 \log n)$ work for Victor.

Note that Victor will always need at least $O(n^2)$ time because he needs to read the input graph G, so this is pretty good.

Proof: Assume for convenience $n = 2^m$ and biject V to $\{0,1\}^m$. Both parties then compute the coefficients of the multilinear function $g : \{0,1\}^{2m} \to \{0,1\}$ defined by

$$g(x_1, ..., x_m, y_1, ..., y_m) = \begin{cases} 1 \text{ if } (x_1, ..., x_m) \text{ has an edge to } (y_1, ..., y_m) \\ 0 \text{ otherwise.} \end{cases}$$

In general, this interpolation calculation takes $O(2^{2m} \cdot 2m) = O(n^2 \log n)$ time.

Once this is done, they set

$$f(\vec{x}, \vec{y}, \vec{z}) := g(\vec{x}, \vec{y})g(\vec{y}, \vec{z})g(\vec{z}, \vec{x}).$$

they can just run the Sum-Check protocol on:

$$\text{number triangles} = \sum_{\vec{x} \in \{0,1\}^m} \sum_{\vec{y} \in \{0,1\}^m} \sum_{\vec{z} \in \{0,1\}^m} f(\vec{x}, \vec{y}, \vec{z})$$

This requires some work from Peggy, but for Victor, the steps in between don't require much work. The final oracle call requires Victor to evaluate

$$g(\vec{x}, \vec{y})g(\vec{y}, \vec{z})g(\vec{z}, \vec{x})$$

for one random choice $(\vec{x}, \vec{y}, \vec{z}) \in \left(\mathbb{F}_p^m\right)^{\times 3}$. Victor can do this because he's already computed all the coefficients of g. □

Remark 14.3: Note that Victor does NOT need to compute f as a polynomial, which is much more work. Victor does need to compute coefficients of g so that it can be evaluated at three points. But then Victor just multiplies those three numbers together.

You could in principle check for counts of any more complicated subgraph as opposed to just the triangle K_3. Just modify the indicator function above.

§14.3.2 Verifying a polynomial vanishes

Suppose $f(T_1, ..., T_n) \in \mathbb{F}_q[T_1, ..., T_n]$ is a polynomial of degree up to 2 in each variable, specified by the coefficients. Now Peggy wants to convince Victor that $f(x_1, ..., x_n) = 0$ whenever $x_i \in \{0, 1\}$.

Of course, Victor could verify this himself by plugging in all 2^n pairs. Because f is the sum of 3^n terms, this takes about 6^n operations. We'd like to get this down. Here's one toy example of how we could do this.

Proposition 14.4: Peggy can convince Victor that $f(x_1, ..., x_n) = 0$ for all $x_i \in \{0, 1\}$ with only a single evaluation to f and a single evaluation to a random multilinear polynomial.

Proof: The idea is to take a random linear combination of the 2^n values. Specifically, Victor picks a multilinear polynomial $g(T_1, ..., T_n) \in \mathbb{F}_q[T_1, ..., T_n]$ coefficient by coefficient out of the q^{2^n} possible multilinear polynomials. Note that this is equivalent to picking the 2^n values of $g(x_1, ..., x_n)$ for $(x_1, ..., x_n) \in \{0, 1\}^n$ uniformly at random. Then we run sum-check to prove that

$$0 = \sum_{\vec{x} \in \{0,1\}^n} f(x_1, ..., x_n) g(x_1, ..., x_n)$$

The polynomial fg is degree up to 3 in each variable, so that's fine. The final "oracle" call is then straightforward, because the coefficients of both f and g are known. □

This takes only $3^n + 2^n$ operations (i.e. one evaluates two polynomials each at one point, rather than 2^n evaluations).

TODO: improvement with eq_w, mention context in which this comes up

§15 The classical PCP protocol

This entire chapter describes the first construction to PCP's. You can think of it as "really long example of sum-check application".

Notational change: in this section, q is a known prime, but it is not on the order of 2^{256}. In fact, as discussed in the later Remark 15.1, we actually prefer q to be pretty small.

§15.1 Pitch: How to grade papers without reading them

Imagine that Peggy has a long proof of some mathematical theorem, and Victor wants to verify it. *A priori*, this would require Victor to actually read every line of the proof. As anyone who has experience grading students knows, that's a whole lot of work. A lazy grader might try to save time by only "spot checking" a small number of lines of the proof. However, it would be really easy to cheat such a grader: all you have to do is make a single wrong step somewhere and hope that particular step is not one of the ones examined by Victor.

A probabilistically checkable proof is a way to try to circumvent this issue. It provides a protocol where Peggy can format her proof such that Victor can get high-probability confidence of its confidence by only checking K bits of the proof at random, for some absolute constant K. This result is one part of the *PCP theorem* which is super famous.

The section hopes to give a high level summary of the ideas that go into the protocol and convince you that the result is possible, because at first glance it seems absurd that a single universal constant K is going to be enough. Our toy protocol uses the following parts:

- Sum-check from the previous chapter.
- Low-degree testing theorem, which we'll just state the result but not prove.
- Statement of the Quad-SAT problem, which is the NP-complete problem which we'll be using for this post: an instance of Quad-SAT is a bunch of quadratic equations in multiple variables that one wants to find a solution to.

We'll staple everything to get a toy PCP protocol. Finally, we'll give an improvement to it using error-correcting codes.

§15.2 Low-degree testing

In the general sum-check protocol we just described, we have what looks like a pretty neat procedure, but the elephant in the room is that we needed to make a call to a polynomial oracle. Just one call, which we denoted $P(r_1, \ldots, r_n)$, but you need an oracle nonetheless.

When we do PCP, the eventual plan is to replace both the oracle call and Peggy's answers with a printed phone book. The phone book contains, among other things, a big table mapping every n-tuple $(r_1, \ldots, r_n) \in \mathbb{F}_q^n$ to its value $P(r_1, \ldots, r_n)$ that Peggy mails to Victor via Fedex or whatever. This is of course a lot of work for Peggy to print and mail this phone book. However, Victor doesn't care about Peggy's shipping costs. After all, it's not like he's *reading* the phone book; he's just checking the entries he needs. (That's sort of the point of a phone book, right?)

But now there's a new issue. How can we trust the entries of the phone book are legitimate? After all, Peggy is the one putting it together. If Peggy was trying to fool Victor, she could write whatever she wanted ("hey Victor, the value of P is 42 at every input") and then just lie during the sum-check protocol. After all, she knows Victor isn't actually going to read the whole phone book.

§15.2.1 Goal of low-degree testing

The answer to this turns out be *low-degree testing*. For example, in the case where $|\mathbb{H}| = 2$ we described earlier, there is a promise that P is supposed a multilinear polynomial. For example, this means that

$$\frac{5}{8}P(100, r_2, ..., r_n) + \frac{3}{8}P(900, r_2, ..., r_n) = P(400, r_2, ..., r_n)$$

should be true. If Victor randomly samples equations like this, and they all check out, he can be confident that P is probably "mostly" a polynomial.

I say "mostly" because, well, there's no way to verify the whole phone book. By definition, Victor is trying to avoid reading. Imagine if Peggy makes a typo somewhere in the phone book — well, there's no way to notice it, because that entry will never see daylight. However, Victor *also* doesn't care about occasional typos in the phone book. For his purposes, he just wants to check the phone book is 99% accurate, since the sum-check protocol only needs to read a few entries anyhow.

§15.2.2 The procedure — the line-versus-point test

This is now a self-contained math problem, so I'll just write the statement and not prove it (the proof is quite difficult). Suppose $g : \mathbb{F}_q^m \to \mathbb{F}_q$ is a function and we want to see whether or not it's a polynomial of total degree at most d.

The procedure goes as follows. The prover prints out two additional posters containing large tables B_0 and B_1, whose contents are defined as follows:

- In the table B_0, for each point $\vec{b} \in \mathbb{F}_q^m$, the prover writes $g\left(\vec{b}\right) \in \mathbb{F}_q$. We denote the entry of the table by $B_0\left[\vec{b}\right]$.
- In the table B_1, for each line

  $$\ell = \left\{\vec{b} + t\vec{m} \mid t \in \mathbb{F}_q\right\}$$

  the prover writes the restriction of g to the line ℓ, which is a single-variable polynomial of degree at most d in $\mathbb{F}_q[t]$. We denote the entry of the table by $B_1[\ell]$.

The verifier then does the obvious thing: pick a random point $\vec{b}$, a random line ℓ through it, and check the tables B_0 and B_1 are consistent.

This simple test turns out to be good enough, though proving this is hard and requires a lot of math. But the statement of the theorem is simple:

§15.3 Quad-SAT

As all NP-complete problems are equivalent, we can pick any one which is convenient. Systems of linear equations don't make for good NP-complete problems, but quadratic equations do, as we saw in Section 7.2. So we are going to use Quad-SAT, in which one has a bunch of variables over a finite field $\mathbb{F}_q$, a bunch of polynomial equations in these variables of degree at most two, and one wishes to find a satisfying assignment.

Let's say there are N variables and E equations, and N and E are both large. Peggy has worked really hard and figured out a satisfying assignment

$$A : \{x_1, ..., x_N\} \to \mathbb{F}_q$$

and wants to convince Victor she has this A. Victor really hates reading, so Victor neither wants to read all N values of the x_i nor plug them into each of the E equations. He's fine receiving lots of stuff in the mail; he just doesn't want to read it.

Remark 15.1 (q is not too big): In earlier chapters, q was usually a large prime like $q \approx 2^{255}$. This is actually not desirable here: Quad-SAT is already interesting even when $q = 2$. So in this chapter, large q is a bug and not a feature.

For our protocol to work, we do need q to be modestly large, but its size will turn out to be around $(\log NE)^{O(1)}$.

§15.4 Description of the toy PCP protocol for Quad-SAT

We now have enough tools to describe a quad-SAT protocol that will break the hearts of Fedex drivers everywhere. In summary, the overview of this protocol is going to be the following:

- Peggy prints q^E phone books, one phone book each for each linear combination of the given Q-SAT equations. We'll describe the details of the phone book contents later.
- Peggy additionally prints the two posters corresponding to a low-degree polynomial extension of A (we describe this exactly in the next section).
- Victor picks a random phone book and runs sum-check on it.
- Victor runs a low-degree test on the posters.
- Victor makes sure that the phone book value he read is consistent with the posters.

Let's dive in.

§15.4.1 Setup

In sum-check, we saw we needed a bijection of $[N]$ into $\mathbb{H}^m$. So let's fix this notation now (it is annoying, I'm sorry). We'll let $\mathbb{H}$ be a set of size $|\mathbb{H}| := \log(N)$ and set $m = \log_{|\mathbb{H}|} N$. This means we have a bijection from $\{1, \ldots, N\} \to \mathbb{H}^m$, so we can rewrite the type-signature of A to be

$$A : \mathbb{H}^m \to \mathbb{F}_q.$$

The contents of the phone books will take us a while to describe, but we can actually describe the posters right now, and we'll do so. Earlier when describing sum-check, we alluded to the following theorem, but we'll state it explicitly now:

Theorem 15.2: Suppose $\varphi : \mathbb{H}^n \to \mathbb{F}_q$ is *any* function. Then there exists a unique polynomial $\tilde{\varphi} : \mathbb{F}_q^n \to \mathbb{F}_q$, which agrees with φ on the values of $\mathbb{H}^n$ and has degree at most $|\mathbb{H}| + 1$ in each coordinate. Moreover, this polynomial $\tilde{\varphi}$ can be easily computed given the values of φ.

Proof: Lagrange interpolation and induction on m. □

We saw this earlier in the special case $\mathbb{H} = \{0, 1\}$ and $n = 3$, where we constructed the multilinear polynomial $5xyz + 9xy + 7z + 8$ out of eight initial values.

In any case, the posters are generated as follows. Peggy takes her known assignment $A : \mathbb{H}^m \to \mathbb{F}_q$ and extends it to a polynomial

$$\tilde{A} \in \mathbb{F}_q[T_1, ..., T_m]$$

using the above theorem; by abuse of notation, we'll also write $\tilde{A} : \mathbb{F}_q^m \to \mathbb{F}_q$. She then prints the two posters we described earlier for the point-versus-line test.

§15.4.2 Taking a random linear combination

The first step of the reduction is to try and generate just a single equation to check, rather than have to check all of them. There is a straightforward (but inefficient; we'll improve it later) way to do this: take a *random* linear combination of the equations (there are q^E possible combinations).

To be really verbose, if $\mathcal{E}_1$, ..., $\mathcal{E}_E$ were the equations, Victor picks random weights λ_1, ..., λ_E in $\mathbb{F}_q$ and takes the equation $\lambda_1\mathcal{E}_1 + ... + \lambda_E\mathcal{E}_E$. In fact, imagine the title on the cover of the phone book is given by the weights $(\lambda_1, ..., \lambda_E) \in \mathbb{F}_q^m$. Since both parties know $\mathcal{E}_1$, ..., $\mathcal{E}_E$, they agree on which equation is referenced by the weights.

We'll just check *one* such random linear combination. This is good enough because, in fact, if an assignment of the variables fails even one of the E equations, it will fail the collated equation with probability $1 - \frac{1}{q}$ — exactly! (To see this, suppose that equation $\mathcal{E}_1$ was failed by the assignment A. Then, for any fixed choice of λ_2, ..., λ_E, there is always exactly one choice of λ_1 which makes the collated equation true, while the other $q - 1$ all fail.)

To emphasize again: Peggy is printing q^E phone books right now and we only use one. Look, I'm sorry, okay?

§15.4.3 Sum-checking the equation (or: how to print the phone book)

Let's zoom in on one linear combination to use sum-check on. (In other words, pick only one of the phone books at random.) Let's agree to describe the equation using the notation

$$c = \sum_{\vec{i} \in \mathbb{H}^m} \sum_{\vec{j} \in \mathbb{H}^m} a_{\vec{i},\vec{j}} x_{\vec{i}} \cdot x_{\vec{j}} + \sum_{\vec{i} \in \mathbb{H}^m} b_{\vec{i}} x_{\vec{i}}.$$

In other words, we've changed notation so both the variables and the coefficients are indexed by vectors in $\mathbb{H}^m$. When we actually implement this protocol, the coefficients need to be actually computed: they came out of $\lambda_1\mathcal{E}_1 + ... + \lambda_E\mathcal{E}_E$. (So for example, the value of c above is given by λ_1 times the constant term of $\mathcal{E}_1$, plus λ_2 times the constant term of $\mathcal{E}_2$, etc.)

Our sum-check protocol that we talked about earlier used a sequence $(r_1, ..., r_n) \in \{0, 1\}^n$. For our purposes, we have these quadratic equations, and so it'll be convenient for us if we alter the protocol to use pairs $\left(\vec{i}, \vec{j}\right) \in \mathbb{F}_q^m \times \mathbb{F}_q^m$ instead. In other words, rather than $f(\vec{v})$ our variables will be indexed instead in the following way:

$$f : \mathbb{H}^m \times \mathbb{H}^m \to \mathbb{F}_q$$

$$f\left(\vec{i}, \vec{j}\right) := a_{\vec{i},\vec{j}} A\left(\vec{i}\right) A\left(\vec{j}\right) + \frac{1}{|\mathbb{H}|^m} b_{\vec{i}} A\left(\vec{i}\right).$$

Hence Peggy is trying to convince Victor that

$$\sum_{\vec{i} \in \mathbb{F}_q^m} \sum_{\vec{j} \in \mathbb{F}_q^m} f\left(\vec{i}, \vec{j}\right) = c.$$

In this modified sum-check protocol, Victor picks the indices two at a time. So in the step where Victor picked r_1 in the previous step, he instead picks i_1 and j_1 at once. Then instead of picking an r_2, he picks a pair (i_2, j_2) and so on.

Then, to run the protocol, the entries of the phone book are going to correspond to

$$P \in \mathbb{F}_q[T_1, ..., T_m, U_1, ..., U_m]$$
$$P(T_1, ..., T_m, U_1, ..., U_m) := \tilde{a}(T_1, ..., T_m, U_1, ..., U_m)\tilde{A}(T_1, ..., T_m)\tilde{A}(U_1, ..., U_m) + \frac{1}{|\mathbb{H}|^m}\tilde{b}(T_1, ..., T_m)\tilde{A}(T_1, ..., T_m)$$

in place of what we called $P(x, y, z)$ in the sum-check section.

I want to stress now the tilde's above are actually hiding a lot of work. Let's unpack it a bit: what does $\tilde{a}$ mean? After all, when you unwind this notational mess we wrote, we realize that the a's and b's came out of the coefficients of the original equations $\mathcal{E}_k$.

The answer is that both Victor and Peggy have a lot of arithmetic to do. Specifically, for Peggy, when she's printing this phone book for $(\lambda_1, ..., \lambda_E)$, needs to apply the extension result three times:

- Peggy views $a_{\vec{i},\vec{j}}$ as a function $\mathbb{H}^{2m} \to \mathbb{F}_q$ and extends it to a polynomial using the above; this lets us define $\tilde{a} \in \mathbb{F}_q[T_1, ..., T_m, U_1, ..., U_m]$ as a *bona fide* $2m$-variate polynomial.
- Peggy does the same for $\tilde{b}_{\vec{i}}$.
- Finally, Peggy does the same on $A : \mathbb{H}^m \to \mathbb{F}_q$, extending it to $\tilde{A} \in \mathbb{F}_q[T_1, ..., T_m]$. (However, this step is the same across all the phone books, so it only happens once.)

Victor has to do the same work for $a_{\vec{i},\vec{j}}$ and $b_{\vec{i}}$. Victor can do this, because he picked the λ's, as he computed the coefficients of his linear combination too. But Victor does *not* do the last step of computing $\tilde{A}$: for that, he just refers to the poster Peggy gave him, which conveniently happens to have a table of values of $\tilde{A}$.

Now we can actually finally describe the full contents of the phone book. It's not simply a table of values of P! We saw in the sum-check protocol that we needed a lot of intermediate steps too (like the $23T + 46$, $161U + 23$, $112V + 197$). So the contents of this phone book include, for every index k, every single possible result that Victor would need to run sum-check at the kth step. That is, the kth part of this phone book are a big directory where, for each possible choice of indices $(i_1, ..., i_{k-1}, j_1, ..., j_{k-1})$, Peggy has printed the two-variable polynomial in $\mathbb{F}_q[T, U]$ that arises from sum-check. (There are two variables rather than one now, because (i_k, j_k) are selected in pairs.)

This gives Victor a non-interactive way to run sum-check. Rather than ask Peggy, consult the already printed phone book. Inefficient? Yes. Works? Also yes.

§15.4.4 Finishing up

Once Victor runs through the sum-check protocol, at the end he has a random $\left(\vec{i}, \vec{j}\right)$ and received the checked the phone book for $P\left(\vec{i}, \vec{j}\right)$.

Assuming it checks out, his other task is to verify that the accompanying posters that Peggy sent — that is, the table of values B_0 and B_2 associated to $\tilde{A}$ — look like they mostly come from a low-degree polynomial. Unlike the sum-check step where we needed to hack the earlier procedure, this step is a direct application of line-versus-point test, without modification.

Up until now the phone book and posters haven't interacted. So Victor has to do one more check: he makes sure that the value of $P\left(\vec{i}, \vec{j}\right)$ he got from the phone book in fact matches the value corresponding to the poster B_0. In other words, he does the arduous task of computing the extensions $\tilde{a}$ and $\tilde{b}$, and finally verifies that

$$P\left(\vec{i},\vec{j}\right) := \tilde{a}\left(\vec{i},\vec{j}\right) B_0\left[\vec{i}\right] B_0\left[\vec{j}\right] + \frac{1}{|\mathbb{H}|^m}\tilde{b}\left(\vec{i}\right) B_0\left[\vec{i}\right]$$

is actually true.

§15.5 Reasons to not be excited by this protocol

The previous section describes a long procedure that has a PCP flavor, but it suffers from several issues (which is why we call it a toy example).

- **Amount of reading**: The amount of reading on Victor's part is not $O(1)$ like we promised. The low-degree testing step with the posters used $O(1)$ entries, but the sum-check required reading roughly

  $$O(|\mathbb{H}|^2) \cdot (m + O(1)) \approx \frac{(\log N)^3}{\log\log N}$$

  entries from the phone book. The PCP theorem promises we can get that down to $O(1)$, but that's beyond this post's scope.

- **Length of proof**: The procedure above involved mailing q^E phone books, which is what we in the business call either "unacceptably inefficient" or "fucking terrible", depending on whether you're in polite company or not. The next section will show how to get this down to qEN if q is large enough.

  For context, in this protocol one wants a reasonably small prime q which is about polynomial in $\log(EN)$. After all, Quad-SAT is already an NP-complete problem for $q = 2$. (In contrast, in other unrelated more modern ecosystems, the prime q often instead denotes a fixed large prime $q \approx 2^{256}$.)

- **Time complexity**: Even though Victor doesn't read much, Peggy and Victor both do quite a bit of computation. For example,

  - Victor has to compute $\tilde{a}_{\vec{i},\vec{j}}$ for his one phone book.
  - Peggy needs to do it for *every* phone book.

- One other weird thing about this result is that, even though Victor has to read only a small part of Peggy's proof, he still has to read the entire *problem statement*, that is, the entire system of equations from the original Quad-SAT. This can feel strange because for Quad-SAT, the problem statement is of similar length to the satisfying assignment!

TODO: some comments from Telegram here

§15.6 Reducing the number of phone books — error correcting codes

We saw that we can combine all the E equations from Quad-SAT into a single one by taking a random linear combination. Our goal is to improve this by taking a "random-looking" combination that still has the same property an assignment failing even one of the E equations is going to fail the collated equation with probability close to 1.

It turns out there is actually a well-developed theory of how to take the "random-looking" linear combination, and it comes from the study of *error-correcting codes*. We will use this to show that if $q \geq 4(\log(EN))^2$ is large enough, one can do this with only $q \cdot EN$ combinations. That's much better than the q^E we had before.

§15.6.1 (Optional) A hint of the idea: polynomial combinations

Rather than simply doing a random linear combination, one could imagine considering the following $100E$ combinations

$$k^1\mathcal{E}_1 + k^2\mathcal{E}_2 + \ldots + k^E\mathcal{E}_E \quad \text{for } k = 1, 2, \ldots, 100E.$$

If any of the equations $\mathcal{E}_i$ are wrong, then we can view this as a degree E polynomial in k, and hence it will have at most E roots. Since we have $100E$ combinations, that means at least 99% of the combinations will fail.

So why don't we just do that? Well, the issue is that we are working over $\mathbb{F}_q$. And this argument only works if $q \geq 100E$, which is too big.

§15.6.2 Definition of error-correcting codes

An **error-correcting code** is a bunch of codewords with the property that any two differ in "many" places. An example is the following set of sixteen bit-strings of length 7:

$$\begin{aligned} C = \{&0000000, 1101000, 0110100, 0011010, \\ &0001101, 1000110, 0100011, 1010001, \\ &0010111, 1001011, 1100101, 1110010 \\ &0111001, 1011100, 0101110, 1111111\} \subseteq \mathbb{F}_2^7 \end{aligned}$$

which has the nice property that any two of the codewords in it differ in at least 3 bits. This particular C also enjoys the nice property that it's actually a vector subspace of $\mathbb{F}_2^7$ (i.e. it is closed under addition). In practice, all the examples we consider will be subspaces, and we call them **linear error-correcting codes** to reflect this.

When designing an error-correcting code, broadly your goal is to make sure the minimum distance of the code is as large as possible, while still trying to squeeze in as many codewords as possible. The notations used for this are:

- Usually we let q denote the alphabet size and n the block length (the length of the codewords), so the codewords live in the set of q^n possible length n strings.
- The **relative distance** is defined as the minimum Hamming distance divided by n; Higher relative distance is better (more error corrections).
- The **rate** is the $\log_{q^n}(\text{num codewords})$. Higher rates are better (more densely packed codewords).

So the example C has relative distance $\frac{3}{7}$, and rate $\log_{2^7}(16) = \frac{4}{7}$.

§15.6.3 Examples of error-correcting codes

TODO: Hadamard, ...

§15.6.4 Composition

TODO: define this

§15.6.5 Recipe

Compose the Reed-Solomon codes

$$C_1 = \mathrm{RS}_{d=E, q=EN}$$
$$C_2 = \mathrm{RS}_{d=\log(EN), q}.$$

This gives a linear code corresponding to an $s \times m$ matrix M, where $s := q \cdot EN$, which has relative distance at least $1 - \frac{1}{\sqrt{q}}$. The rows of this matrix (just the rows, not their row span) then correspond to the desired linear combinations.