# Three Easy Pieces in Programmable Cryptography

0xPARC

28 June 2024

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

## Table of contents

# §1 Introduction

## §1.1 What is programmable cryptography?

Cryptography is everywhere now as a field that is widely used in everyday life. 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. *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. The goal of this "second-generation cryptography" can be described as:

> *We want to devise cryptographic primitives that can 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 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

## §1.2 Ideas in programmable cryptography

These notes address programmable cryptography through expositions on specific topics. We quickly bpreview them here.

### §1.2.1 2PC: Two-party computation

In a *two-party computation*, two people want to jointly compute some known function

$$F(x_1, x_2),$$

where the $i$th person only knows the input $x_i$ — and they want to do it without either person learning the other person's input.

For example, in 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 $F = \max()$, and $x_i$ is the $i$'th person's income.

Two-party computation makes a promise that we'll be able to do this for *any* function $F$ as long as we can implement it in code. It generalizes to *multi-party computation (MPC)*, which is one of the main classes of programmable cryptography.

### §1.2.2 SNARK: proofs of general problems

The *SNARK*, first described in 2012, provides a way to produce proofs of **arbitrary** problem statements, once the problem statements are encoded as a system of equations in a certain way. The name stands for:

- *Succinct*: the proof length is short (actually constant length).

- *Non-interactive*: the protocol does not require back-and-forth communication.
- *Argument*: basically a proof. There's a technical difference, but we won't worry about it.
- *of Knowledge*: the proof doesn't just show the system of equations has a solution; it also shows that the prover knows one.

One additional feature (which we will not cover in these notes) is *zero-knowledge (zk)* (which turns the abbreviation into "zkSNARK"): with a zero-knowledge proof, a person reading the proof doesn't learn anything about the solution besides that it's correct.

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

This is an active area of research, and many different proof systems are known. These notes focus on one construction, called PLONK (Section 4.2).

### §1.2.3 FHE: Fully homomorphic encryption

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.

## §1.3 Where these fit together

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

This whole field is an active area of research.

- Can we make existing tools (SNARKS, etc.) more efficient? For example, the cost of proving a computation in a SNARK is currently about $10^6$ times the cost of doing the computation directly. Can we bring that number down?
- 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 Two-party Computation

## §2.1 Garbled Circuits

Imagine Alice and Bob each have some secret values $a$ and $b$, and would like to jointly compute some function $f$ over their respective inputs. Furthermore, they'd like to keep their secret values hidden from each other: if Alice and Bob follow the protocol honestly, they should both end up learning the correct value of $f(a, b)$, but Alice shouldn't learn anything about $b$ (other than what could be learned by knowing both $a$ and $f(a, b)$), and likewise for Bob.

Yao's Garbled Circuits is one of the most well-known 2PC protocols (Vitalik has a great explanation on his blog). The protocol is quite clever, and optimized variants of the protocol are being implemented and used today.

### §2.1.1 The Problem

Here is our problem setting, slightly more formally:

- $A$ knows a secret bitstring $a$ of length $s$ bits
- $B$ knows a secret bitstring $b$ of length $t$ bits
- $C$ is a binary circuit, which takes in $s + t$ bits, and runs them through some $n$ gates. The outputs of some of the gates are the public outputs of the circuit. Without loss of generality, let's also suppose that each gate in $C$ accepts either 1 or 2 input bits, and outputs a single output bit.
- $A$ and $B$ would like to jointly compute $C(a, b)$ without revealing to each other their secrets.

### §2.1.2 Outline of Solution

Our solution will contain two key components:

- Alice constructs a *garbled circuit* that takes in the value $b$ (whatever it is) and spits out $f(a, b)$. A *garbled circuit*, roughly speaking, is an "encrypted" circuit that takes encrypted input and creates encrypted output."

- An *oblivious transfer* is a protocol where X has two messages, $m_0$ and $m_1$. Y can get exactly one of them, $m_i$, without letting X know what $i$ is. In this context, Alice ends up sending Bob a password for his input in a way that Bob doesn't learn the passwords for any other inputs, and Alice doesn't find out which password she sent to Bob.

> **TODO**: Ask Gub to paraphrase, he writes well
>
> I rewrote the above slightly. Feel free to discard/give to Gub still [-YZ]

In slightly more detail:

1. Whatever the function $f$ is, we'll assume that it takes $m + n$ bits of input $a_1, ..., a_m$ and $b_1, ..., b_n$, and that it's computed by some sort of circuit made of AND, OR and NOT gates.
2. Alice's first task is to "plug her own inputs into this circuit $f$." The result will be a new circuit (you might call it $f_a$) that has just $n$ input slots for Bob's $n$ bits $b_1, ..., b_n$.
3. Now, Alice is going to "garble" the circuit $f_a$. Once it's garbled, Bob won't be able to see how it works. He'll only be able to plug his own input $(b_1, ..., b_n)$ into the circuit.
4. To prevent Bob from plugging other inputs in as well, a garbled circuit will require a "password" for each input Bob wants to plug in – a different password for every possible input. If Bob has the password for $(b_1, ..., b_n)$, he can learn $f_{a(b_1,...,b_n)} = f(a, b)$ – but he won't learn anything else about how the circuit works.
5. Now, Alice has all the passwords for all the possible inputs, but how can she give Bob the password for $(b_1, ..., b_n)$? Alice doesn't want to let Bob have any other passwords – and Bob isn't willing to tell Alice which password he is asking for. This is where we will use the "oblivious transfer."

We now flesh out this outline, starting with garbled circuits.

### §2.1.3 Garbled gates

Our garbled circuits are going to be built out of *garbled gates*. A garbled gate is like a traditional gate (like AND, OR, NAND, NOR), except its functionality is hidden.

What does that mean? Let's say the gate has two input bits, so there are four possible inputs to the gate: $(0,0), (0,1), (1,0), (1,1)$. For each of those four inputs $(x, y)$, there is a secret password $P_{x,y}$. The gate $G$ will only reveal its value $G(x, y)$ if you give it the password $P_{x,y}$.

Here is a natural approach to make a garbled gate. Choose a *symmetric-key* [1] encryption scheme Enc [2] and publish the following table:

| | |
|---|---|
| $(0,0)$ | $\mathrm{Enc}_{P_{0,0}}(G(0,0))$ |
| $(0,1)$ | $\mathrm{Enc}_{P_{0,1}}(G(0,1))$ |
| $(1,0)$ | $\mathrm{Enc}_{P_{1,0}}(G(1,0))$ |
| $(1,1)$ | $\mathrm{Enc}_{P_{1,1}}(G(1,1))$ |

If you have the values $x$ and $y$, and you know the password $P_{x,y}$, you just go to the $(x, y)$ row of the table, look up

$$\mathrm{Enc}_{P_{x,y}}(G(x, y)),$$

decrypt it, and learn $G(x, y)$.

But if you don't know the password $P_{x,y}$, assuming Enc is a suitably secure encryption scheme, you won't learn anything about the value $G_{x,y}$ from its encryption.

[1] Symmetric-key encryption is probably what you think of when you think of plain-vanilla encryption: You use a secret key $K$ to encrypt a message $m$, and then you use the same secret key $K$ to decrypt it.

[2] We'll talk later about what sort of encryption scheme is suitable for this…

### §2.1.4 Chaining garbled gates

The next step to combine a bunch of garbled gates into a circuit. We'll need to make two changes to the protocol.

1. To chain garbled gates together, we need to modify the output of each gate: In addition to outputting the bit $z = G_i(x, y)$, the $i$-th gate $G_i$ will also output a password $P_z$ that Bob can use at the next step.

   Now Bob has one bit coming in for the left-hand input $x$, and it came with some password $P_x^{\text{left}}$ – and then another bit coming in for $y$, that came with some password $P_y^{\text{right}}$. To get the combined password $P_{x,y}$, Bob will just concatenate the two passwords $P_x^{\text{left}}$ and $P_y^{\text{right}}$.

2. To keep the functionality of the circuit hidden, we don't want Bob to learn anything about the structure of the individual gates – even the single bit he gets as output.

   This is an easy fix: Instead of having the gate output both the bit $z$ and the password $P_z$, we'll just have the gate output the password $P_z$.

   But now how does Bob know what values to feed into the next gate? The left-hand column of the "gate table" needs to be indexed by the passwords $P_x^{\text{left}}$ and $P_y^{\text{right}}$, not by the bits $(x, y)$. But we don't want Bob to learn the other passwords from the table!

   Let's say this again. We want:
   - If Bob knows both passwords $P_x^{\text{left}}$ and $P_y^{\text{right}}$, Bob can find the row of the table for the input $(x, y)$.
   - If Bob doesn't know the passwords, he can't learn them by looking at the table.

   Of course, the solution is to use a hash function! So here is the new version of our garbled gate. For simplicity, I'll assume it's an AND gate – so the outputs will be (the passwords encoding) 0, 0, 0, 1.

| | |
|---|---|
| $\text{hash}\left(P_0^{\text{left}}, P_0^{\text{right}}\right)$ | $\text{Enc}_{P_0^{\text{left}}, P_0^{\text{right}}}\left(P_0^{\text{out}}\right)$ |
| $\text{hash}\left(P_0^{\text{left}}, P_1^{\text{right}}\right)$ | $\text{Enc}_{P_0^{\text{left}}, P_1^{\text{right}}}\left(P_0^{\text{out}}\right)$ |
| $\text{hash}\left(P_1^{\text{left}}, P_0^{\text{right}}\right)$ | $\text{Enc}_{P_1^{\text{left}}, P_0^{\text{right}}}\left(P_0^{\text{out}}\right)$ |
| $\text{hash}\left(P_1^{\text{left}}, P_1^{\text{right}}\right)$ | $\text{Enc}_{P_1^{\text{left}}, P_1^{\text{right}}}\left(P_1^{\text{out}}\right)$ |

### §2.1.5 How Bob uses one gate

Let's play through one round of Bob's gate-using protocol.

1. Suppose Bob's input bits are 0 (on the left) and 1 (on the right). Bob doesn't know he has 0 and 1 (but we do!). Bob knows his left password is some value $P_0^{\text{left}}$, and his right password is some other value $P_1^{\text{right}}$.
2. Bob takes the two passwords, concatenates them, and computes a hash. Now Bob has
$$\text{hash}\left(P_0^{\text{left}}, P_1^{\text{right}}\right).$$
3. Bob finds the row of the table indexed by $\text{hash}\left(P_0^{\text{left}}, P_1^{\text{right}}\right)$, and he uses it to look up
$$\text{Enc}_{P_0^{\text{left}}, P_1^{\text{right}}}\left(P_0^{\text{out}}\right).$$
4. Bob uses the concatenation of the two passwords $P_0^{\text{left}}, P_1^{\text{right}}$ to decrypt $P_0^{\text{out}}$.
5. Now Bob has the password for the bit 0, to feed into the next gate – but he doesn't know his bit is 0.

So Bob is exactly where he started: he knows the password for his bit, but he doesn't know his bit. So we can chain together as many of these garbled gates as we like to make a full garbled circuit.

### §2.1.6 How the circuit ends

At the end of the computation, Bob needs to learn the final result. How?

Easy! The final output gates are different from the intermediate gates. Instead of outputting a password, they will just output the resulting bit in plain text.

### §2.1.7 How the circuit starts

This is trickier. At the beginning of the computation, Bob needs to learn the passwords for all of his input bits. Let's just frame the problem for a single bit.

- Alice has two passwords, $P_0$ and $P_1$.
- Bob has a bit $b$, either 0 or 1.
- Bob wants to learn one of the passwords, $P_b$, from Alice.
- Bob does not want Alice to learn the value of $b$.
- Alice does not want Bob to learn the other password.

This is where *oblivious transfer* comes in, which we'll see in Section 2.2.

## §2.2 Oblivious Transfer

Alice has $n$ messages $x_1, ..., x_n$. Bob wants to request the $i$-th message, without letting Alice learn anything about the value of $i$. Alice wants to send Bob $x_i$, without letting him learn anything about the other $n-1$ messages. An *oblivious transfer (OT)* allows Alice to transfer a single message to Bob, but she remains oblivious as to which message she has transferred. We'll see two simple protocols to achieve this.

(In fact, for two-party computation, we only need "1-of-2 OT": Alice has $x_1$ and $x_2$, and she wants to send one of those two to Bob. But "1-of-$n$ OT" isn't any harder, so we'll do 1-of-$n$.)

### §2.2.1 Commutative encryption

Let's imagine that Alice and Bob have access to some encryption scheme that is *commutative*:

$$\mathrm{Dec}_{B\left(\mathrm{Dec}_A\left(\mathrm{Enc}_B\left(\mathrm{Enc}_{A(x)}\right)\right)\right)} = x.$$

In other words, if Alice encrypts a message, and Bob applies a second-layer of encryption to the encrypted message, it doesn't matter which order Alice and Bob decrypt the message in – they will still get the original message back.

A metaphor for commutative encryption is a box that's locked with two padlocks. Alice puts a message inside the box, lock it with her lock, and ship it to Bob. Bob puts his own lock back on the box and ships it back to Alice. What's special about commutative encryption is that Bob's lock doesn't block Alice from unlocking her own – so Alice can remove her lock and send it back to Bob, and then Bob removes his lock and recovers the message.

Mathematically, you can get commutative encryption by working in a finite group (for example $\mathbb{Z}_p^\times$, or an elliptic curve).

1. Alice's secret key is an integer $a$;

she encrypts a message $g$ by raising it to the $a$-th power, and she sends Bob $g^a$.

2. Bob encrypts again with his own secret key $b$,

and he sends $(g^a)^b = g^{ab}$ back to Alice.

3. Now Alice removes her lock by taking an $a$-th root. The result is $g^b$, which she sends back to Bob. And Bob takes another $b$-th root, recovering $g$.

### §2.2.2 OT using commutative encryption

Our first oblivious transfer protocol is built on the commutative encryption we just described.

Alice has $n$ messages $x_1, ..., x_n$, which we may as well assume are elements of the group $G$. Alice chooses a secret key $a$, encrypts each message, and sends all $n$ ciphertexts to Bob:

$$\mathrm{Enc}_{a(x_1)}, ..., \mathrm{Enc}_{a(x_n)}\,.$$

But crucially, Alice sends the ciphertexts in order, so Bob knows which is which.

At this point, Bob can't read any of the messages, because he doesn't know the keys. No problem! Bob just picks out the $i$-th ciphertext $\text{Enc}_{a(x_i)}$, adds his own layer of encryption onto it, and sends the resulting doubly-encoded message back to Alice:

$$\text{Enc}_{b\left(\text{Enc}_{a(x_i)}\right)}.$$

Alice doesn't know Bob's key $b$, so she can't learn anything about the message he encrypted – even though it originally came from her. Nonetheless she can apply her own decryption method $\text{Dec}_a$ to it. Since the encryption scheme is commutative, the result of Alice's decryption is simply

$$\text{Enc}_{b(x_i)},$$

which she sends back to Bob.

And Bob decrypts the message to learn $x_i$.

### §2.2.3 OT in one step

The protocol above required one and a half rounds of communication: Alice sent two messages to Bob, and Bob sent one message back to Alice.

We can do better, using public-key cryptography.

Let's start with a simplified protocol that is not quite secure. The idea is for Bob to send Alice $n$ keys

$$b_1, \ldots, b_n.$$

One of the $n$, say $b_i$, is a public key for which Bob knows the private key. The other $n - 1$ are random garbage.

Alice then uses one key to encrypt each message, and sends back to Bob:

$$\text{Enc}_{b_1}(x_1), \ldots, \text{Enc}_{b_n}(x_n).$$

Now Bob uses the private key for $b_i$ to decrypt $x_i$, and he's done.

Is Bob happy with this protocol? Yes. Alice has no way of learning the value of $i$, as long as she can't distinguish a true public key from a random fake key (which is true of public-key schemes in practice).

But is Alice happy with it? Not so much. A cheating Bob could send $n$ different public keys, and Alice has no way to detect it – like we just said, Alice can't tell random garbage from a true public key! And then Bob would be able to decrypt all $n$ messages $x_1, ..., x_n$.

But there's a simple trick to fix it. Bob chooses some "verifiably random" value $r$; to fix ideas, we could agree to use $r = \text{hash}(1)$. Then we require that the numbers $b_1, ..., b_n$ form an arithmetic progression with common difference $r$. Bob chooses $i$, computes a public-private key pair, and sets $b_i$ equal to that key. Then all the other terms $b_1, ..., b_n$ are determined by the arithmetic progression requirement $b_j = b_i + (j - i)r$. (Or if the keys are elements of a group in multiplicative notation, we could write this as $b_j = r^{j-i} \cdot b_i$.)

Is this secure? If we think of the hash function as a random-number generator, then all $n - 1$ "garbage keys" are effectively random values. So now the question is: Can Bob compute a private key for a given (randomly generated) public key? It's a standard assumption in public-key cryptography that Bob can't do this: there's no algorithm that reads in a public key and spits out the corresponding private key. (Otherwise, the whole enterprise is doomed.) So Alice is guaranteed that Bob only knows how to decrypt (at most) one message.

In fact, some public-key cryptosystems (like ElGamal) have a sort of "homomorphic" property: If you know the private keys for to two different public keys $b_1$ and $b_2$, then you can compute the private key for the public key $b_2 b_1^{-1}$. (In ElGamal, this is true because the private key is just a discrete logarithm.) So, if Bob could dishonestly decrypt two of Alice's messages, he could compute the private key for the public key $r$. But $r$ is verifiably random, and it's very hard (we assume) for Bob to find a private key for a random public key.

## §2.3 2PC Takeaways

1. A *garbled circuit* allows Alice and Bob to jointly compute some function over their respective secret inputs. We can think of this as your prototypical *2PC* (two-party computation).
2. The main ingredient of a garbled circuit are *garbled gates* which is a gate whose functionality is hidden. This can be done e.g. by Alice precomputing different outputs of the garbled circuit based on all possible inputs of Bob, and then letting Bob pick one.
3. Bob "picks an input" with the technique of *oblivious transfer (OT)*. This can be built in various ways, including with commutative encryption or public-key cryptography.
4. More generally, this means in theory a group of people can compute whatever secret function they want, which is the field of *multiparty computation (MPC)*.

# §3 SNARKs Prelude: Elliptic Curves and Polynomial Commitments

Before we talk about SNARKs (specifically, PLONK), it helps to separate out an ingredient that underlies much of programmable cryptography, which is the idea of a *polynomial commitment*. Specifically, we will talk about the KZG polynomial commitment, which plays an important role in PLONK (and many other protocols). For a higher resolution understanding of KZG, it helps to understand *elliptic curves* (especially in the context of *pairings*), which are ubiquitous in cryptography. If you are uninterested (or experienced) in mathematical details, you can and should skip elliptic curves and jump to Section 3.4. If you are comfortable with black-boxing Section 3.4, you can even jump straight to SNARKs into the next chapter.

The roadmap goes roughly as follows:

- In Section 3.1 we will define *elliptic curves* and describe one standard elliptic curve $E$, the *BN254 curve*, that will be used in these notes.
- In Section 3.2 we describe the *discrete logarithm assumption* (Assumption 3.6), which we need to make to provide security to our protocols. As an example, in Section 3.2.2 we describe how Assumption 3.6 can be used to construct a signature scheme, namely EdDSA.
- The EdDSA idea will later grow up to be the KZG commitment scheme in Section 3.4.

## §3.1 Elliptic curves

Every modern cryptosystem rests on a hard problem – a computationally infeasible challenge whose difficulty makes the protocol secure. The best-known example is RSA, which is secure because it is hard to factor a composite number (like 6177) into prime factors ($6177 = 71 \cdot 87$).

Our SNARK protocol will be based on a different hard problem: the discrete logarithm problem (Section 3.2) on elliptic curves. But be-

fore we get to the problem, we need to introduce some of the math behind elliptic curves.

An *elliptic curve* is a set of points with a group operation. The set of points is the set of solutions $(x, y)$ to an equation in two variables; the group operation is a rule for "adding" two of the points to get a third point. Our first task will be to understand what all this means. Rather than set up a general definition of elliptic curve, for these notes we will be satisfied to describe one specific elliptic curve that can be used for all the protocols we describe later. The curve we choose for these notes is the *BN254 curve*.

The BN254 specification fixes a specific[3] 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

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

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

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:

**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.

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 the elliptic curve. This addition will make $E(\mathbb{F}_p)$ into an abelian group whose identity element is the point at infinity $O$. This addition can be formalized as a *group law*, which is an equation that points on the curve must follow.

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. We provide details below to the interested reader.

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 $P + Q$? 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? 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 $Q = (x_Q, y_Q)$, 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.

In summary we have endowed the set of points $E(\mathbb{F}_p)$ 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 $\mathbb{Z}/q\mathbb{Z}$)**:** 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. Moving forward we'll abuse notation slightly and just write $E$ instead of $E(\mathbb{F}_p)$. In fancy language, $E$ will be a one-dimensional vector space over $\mathbb{F}_q$. In less fancy language, we'll be working with points on $E$ as black boxes. We'll be able to add them, subtract them, and multiply them by arbitrary scalars from $\mathbb{F}_q$.

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 $\mathbb{F}_p$).

**Remark 3.5**: 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

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

**Assumption 3.6** (Discrete logarithm assumption): 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 $n \cdot g = g'$. (the act of obtaining this integer is called the *discrete logarithm problem*)

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 about $2^{254}$ in size.

**Remark 3.7** (The name "discrete log"): This problem is called discrete log because if one used multiplicative notation for the group operation, 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.

### §3.2.1 Curves other than BN254

We comment briefly on how the previous definitions 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 non-degeneracy 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 3.6. 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 3.3.2 when we need it later. (In contrast, Curve25519 does not have this property.)

### §3.2.2 Example application: EdDSA signature scheme

We'll show how Assumption 3.6 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.)

**Definition 3.8**: 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.

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

**Algorithm 3.9** (EdDSA public and secret key):

1. Alice picks a random integer $d \in \mathbb{F}_q$ as her *secret key* (a piece of information that she needs to keep private for the security of the protocol).
2. Alice publishes $[d] \in E$ as her *public key* (a piece of information which, even when obtained by adversaries, does not challenge the security of the protocol).

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

**Algorithm 3.10** (EdDSA signature generation): Suppose Alice wants to sign a message $\mathsf{msg}$.

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 $\mathsf{msg}$ with all public information, say
$$n := \mathrm{hash}([r], \mathsf{msg}, [d]).$$
3. Alice publishes the integer
$$s := (r + dn) \bmod q.$$

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

**Algorithm 3.11** (EdDSA signature generation): For Bob to verify a signature $([r], s)$ for $\mathsf{msg}$:

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 3.6.

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 3.4 we will use ideas quite similar to this to build the KZG commitment scheme.

### §3.2.3 Example application: Pedersen commitments

A *commitment scheme* is a protocol where Alice wants to commit some value $x$ to Bob that is later revealed. Typically Alice gives Bob some "commitment" $c(x)$ and later reveals $x$. What we want is that this protocol is both *binding* (Alice cannot change her mind about $x$ depending on Bob's later actions) and *hiding* (Bob does not get any information about $x$ from $c(x)$). The KZG scheme we are building towards will be a commitment scheme for polynomials, but we can already use elliptic curves to commit **numbers** with something called a Pedersen commitment, which we will now describe.

A multivariable generalization of Assumption 3.6 is that if $g_1, \ldots, g_n \in E$ are a bunch of randomly chosen points of $E$ with order $q$, then it's computationally infeasible to find $(a_1, \ldots, a_n) \neq (b_1, \ldots, b_n) \in \mathbb{F}_q^n$ such that

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

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

**Definition 3.12**: 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 3.13**: 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 3.14**: 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 $a_i$; but given the entire vector $\vec{a}$ one can compute the Pedersen commitment easily.

We won't use Pedersen commitments in this book, but they will be closely related to KZG.

## §3.3 Bilinear pairings on elliptic curves

Before we are ready for KZG, there is one more piece of elliptic curve math that we need.

Recall that 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 3.15**: 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 3.16**: 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 $N$. We think this should be called a *bilinear pairing*, but for some reason everyone just says *pairing* instead. A

curve is called *pairing-friendly* if this pairing can be computed reasonably quickly (e.g. BN254 is pairing-friendly, but Curve25519 is not).

This construction actually uses some really deep number theory (heavier than all the math in Section 3.1) 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 3.3.2 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 3.17**: 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 3.18** (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 knowing one of $[ab]$, $[bc]$, $[ca]$).

### §3.3.1 Verifying more complicated claims

**Example 3.19**: 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 verifying 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.

### §3.3.2 So which curves are pairing-friendly?

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

**Proposition 3.20**: 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 $\text{pair}(a, b) \in \mathbb{F}_{p^k}^\times$ – 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 $q$th root of unity in $\mathbb{F}_{p^k}$, and it will satisfy $\text{pair}([m], [n]) = \zeta^{mn}$, where $\zeta$ is some fixed $q$th 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 $256k$ 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.

## §3.4 KZG commitments

The goal of a *polynomial commitment scheme* 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 *Kate-Zaverucha-Goldberg (KZG)* commitment scheme is amazingly efficient because both the commitment and proof lengths are a single point on $E$, encodable in 256 bits.

### §3.4.1 The setup

Remember the notation $[n] := n \cdot g \in E$ defined in Definition 3.8. 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.

The setup 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 3.21**: 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 $[s^0] = [1], [s^1], ..., [s^M]$. This is why we call them "trusted": the security of KZG depends on them not saving the value of $s$.

Given the 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.

### §3.4.2 The KZG commitment scheme

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

**Algorithm 3.22** (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 3.23** (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
$$\text{pair}([Q(s)], [s] - [z]) = \text{pair}([P(s)] - [y], [1]) \quad (2)$$
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.

### §3.4.3 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 curve point for the proof.

For example, suppose Peggy wanted to prove that $P(1) = 100$, $P(2) = 400$, ..., $P(9) = 8100$. (We chose these numbers so that $P(X) = 100X^2$ for $X = 1, ..., 9$.)

Evaluating a polynomial at $1, 2, ..., 9$ is essentially the same as dividing by $(X - 1)(X - 2)...(X - 9)$ and taking the remainder. In

other words, if Peggy does a polynomial long division, she will find that

$$P(X) = Q(X)((X-1)(X-2)...(X-9)) + 100X^2.$$

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

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

The full generality just replaces the $100X^2$ 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 3.24** (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
$$\begin{aligned}&\text{pair}([Q(s)], [(s-z_1)(s-z_2)...(s-z_n)])\\ &= \text{pair}([P(s)] - [f(s)], [1]).\end{aligned}$$

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 $f$.

### §3.4.4 Root check (using long division with commitment schemes)

To make PLONK work, we're going to need a small variant of the multi-opening protocol for KZG commitments (Section 3.4.3), which

we call *root-check* (not a standard name). Here's the problem statement:

> *Problem 3.25*: 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 $S$.

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 $\lambda$.

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

$$F(X) := P_1(X) + P_2(X)P_3(X)^4 + P_4(X)P_5(X)P_6(X) - 1337$$

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 3.26** (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 $\text{Com}(H)$.
4. Victor picks a random challenge $\lambda \in \mathbb{F}_q$ and asks Peggy to open $\text{Com}(H)$ at $\lambda$, as well as the value of $F$ at $\lambda$.
5. Victor verifies $F(\lambda) = Z(\lambda)H(\lambda)$.

## §3.5 KZG Takeaways

1. *Elliptic curves* are very useful in cryptography. Roughly speaking, they are sets of points (usually in $\mathbb{F}_p^2$) that satisfy some group law / "addition." The BN254 curve is a good "typical curve" to keep in mind.
2. The *discrete logarithm* assumption is a common "hard problem assumption" used in cryptography with different groups. Specifically, since elliptic curves are groups, discrete logarithm over elliptic curves is very often used.
3. *Commitment schemes* are ways for one party to commit values to another. Elliptic curves enable *Pedersen commitments*, a very useful example of a commitment scheme.
4. Specifically, *polynomial commitment schemes* are commitments of polynomials that are small and easy to "open" (evaluate at different points). KZG is one of the main polynomial commitment schemes being used in cryptography, such as in PLONK (coming up).

# §4 SNARKs

## §4.1 Introduction to SNARKs

Peggy has done some very difficult calculation. She wants to prove to Victor that she did it. Victor wants to check that Peggy did it, but he is too lazy to redo the whole calculation himself.

- Maybe Peggy wants to keep part of the calculation secret. Maybe her calculation was "find a solution to this puzzle," and she wants to prove that she found a solution without saying what the solution is.
- Maybe it's just a really long, annoying calculation, and Victor doesn't have the energy to check it all line-by-line.

A *SNARK* lets Peggy (the "prover") send Victor (the "verifier") a short proof that she has indeed done the calculation correctly. The proof will much shorter than the original calculation, and Victor's verification is much faster. (As a tradeoff, writing a SNARK proof of a calculation is much slower than just doing the calculation.)

We won't discuss it here, but it is also possible and frequently useful to make a *zero knowledge (zk)* SNARK. These are typically called "zkSNARKs." This gives Peggy a guarantee that Victor will not learn anything about the intermediate steps in her calculation, aside from any particular steps Peggy chooses to reveal.

### §4.1.1 What can you do with a SNARK?

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 message $M$ 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.

## §4.2 PLONK, a zkSNARK protocol

### §4.2.1 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 $\mathbb{F}_q$**.

In other words, PLONK is going to give us the ability to prove that we have solutions to a system of a system of quadratic equations.

**Situation 4.1**: Suppose we have a system of $m$ equations in $k$ variables $x_1, ..., x_k$:

$$\begin{aligned} Q_1(x_1, ..., x_k) &= 0 \\ &\vdots \\ Q_m(x_1, ..., x_k) &= 0. \end{aligned}$$

Of these $k$ variables, the first $\ell$ $(x_1, ..., x_\ell)$ have publicly known, fixed values; the remaining $\ell - k$ are unknown.

PLONK will let Peggy prove to Victor the following claim: I know $\ell - k$ values $x_{\ell+1}, ..., x_k$ such that (when you combine them with the $k$ public fixed values $x_1, ..., x_k$) the $\ell$ values $x_1, ..., x_k$ satisfy all $m$ quadratic equations.

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 4.2** (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.

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.

So, the first step in proving a claim like "I have a message $M$ such that $\text{sha}(M) = 0\text{xa91af3ac}\ldots$" is to translate the claim into a system of quadratic equations. This process is called "arithmetization."

One approach (suggested by Remark 4.2) is to represent each bit involved in the calculation by a variable $x_i$ (which would then be constrained to be either 0 or 1 by an equation $x_i^2 = x_i$). In this setup, the value 0xa91af3ac would be represented by 32 public bits $x_1, \ldots, x_{32}$; the unknown message $M$ would be represented by some private variables; and the calculation of sha would introduce a series of constraints, maybe involving some additional variables.

We won't get into any more details of arithmetization here.

### §4.2.2 An instance of PLONK

PLONK is actually going to prove solutions to systems of quadratic equations of a very particular form:

**Definition 4.3**: 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.

**Remark 4.4** (From Quad-SAT to PLONK): PLONK might look less general than Quad-SAT, but it turns out you can convert any Quad-SAT problem to PLONK.

First off, note that if we set

$$(q_{L,i}, q_{R,i}, q_{O,i}, q_{M,i}, q_{C,i}) = (1, 1, -1, 0, 0),$$

we get an "addition" gate $a_i + b_i = c_i$, while if we set

$$(q_{L,i}, q_{R,i}, q_{O,i}, q_{M,i}, q_{C,i}) = (1, 1, 0, -1, 0),$$

we get a "multiplication" gate $a_i b_i = c_i$. Finally, if $q$ is any constant, then

$$(q_{L,i}, q_{R,i}, q_{O,i}, q_{M,i}, q_{C,i}) = (1, 0, 0, 0, -q),$$

gives the constraint $a_i = q$.

Now imagine we want to encode some quadratic equation like $y = x^2 + 2$ in PLONK. We'll break this down into two steps:

$$\begin{aligned} x \cdot x &= (x^2) \quad \text{(multiplication)} \\ t &= 2 \quad \text{(constant)} \\ (x^2) + t &= y \quad \text{(addition)}. \end{aligned}$$

We'll assign the variables $a_i, b_i, c_i$ for these two gates by looking at the equations:

$$\begin{aligned} (a_1, b_1, c_1) &= (x, x, x^2) \\ (a_2, b_2, c_2) &= (t = 2, 0, 0) \\ (a_3, b_3, c_3) &= (x^2, t = 2, y). \end{aligned}$$

And finally, we'll assign copy constraints to make sure the variables are faithfully copied from line to line:

$$\begin{aligned} a_1 &= b_1 \\ c_1 &= a_3 \\ a_2 &= b_3. \end{aligned}$$

If the variables $a_i, b_i, c_i$ satisfy the gate and copy constraints, then $x = a_1$ and $y = c_3$ are forced to satisfy the original equation $y = x^2 + 2$.

Back to PLONK: Our protocol needs 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.

### §4.2.3 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 $n$th root of unity.

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

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

We specifically choose $\omega^i$ because that way, if we use Algorithm 3.26 on the set $\{\omega, \omega^1, ..., \omega^n\}$, then the polynomial called $Z$ is just $Z(X) = (X - \omega)...(X - \omega^n) = X^n - 1$, which is really nice. In fact, often $n$ is chosen to be a power of 2 so that $A$, $B$, and $C$ are very 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 4.5** (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 single value – a 256-bit elliptic curve point – that can later be "opened" at any value $x \in \mathbb{F}_q$.

### §4.2.4 Step 2: Gate-check

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

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

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

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

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

However, Peggy has committed $A$, $B$, $C$ already, while all the $Q_*$ polynomials are globally known. So this is a direct application of Algorithm 3.26:

**Algorithm 4.6** (Gate-check):

1. Both parties interpolate five polynomials $Q_* \in \mathbb{F}_q[X]$ from the $5n$ coefficients $q_*$ (globally known from the PLONK instance).
2. Peggy uses Algorithm 3.26 to convince Victor that Equation 4 holds for $X = \omega^i$ (that is, the left-hand side is is indeed divisible by $Z(X) := X^n - 1$).

### §4.2.5 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.

#### §4.2.5.1 Easier case: permutation-check

Suppose we have polynomials $P, Q \in \mathbb{F}_q[X]$ which encode 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

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

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: if the two sides of Equation 5 aren't the same polynomial, then the two sides can have at most $n - 1$ common values. So for a randomly chosen $\lambda$ (chosen from a field with $q \approx 2^{256}$ elements), the chances that $T = \lambda$ passes Equation 5 are extremely small.

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 prove Equation 5, the following algorithm works:

**Algorithm 4.7** (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 3.26 to prove all of the following statements:
   - $F_P(X) - (\lambda + P(X))$ vanishes at $X = \omega$;
   - $F_P(\omega X) - (\lambda + P(\omega X))F_P(X)$ 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 $X = 1$.

### §4.2.5.2 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:

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

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 $\eta^2$ doesn't happen to be a power of $\omega$, so all the tags are distinct. Specifically, if Equation 6 is true, then for any $\mu \in \mathbb{F}_q$, we also have

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

Now how can the prover establish Equation 7 succinctly? The answer is to run a permutation-check on the $3n$ entries of Equation 7! The prover will simply prove that the twelve matrix entries of the matrix on the left are a permutation of the twelve matrix entries of the matrix on the right.

The reader should check that this is correct! If the prover starts with values $a_i$, $b_i$, and $c_i$ that don't satisfy all the copy constraints, then a randomly selected $\mu$ is very unlikely to satisfy this permutation check. The right-hand side will not be a permutation of the left-hand side, and the check will fail.

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: We want to prove

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

is a permutation of

$$\begin{pmatrix} a_1 + \eta^2\omega^4\mu & b_1 + \eta\omega^1\mu & c_1 + \eta\omega^2\mu \\ a_2 + \omega^2\mu & b_2 + \eta^2\omega^1\mu & c_2 + \eta^2\omega^2\mu \\ a_3 + \omega^3\mu & b_3 + \eta\omega^3\mu & c_3 + \eta^2\omega^3\mu \\ a_4 + \omega^1\mu & b_4 + \eta\omega^4\mu & c_4 + \omega^4\mu \end{pmatrix}. \tag{8}$$

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:

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

Then the prover can start defining accumulator polynomials, after re-introducing the random challenge $\lambda$ 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

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

while the ones on the right have the extra permutation polynomials

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

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

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

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 4.8** (Copy-check):

0. Peggy has already sent the three commitments $\text{Com}(A), \text{Com}(B), \text{Com}(C)$ to Victor; these commitments bind her to the values of all the variables $a_i$, $b_i$, and $c_i$.
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 3.26 to prove Equation 11 holds.
5. Peggy uses Algorithm 3.26 to prove Equation 12 holds for $X \in \{\omega, \omega^2, ..., \omega^{n-1}\}$.
6. Peggy uses Algorithm 3.26 to prove Equation 13 holds.

### §4.2.6 Public and private witnesses

**TODO**: (Gub ignore, Aard and Evan to discuss) 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 3.26. 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, $A^{\text{public}}$, 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 $A^{\text{public}}$, since all the values $a_i$ are publicly known!

Now the prover runs Algorithm 3.26 to prove that $A - A^{\text{public}}$ vanishes on $S$. (And similarly for $B$ and $C$, if needed.) And we're done.

## §4.3 Making it non-interactive: Fiat–Shamir

**TODO**: Gub could you edit this section for good explaininess?

As we described it, PLONK is an interactive protocol. Peggy sends Victor some data; Victor reads that data and sends back a random challenge. Peggy sends back some more data; Victor replies with more challenges. After a few rounds of this, the protocol is complete, and Victor is convinced of the truth of Peggy's claim.

We want to turn this into a non-interactive protocol. Peggy sends Victor some data once. Victor reads the data, does some calculation, and convinces himself of the truth of Peggy's claim.

We will do this using a general trick, called the "Fiat–Shamir heuristic."

Let us step back and philosophize for a moment. Why does Victor need to send challenges at all? This is actually what makes the whole SNARK thing work. Peggy condenses a long calculation down into a very short proof, which she sends to Victor. What keeps her from cheating is that she has to be prepared to respond to any challenge Victor could possibly send back. If Peggy knew what challenge Victor was going to send, Peggy could use that foreknowledge to create a false proof. But by sending a random challenge after Peggy's original commitment, Victor prevents her from adapting her proof to the challenge.

The idea of Fiat and Shamir is to replace Victor's random number generator with a (cryptographically secure) hash function. Instead

of interacting with Victor, Peggy simply runs this hash function to generate the challenge for each round.

For example, consider the following (slightly simplified) version of Algorithm 3.26.

**Algorithm 4.9**: Peggy wants to prove to Victor that two polynomials $F$ and $H$ (known only to Peggy) satisfy $F(X) = Z(X)H(X)$, where $Z(X) = \prod_{z \in S}(X - z)$ is a fixed polynomial known to both Peggy and Victor.

1. Peggy sends $\mathrm{Com}(F)$ and $\mathrm{Com}(H)$.
2. Victor picks a random challenge $\lambda \in \mathbb{F}_q$
3. Peggy opens both $\mathrm{Com}(F)$ and $\mathrm{Com}(H)$ at $\lambda$.
4. Victor verifies $F(\lambda) = Z(\lambda)H(\lambda)$.

Fiat–Shamir turns it into the following noninteractive protocol.

**Algorithm 4.10**: Peggy wants to prove to Victor that two polynomials $F$ and $H$ (known only to Peggy) satisfy $F(X) = Z(X)H(X)$, where $Z(X) = \prod_{z \in S}(X - z)$ is a fixed polynomial known to both Peggy and Victor.

1. Peggy sends $\mathrm{Com}(F)$ and $\mathrm{Com}(H)$.
2. Peggy computes $\lambda \in \mathbb{F}_q$ by $\lambda = \mathrm{hash}(\mathrm{Com}(F), \mathrm{Com}(H))$.
3. Peggy opens both $\mathrm{Com}(F)$ and $\mathrm{Com}(H)$ at $\lambda$.
4. Victor verifies that $\lambda = \mathrm{hash}(\mathrm{Com}(F), \mathrm{Com}(H))$ and $F(\lambda) = Z(\lambda)H(\lambda)$.

We can apply the Fiat–Shamir heuristic to the full PLONK protocol. Now Peggy can write the whole proof herself (without waiting for Victor's challenges), and publish it. Victor can then verify the proof at leisure.

## §4.4 SNARK Takeaways

1. A *SNARK* can be used to succinctly prove that a piece of computation has been done correctly; specifically, it proves to some Verifier that the Prover had the K(nowledge) of some information that worked as feasible inputs to some computational circuit.
2. The *arithmetization* of the circuit is a way of converting circuits to arithmetic. Specifically for PLONK (but also other SNARKs, e.g. Groth16), our arithmetization is systems of quadratic equations over $F_q$, meaning that what PLONK does under the hood is proving that a system of these equations are satisfied.
3. The work under the hood of PLONK comes down to polynomial commitments (specifically KZG). The constructions "powered by" KZG power concepts such as copy checks and permutation checks, which are key to PLONK.
4. The N(oninteractivity) of SNARKs basically come down to the *Fiat-Shamir heuristic*, which is very common in this field. Generally speaking, the "meat" of zkSNARKs are mostly about S(uccinctness) of the AR(guments).

# §5 Fully Homomorphic Encryption

## §5.1 Introduction to FHE

Alice has a secret $x$, and Bob has a function $f$. They want to compute $f(x)$. Actually, Alice wants Bob to compute $f(x)$ – but she doesn't want to tell him $x$.

Alice wants to encrypt $x$ and send Bob $\text{Enc}(x)$. Then Bob is going to "apply $f$ to the ciphertext", to turn $\text{Enc}(x)$ into $\text{Enc}(f(x))$. Finally, Bob sends $\text{Enc}(f(x))$ back, and Alice decrypts it to learn $f(x)$. This is *fully homomorphic encryption (FHE)*.

*Levelled FHE* is a sort of weaker version of FHE. Like FHE, levelled FHE lets you perform operations on encrypted data. But unlike FHE, there will be a limit on the number of operations you can perform before the data must be decrypted.

Loosely speaking, the encryption procedure will involve some sort of "noise" or "error." As long as the error is not too big, the message can be decoded without trouble. But each operation on the encrypted data will cause the error to grow – and if it grows beyond some maximum error tolerance, the message will be lost. So there is a limit on how many operations you can do before the error gets too big.

As a sort of silly example, imagine your message is a whole number between 0 and 10 (so it's one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10), and your "encryption" scheme encodes the message as a real number that is very close to the message. So if the ciphertext is 1.999832, well then that means the original message was 2. The decryption procedure is "round to the nearest integer."

(You might be thinking: This is some pretty terrible cryptography, because the message isn't secure. Anyone can figure out how to round a number, no secret key required. Yep, you're right. The actual encryption scheme is more complicated. But it still has this "rounding-off-errors" feature, and that's what I want to focus on right now.)

Now imagine that the "operations" you want to perform are addition. (If you like, imagine doing the addition modulo 11, so if a number gets too big, it "wraps around.") Well, every time you add two encrypted numbers ($1.999832 + 2.999701 = 4.999533$), the errors add as well. After too many operations, the error will exceed 0.5, and the rounding procedure won't give the right answer anymore. But as long as you're careful not to go over the error limit, you can add ciphertexts with confidence.

For our levelled FHE protocol, our message will be a bit (either 0 or 1) and our operations will be the logic gates AND and NOT. Because any logic circuit can be built out of AND and NOT gates, we'll be able to perform arbitrary calculations within the FHE encryption.

Our protocol uses a cryptosystem built from a problem called "learning with errors." "Learning with errors" is kind of a strange name; we'd call it "approximate linear algebra modulo $q$." Anyway, we'll start with the learning-with-errors problem (Section 5.2) and how to build cryptography on top of it (Section 5.3) before we get back to levelled FHE.

## §5.2 A hard problem: learning with errors

As we've seen (Section 3.1), a lot of cryptography relies on hard math problems. RSA is based on the difficulty of integer factorization; elliptic curve cryptography depends on the discrete log problem.

Our protocol for levelled FHE relies on a different hard problem: the learning with errors problem (LWE). The problem is to solve systems of linear equations, except the equations are only approximately true – they permit a small "error" – and instead of solving for rational or real numbers, you're solving for integers modulo $q$.

Here's a concrete example of a LWE problem and how one might attack it "by hand." This exercise will make the inherent difficulty of the problem quite intuitive.

> *Problem 5.1*: We are working over $\mathbb{F}_{11}$, and there is some secret vector $a = (a_1, ..., a_4)$. There are two sets of claims. Each claim "$(x_1, ..., x_4) : y$" purports the relationship

$$y = a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + \varepsilon, \qquad \varepsilon \in \{0, 1\}.$$

(The $\varepsilon$ is different from equation to equation.)

One of the sets of claims is "genuine" and comes from a consistent set of $a_i$, while the other set is "fake" and has randomly generated $y$ values. Tell them apart and find the correct secret vector $(a_1, ..., a_4)$.

| Blue Set | Red Set |
|---|---|
| (1, 0, 1, 7) : 2 | (5, 4, 5, 2) : 2 |
| (5, 8, 4, 10) : 9 | (7, 7, 7, 8) : 5 |
| (7, 7, 8, 5) : 3 | (6, 8, 2, 2) : 0 |
| (5, 1, 10, 6) : 3 | (10, 4, 4, 3) : 1 |
| (8, 0, 2, 4) : 1 | (1, 10, 8, 6) : 6 |
| (9, 3, 0, 6) : 9 | (2, 7, 7, 4) : 4 |
| (0, 6, 1, 6) : 9 | (8, 6, 6, 9) : 1 |
| (0, 4, 9, 7) : 5 | (10, 6, 1, 6) : 9 |
| (10, 7, 4, 10) : 10 | (3, 1, 10, 9) : 7 |
| (5, 5, 10, 6) : 8 | (2, 4, 10, 3) : 7 |
| (10, 7, 3, 1) : 9 | (10, 4, 6, 4) : 2 |
| (0, 2, 5, 5) : 6 | (8, 5, 7, 2) : 2 |
| (9, 10, 2, 1) : 2 | (4, 7, 0, 0) : 8 |
| (3, 7, 2, 1) : 5 | (0, 3, 0, 0) : 0 |
| (2, 3, 4, 5) : 3 | (8, 3, 2, 7) : 8 |
| (2, 1, 6, 9) : 3 | (4, 6, 6, 3) : 2 |

(solution sketch; can be skipped safely): One way to start would be to define an *information vector*

$$(x_1, x_2, x_3, x_4|y|S),$$

where $S \subset F_{11}$, to mean the statement

$$\sum a_i x_i = y + s, \text{ where } s \in S.$$

In particular, a purported approximation $(x_1, x_2, x_3, x_4) : y$ in the LWE protocol corresponds to the information vector

$$(x_1, x_2, x_3, x_4 | y | \{0, -1\}).$$

The benefit of this notation is that we can take linear combinations of them. Specifically, if $(X_1|y_1|S_1)$ and $(X_2|y_2|S_2)$ are information vectors (where $X_i$ are vectors), then

$$(\alpha X_1 + \beta X_2 | \alpha y_1 + \beta y_2 | \alpha S_1 + \beta S_2),$$

where $\alpha S = \{\alpha s \mid s \in S\}$ and $S + T = \{s + t \mid s \in S, t \in T\}$.

We can observe the following:

1. If we obtain two vectors $(X|y|S_1)$ and $(X|y|S_2)$, then we have the information (assuming the vectors are accurate) $(X|y|S_1 \cap S_2)$. So if we are lucky enough, say, to have $|S_1 \cap S_2| = 1$, then we have found an actual equation with no error.
2. As we linearly combine vectors, their "error part" $S$ gets bigger exponentially. So we can only add vectors very few times, ideally just 1 or 2 times, before they start being unusable.

With these heuristics, we can start by looking at the Red Set, and make vectors with many 0's in the same places.

1. Our eyes are drawn to the juicy-looking $(0, 3, 0, 0|0|\{0, -1\})$, which immediately gives $a_2 \in \{0, 7\}$.
2. $(4, 7, 0, 0|8|\{0, -1\})$ gives $4a_1 + 7a_2 \in \{7, 8\}$, Since $7a_2 \in \{0, 5\}$,

$$4a_1 \in \{7, 8\} - \{0, 5\} = \{7, 8, 2, 3\},$$

   and $a_1 \in \{10, 2, 6, 9\}$.
3. Adding

$$(10, 4, 4, 3|1|\{0, -1\}) + (7, 7, 7, 8|5|\{0, -1\})$$

gives $(6, 0, 0, 0, 0|6|\{0, -1, -2\})$, which is nice because it has 3 zeroes! This gives $a_1 \in \{1, 8, 10\}$. Combining with (2), we conclude that $a_1 = 10$.

4. ...

We omit the rest of the solution, which makes for some fun tinkering.

## §5.3 Public-Key Cryptography from LWE

The learning with errors problem (Section 5.2), like the discrete log assumption, is one of those "hard problems that you can build cryptography on." The problem is to solve for constants

$$a_1, ..., a_n \in \mathbb{Z}/q\mathbb{Z},$$

given a bunch of *approximate* equations of the form

$$a_1 x_1 + ... + a_n x_n = y + \epsilon,$$

where each $\epsilon$ is a "small" error (in the linked example, $\epsilon$ is either 0 or 1).

In Section 5.2 we saw how even a small case of this problem ($q = 11$, $n = 4$) can be annoyingly tricky. In the real world, you should imagine that $n$ and $q$ are much bigger – maybe $n$ is in the range $100 \leq n \leq 1000$, and $q$ could be anywhere from $n^2$ to $2^{\sqrt{n}}$, say.

Now let's see how to turn this into a public-key cryptosystem. We'll use the same numbers from the "blue set" in Section 5.2. In fact, that "blue set" will be exactly the public key.

| Public Key |
|---|
| (1, 0, 1, 7) : 2 |
| (5, 8, 4, 10) : 9 |
| (7, 7, 8, 5) : 3 |
| (5, 1, 10, 6) : 3 |
| (8, 0, 2, 4) : 1 |
| (9, 3, 0, 6) : 9 |
| (0, 6, 1, 6) : 9 |
| (0, 4, 9, 7) : 5 |
| (10, 7, 4, 10) : 10 |
| (5, 5, 10, 6) : 8 |
| (10, 7, 3, 1) : 9 |
| (0, 2, 5, 5) : 6 |
| (9, 10, 2, 1) : 2 |
| (3, 7, 2, 1) : 5 |
| (2, 3, 4, 5) : 3 |
| (2, 1, 6, 9) : 3 |

The private key is simply the vector $a$.

| Private Key |
|---|
| $\mathbf{a}$ = (10, 8, 10, 10) |

### §5.3.1 How to encrypt $\mu$?

Suppose you have a message $m \in \{0, 5\}$. (You'll see in a moment why we insist that $\mu$ is one of these two values.) The ciphertext to encrypt $m$ will be a pair $(\mathbf{x} : y)$, where $x$ is a vector, $y$ is a scalar, and $\mathbf{x} \cdot \mathbf{a} + \epsilon = y + \mu$, where $\epsilon$ is "small".

How to do the encryption? If you're trying to encrypt, you only have access to the public key – that list of pairs $(\mathbf{x} : y)$ above. You want to make up your own $\mathbf{x}$, for which you know approximately the value $\mathbf{x} \cdot \mathbf{a}$. You could just take one of the vectors $\mathbf{x}$ from the table, but

that wouldn't be very secure: if I see your ciphertext, I can find that $\mathbf{x}$ in the table and use it to decrypt $\mu$.

Instead, you are going to combine several rows of the table to get your vector $\mathbf{x}$. Now you have to be careful: when you combine rows of the table, the errors will add up. We're guaranteed that each row of the table has $\epsilon$ either 0 or 1. So if you add at most 4 rows, then the total $\epsilon$ will be at most 4. Since $\mu$ is either 0 or 5 (and we're working modulo $q = 11$), that's just enough to determine $\mu$ uniquely.

So, here's the method. You choose at random 4 (or fewer) rows of the table, and add them up to get a pair $(\mathbf{x} : y_0)$ with $\mathbf{x} \cdot \mathbf{a} \approx y_0$. Then you take $y = y_0 - \mu$ (mod $q = 11$ of course), and send the message $(\mathbf{x} : y)$.

### §5.3.2 An example

Let's say you randomly choose the first 4 rows:

| Some rows of public key |
|---|
| (1, 0, 1, 7) : 2 |
| (5, 8, 4, 10) : 9 |
| (7, 7, 8, 5) : 3 |
| (5, 1, 10, 6) : 3 |

Now you add them up to get the following.

| $\mathbf{x} : y_0$ |
|---|
| (7, 5, 1, 6) : 6 |

Finally, let's say your message is $m = 5$. So you set $y = y_0 - m = 6 - 5 = 1$, and send the ciphertext:

| $\mathbf{x} : y$ |
|---|
| (7, 5, 1, 6) : 1 |

### §5.3.3 Decryption

Decryption is easy! The decryptor knows

$$\mathbf{x} \cdot \mathbf{a} + \epsilon = y + \mu$$

where $0 \leq \epsilon \leq 4$.

Plugging in $\mathbf{x}$ and $\mathbf{a}$, the decryptor computes

$$\mathbf{x} \cdot \mathbf{a} = 4.$$

Plugging in $y = 1$, we see that

$$4 + \epsilon = 1 + \mu.$$

Now it's a simple "rounding" problem. We know that $\epsilon$ is small and positive, so $1 + \mu$ is either 4 or ... a little more. (In fact, it's one of $4, 5, 6, 7, 8$.) On the other hand, since $\mu$ is 0 or 5, well, $1 + \mu$ had better be 1 or 6... so the only possibility is that $1 + \mu = 6$, and $\mu = 5$.

### §5.3.4 How does this work in general?

In practice, $n$ and $q$ are often much larger. Maybe $n$ is in the hundreds, and $q$ could be anywhere from "a little bigger than $n$" to "almost exponentially large in $n$," say $q = 2^{\sqrt{n}}$. In fact, to do FHE, we're going to want to take $q$ pretty big, so you should imagine that $q \approx 2^{\sqrt{n}}$.

For security, the encryption algorithm shouldn't just take add up 3 or 4 rows of the public key. In fact we want the encryption algorithm to add at least $\log(q^n) = n \log q$ rows – to be safe, maybe make that number a little bigger, say $m = 2n \log q$. Of course, for this to work, the public key has to have at least $m$ rows.

So in practice, the public key will have $m = 2n \log q$ rows, and the encryption algorithm will be "select some subset of the rows at random, and add them up".

Of course, combining $m$ rows will have the effect of multiplying the error by $m$ – so if the initial $\epsilon$ was bounded by 1, then the error in the ciphertext will be at most $m$. But remember that $q$ is exponen-

tially large compared to $m$ and $n$ anyway, so a mere factor of $m$ isn't going to scare us!

Now we could insist that the message is just a single bit – either 0 or $\lfloor \frac{q}{2} \rfloor$. Or we could allow the message to be any multiple of some constant $r$, where $r$ is bigger than the error bound (right now that's $m$) – which allows you to encode a message space of size $q/r$ rather than just a single bit.

When we do FHE, we're going to apply many operations to a ciphertext, and each is going to cause the error to grow. We're going to have to put some effort into keeping the error under control – and the size of $q/r$ will determine how many operations we can do before the error grows too big.

## §5.4 Levelled Fully Homomorphic Encryption from Learning with Errors

### §5.4.1 The main idea: Approximate eigenvalues

Now we want to turn the public-key encryption from Section 5.3 into a levelled FHE scheme. In other words: We want to be able to encrypt bits (0s and 1s) and operate on them with AND and NOT gates.

It might help you to imagine that, instead of AND and NOT, the operations we want to encrypt are addition and multiplication. If $x$ and $y$ are bits, then NOT $x$ is just $1 - x$, and $x$ AND $y$ is just $xy$. But it's easier to do algebra with $+$ and $\times$.

Recall the setup from Section 5.3: We're going pick some large integer $q$ (in practice $q$ could be anywhere from a few thousand to $2^{1000}$), and do "approximate linear algebra" modulo $q$. In other words, we'll do linear algebra, where all our calculations are done modulo $q$ – but we'll also allow the calculations to have a small "error" $\epsilon$, which will typically be much, much smaller than $q$.

Here's the new idea. Our *secret key* will be a vector

$$\mathbf{v} = (v_1, ..., v_n) \in (\mathbb{Z}/q\mathbb{Z})^n$$

– a vector of length $n$, where the entries are integers modulo $q$. Suppose we want to encode a message $\mu$ that's just a single bit, let's say $\mu \in \{0, 1\}$. Our ciphertext will be a square $n$-by-$n$ matrix $C$ such that

$$C\mathbf{v} \approx \mu\mathbf{v}.$$

Now if we assume that $\mathbf{v}$ has at least one "big" entry (say $v_i$), then decryption is easy: Just compute the $i$-th entry of $C\mathbf{v}$, and determine whether it is closer to 0 or to $v_i$.

With a bit of effort, it's possible to make this into a public-key cryptosystem. Just like in Section 5.3, the main idea is to release a table of vectors $\mathbf{x}$ such that

$$\mathbf{x} \cdot \mathbf{v} \approx 0,$$

and use that as a public key. Given $\mu$ and the public key, you can find a matrix $C_0$ such that

$$C_0\mathbf{v} \approx 0$$

– then take

$$C = C_0 + \mu\mathrm{Id}$$

, where Id is the identity matrix. And $C_0$ can be built row-by-row... but we won't get into the details here.

Indeed homomorphic encryption is already interesting without the public-key feature. If you assume the person encrypting the data knows $\mathbf{v}$, it's easy (linear algebra, again) to find $C$ such that

$$C\mathbf{v} \approx \mu\mathbf{v}.$$

To make homomorphic encryption work, we need to explain how to operate on $\mu$. We'll describe three operations: addition, NOT, and multiplication (aka AND).

Addition is simple: Just add the matrices. If $C_1\mathbf{v} \approx \mu_1\mathbf{v}$ and $C_2\mathbf{v} \approx \mu_2\mathbf{v}$, then

$$(C_1 + C_2)\mathbf{v} = C_1\mathbf{v} + C_2\mathbf{v} \approx \mu_1\mathbf{v} + \mu_2\mathbf{v} = (\mu_1 + \mu_2)\mathbf{v}.$$

Of course, addition on bits isn't a great operation, because if you add $1 + 1$, you get 2, and 2 isn't a legitimate bit anymore. So we won't really use this.

Negation of a bit (NOT) is equally simple, though. If $\mu \in \{0, 1\}$ is a bit, then its negation is simply $1 - \mu$. And if $C$ is a ciphertext for $\mu$, then $\mathrm{Id} - C$ is a ciphertext for $1 - \mu$, since

$$(\mathrm{Id} - C)\mathbf{v} = \mathbf{v} - C\mathbf{v} \approx (1 - \mu)\mathbf{v}.$$

Multiplication is also a good operation on bits – it's just AND. To multiply two bits, you just multiply (matrix multiplication) the ciphertexts:

$$C_1 C_2 \mathbf{v} \approx C_1(\mu_2 \mathbf{v}) = \mu_2 C_1 \mathbf{v} \approx \mu_2 \mu_1 \mathbf{v} = \mu_1 \mu_2 \mathbf{v}.$$

(At this point you might be concerned about this symbol $\approx$ and what happens to the size of the error. That's an important issue, and we'll come back to it.)

Anyway, once you have AND and NOT, you can build arbitrary logic gates – and this is what we mean when we say you can perform arbitrary calculations on your encrypted bits, without ever learning what those bits are. At the end of the calculation, you can send the resulting ciphertexts back to be decrypted.

### §5.4.2 A constraint on the secret key $\mathbf{v}$ and the "Flatten" operation

In order to make the error estimates work out, we're going to need to make it so that all the ciphertext matrices $C$ have "small" entries. In fact, we will be able to make it so that all entries of $C$ are either 0 or 1.

To make this work, we will assume our secret key $\mathbf{v}$ has the special form

$$\begin{aligned}\mathbf{v} = \big(&a_1, 2a_1, 4a_1, ..., 2^k a_1,\\ &a_2, 2a_2, 4a_2, ..., 2^k a_2,\\ &\vdots\\ &a_r, 2a_r, 4a_r, ..., 2^k a_r\big),\end{aligned} \tag{14}$$

where $k = \lfloor \log_2 q \rfloor$.

To see how this helps us, try the following puzzle. Assume $q = 11$ (so all our vectors have entries modulo 11), and $r = 1$, so our secret key has the form

$$\mathbf{v} = (a_1, 2a_1, 4a_1, 8a_1).$$

You know $\mathbf{v}$ has this form, but you don't know the specific value of $a_1$.

Now suppose I give you the vector

$$\mathbf{x} = (9, 0, 0, 0).$$

I ask you for another vector

$$\text{Flatten}(\mathbf{x}) = \mathbf{x}',$$

where $\mathbf{x}'$ has to have the following two properties:

- $\mathbf{x}' \cdot \mathbf{v} = \mathbf{x} \cdot \mathbf{v}$, and
- All the entries of $\mathbf{x}'$ are either 0 or 1.

And you have to find this vector $\mathbf{x}'$ without knowing $a_1$.

The solution is to use binary expansion: take $\mathbf{x}' = (1, 0, 0, 1)$. You should check for yourself to see why this works – it boils down to the fact that $(1, 0, 0, 1)$ is the binary expansion of 9.

How would you flatten a different vector, like

$$\mathbf{x} = (9, 3, 1, 4)?$$

I'll leave this as an exercise to you! As a hint, remember we're working with numbers modulo 11 – so if you come across a number that's bigger than 11 in your calculation, it's safe to reduce it mod 11.

In general, if you know $\mathbf{v}$ has the form in Equation 14 and you are given some matrix $C$ with coefficients in $\mathbb{Z}/q\mathbb{Z}$, then you can compute another matrix $\text{Flatten}(C)$ such that:

- $\text{Flatten}(C)\mathbf{v} = C\mathbf{v}$, and
- All the entries of $\text{Flatten}(C)$ are either 0 or 1.

The Flatten process is essentially the same binary-expansion process we used above to turn $\mathbf{x}$ into $\mathbf{x}'$, applied to each $k+1$ entries of each row of the matrix $C$.

So now, using this Flatten operation, we can insist that all of our ciphertexts $C$ are matrices with coefficients in $\{0, 1\}$. For example, to multiply two messages $\mu_1$ and $\mu_2$, we first multiply the corresponding ciphertexts, then flatten the resulting product:

$$\text{Flatten}(C_1 C_2).$$

Of course, revealing that the secret key $\mathbf{v}$ has this special form will degrade security. This cryptosystem is as secure as an LWE problem on vectors of length $r$, not $n$. So we need to make $n$ bigger, say $n \approx r \log q$, to get the same level of security.

### §5.4.3 Error analysis

Now let's compute more carefully what happens to the error when we add, negate, and multiply bits. Suppose

$$C_1\mathbf{v} = \mu_1\mathbf{v} + \epsilon_1,$$

where $\epsilon_1$ is some vector with all its entries bounded by a bound $B$. (And similarly for $C_2$ and $\mu_2$.)

When we add two ciphertexts, the errors add:

$$(C_1 + C_2)\mathbf{v} = (\mu_1 + \mu_2)\mathbf{v} + (\epsilon_1 + \epsilon_2).$$

So the error on the sum will be bounded by $2B$.

Negation is similar to addition – in fact, the error won't change at all.

Multiplication is more complicated, and this is why we insisted that all ciphertexts have entries in $\{0, 1\}$. We compute

$$C_1 C_2 \mathbf{v} = C_1(\mu_2 \mathbf{v} + \epsilon_2) = \mu_1 \mu_2 \mathbf{v} + (\mu_2 \epsilon_1 + C_1 \epsilon_2).$$

Now since $\mu_2$ is either 0 or 1, we know that $\mu_2 \epsilon_1$ is a vector with all entries bounded by $B$. What about $C_1 \epsilon_2$? Here you have to think for a second about matrix multiplication: when you multiply an $n$-by-$n$ matrix by a vector, each entry of the product comes as a sum of $n$ different products. Now we're assuming that $C_1$ is a $0-1$ matrix, and all entries of $\epsilon_2$ are bounded by $B$... so the product has all entries bounded by $nB$. Adding this to the error for $\mu_2 \epsilon_1$, we get that the total error in the product $C_1 C_2 \mathbf{v}$ is bounded by $(n+1)B$.

In summary: We can start with ciphertexts having a very small error (if you think carefully about this protocol, you will see that the error is bounded by approximately $n \log q$). Every addition operation will double the error bound; every multiplication (AND gate) will multiply it by $(n+1)$. And you can't allow the error to exceed $q/2$ – otherwise the message cannot be decrypted. So you can perform calculations of up to approximately $\log_n q$ steps. (In fact, it's a question of *circuit depth*: you can start with many more than $\log_n q$ input bits, but no bit can follow a path of length greater than $\log_n q$ AND gates.)

This gives us a *levelled* fully homomorphic encryption protocol: it lets us evaluate arbitrary circuits on encrypted data, as long as those circuits have bounded depth. If we need to evaluate a bigger circuit, we have two options.

1. Increase the value of $q$. Of course, the cost of the computations increases with $q$.
2. Use some technique to "reset" the error and start anew, as if with a freshly encrypted ciphertext.

   This approach is called "bootstrapping" and it incurs some hefty computational costs. But for very, very large circuits, it's the only viable option.

Bootstrapping is beyond the scope of this book.

## §5.5 FHE Takeaways

1. A *fully homomorphic encryption* protocol allows Alice to delegate Bob to compute some function $f(x)$ for Alice in a way that Bob doesn't get to know $x$.
2. The hard problem backing known FHE protocols is the *learning with errors (LWE)* problem, which comes down to deciding if a system of "approximate equations" over $F_q$ are consistent.
3. The main idea of this approach to FHEs is to use approximate eigenvalues as the encrypted computation and an "approximate eigenvector" as the secret key. Intuitively, adding and multiplying two matrices with different approximate eigenvalues for the same eigenvector approximately adds and multiplies the eigenvalues, respectively.
4. To carefully do this, we actually need to control the error blowup with the *flatten* operation. This creates a *leveled FHE* protocol.