Programmable Cryptography: Four Easy Pieces

Edited by Evan Chen, Brian Lawrence, Yan X Zhang

February 11, 2025

Contents

1 Introduction

Brian Gu and Yan X Zhang

1.1 What is programmable cryptography?

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

To be concrete, let us consider two examples of what protocols designed by classical cryptography can do:

- *Digital signatures.* RSA and ElGamal are examples of digital signature algorithms, where Alice can perform some protocol to prove to Bob that she endorses a message. A more complicated example might be a group signature scheme,[1] which allows one member of a group to sign a message on behalf of the group.
- *Confidential computing.* For example, consider Yao's millionaire problem,[2] where Alice and Bob want to know which of them makes more money without learning anything more about each other's incomes. With cryptography, Alice and Bob could use a two-party computation protocol designed specifically for this purpose.

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

> We want cryptography 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 is like going from 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.

Remark 1.1. The quote on the title page ("I have a message M such that $\text{sha}(M) = \text{"}0x91af3ac...\text{"}$") 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 SHA-256.

[1] https://en.wikipedia.org/wiki/Group_signature
[2] https://en.wikipedia.org/wiki/Yao%27s_Millionaires%27_problem

1.2 Ideas in programmable cryptography

Our work presents programmable cryptography through specific topics in several self-contained "easy pieces," imitating Richard Feynman's wonderful approach to physics exposition. We quickly preview them here.

1.2.1 2PC: two-party computation

In a *two-party computation (2PC)*, 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, 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 their own amounts. This is the case where F is the comparison function ($F(x_1, x_2)$ is 1 if $x_1 > x_2$, 2 if $x_2 > x_1$, and 0 if the two inputs are equal), and x_i is the i-th person's income.

Two-party computation makes a promise that we will 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 SNARKs: proofs of general statements

A powerful way of thinking about a signature scheme is that it is a **proof**. Specifically, Alice's signature is a proof that "I [the person who generated the signature] know Alice's private key." Similarly, a group signature can be thought of as a succinct proof that "I know one of Alice, Bob, or Charlie's private keys."

In the spirit of programmable cryptography, a *SNARK* generalizes this concept as a "proof system" protocol that produces efficient proofs of **arbitrary** statements of the form:

"I know X such that $F(X, Y) = Z$, where F, Y, Z are public," once the statement is encoded as a system of equations. One such statement would be "I know M such that $\text{sha}(M) = Y$."

SNARKs are an active area of research, and many different SNARKs are known. We will focus on a particular example, PLONK (Section 4.2).

1.2.3 FHE: fully homomorphic encryption

Imagine you have some private text that you want to translate into another language. While many services today will do this, even for free, we can also imagine that you care about security a lot and you really do not want the translating service to know anything about your text at all.

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

With this technology, you have a solution to your problem! You simply encrypt your text $\text{Enc}(x)$ and send it to your FHE machine translation server. The server will faithfully translate it into another language and give you $\text{Enc}(y)$, where y is the translation of x. You can then decrypt and obtain y, knowing that the server cannot extract anything meaningful from $\text{Enc}(x)$ without your secret key.

1.2.4 ORAM: Oblivious RAM

You want to perform a private computation on a large database. The database is so large that you cannot store it yourself – and you do not trust the server it is stored on.

First off, you will encrypt the data, so the server cannot read it. But the server still has an attack: They can study your *access patterns*. For example, they can see which records you access most frequently, or which records you access at the same time as other records. In many applications this is enough for the server to learn sensitive information.

Oblivious RAM (ORAM) protects against exactly this sort of attack. Oblivious RAM is an algorithm you use to "scramble" your memory access requests. When you feed your request into the ORAM algorithm, the ORAM algorithm sends some scrambled read and write requests to the server. Only one of the scrambled requests is the request you are interested in; the others keep the server from learning which request you care about.

1.3 Programmable cryptography in the world

In the past decade, there has been a surprisingly high amount of theoretical work but also a surprisingly low amount of implementation work on primitives in programmable cryptography. However, recent advances in areas like blockchain and other decentralized systems are rapidly driving demand for practical implementations of programmable cryptography. The gap that is being revealed right now, as theory meets reality, is exciting and enlightening.

Many of the protocols we mention in this book can be implemented today, but only at a very high cost (for example, the cost of proving a computation in a SNARK can be millions of times the cost of performing the computation directly). As we study the theory of programmable cryptography, it is useful to keep in mind some practical questions. Can we reduce the theoretical overhead of programmable cryptography? How can we make programmable cryptography systems more performant for modern hardware and software systems? What other systems or applications can be built on top of this technology?

It is easy to be carried away by the staggering possibilities, and to imagine a perfect "post-cryptographic" world where everyone has control over all their data and everyone's security preferences are completely fulfilled. It is also easy to be cynical and assume that these ideas will go nowhere. Reality is always somewhere in the middle; the Internet today offers free search and

civilization-scale repositories of information to everyone, but is also used for plenty of frivolous or even antisocial activity.

No matter what the future actually holds, one thing is clear - it is up to people who are capable, curious, and optimistic to guide the next stage of the evolution of cryptography-based systems. We hope that these "easy pieces" will inspire you to read, imagine, and build.

2 Two-Party Computation

Brian Gu and Brian Lawrence

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 would 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 should not 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. The protocol is quite clever, and optimized variants of the protocol are being implemented and used today.[3]

2.1.1 The problem

Here is our problem setting, slightly more formally:

- Alice knows a secret bitstring a of length m bits.
- Bob knows a secret bitstring b of length n bits.
- f is a binary circuit, which takes in $m + n$ bits, and runs them through some k gates. The outputs of some of the gates are the public outputs of the circuit. Without loss of generality, let us also suppose that each gate in f accepts either 1 or 2 input bits, and outputs a single output bit.
- Alice and Bob would like to jointly compute $f(a, b)$ without revealing their secrets to each other.

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.

[3] https://github.com/privacy-scaling-explorations/mpz/tree/dev/crates/mpz-garble

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

In slightly more detail:

1. Whatever the function f is, we will assume that it takes $m + n$ bits of input $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$, and that it is 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 $b_1, \ldots, b_n$.

3. Now, Alice is going to "garble" the circuit f_a. Once it is garbled, Bob will not be able to see how it works. He will only be able to plug his own input $(b_1, \ldots, b_n)$ 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 $(b_1, \ldots, b_n)$, he can learn $f_a(b_1, \ldots, b_n) = f(a, b)$, but he will not 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 $(b_1, \ldots, b_n)$? Alice does not want to let Bob have any other passwords – and Bob is not 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 us 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*[4] encryption scheme Enc and publish the following table:

[4]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 (or someone else) need the same secret key K to decrypt it.

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

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 do not know the password $P_{x,y}$, assuming Enc is a suitably secure encryption scheme, you will not learn anything about the value $G_{x,y}$ from its encryption.

2.1.4 Chaining garbled gates

The next step is to combine a bunch of garbled gates into a circuit. We will 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^{left} – and then another bit coming in for y, with some password P_y^{right}. To get the combined password $P_{x,y}$, Bob concatenates the two passwords P_x^{left} and P_y^{right}.

2. To keep the functionality of the circuit hidden, we do not 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 will have the gate just output 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^{left} and P_y^{right}, not by the bits (x,y). But we do not want Bob to learn the other passwords from the table!

Let us say this again. We want:

- If Bob knows both passwords P_x^{left} and P_y^{right}, Bob can find the row of the table for the input (x,y).
- If Bob does not know the passwords, he cannot 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, we will assume it is an AND gate – so the outputs will be (the passwords encoding) 0, 0, 0, 1.

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

2.1.5 How Bob uses one gate

Let us 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 does not know he has 0 and 1 (but we do!). Bob knows his left password is some value P_0^{left}, and his right password is some other value P_1^{right}.

2. Bob takes the two passwords, concatenates them, and computes a hash. Now Bob has
$$\text{hash}(P_0^{\text{left}}, P_1^{\text{right}}).$$

3. Bob finds the row of the table indexed by $\text{hash}(P_0^{\text{left}}, P_1^{\text{right}})$, and he uses it to look up
$$\text{Enc}_{P_0^{\text{left}}, P_1^{\text{right}}}(P_0^{\text{out}}).$$

4. Bob uses the concatenation of the two passwords $P_0^{\text{left}}, P_1^{\text{right}}$ to decrypt P_0^{out}.

5. Now Bob has the password for the bit 0 to feed into the next gate – but he does not know his bit is 0.

So Bob is exactly where he started: he knows the password for his bit, but he does not 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 us frame the problem for just 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 will see in (Section 2.2).

2.2 Oblivious transfer

Alice has n messages $x_1, \ldots, x_n$. We will assume the messages are essentially unrelated to each other (since we could always pad them with random bits). 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 will 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" is not any harder, so we will do 1-of-n.)

2.2.1 Commutative encryption

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

$$\mathrm{Dec}_b(\mathrm{Dec}_a(\mathrm{Enc}_b(\mathrm{Enc}_a(x)))) = x.$$

In other words, if Alice encrypts a message, and Bob applies a second layer of encryption to the encrypted message, it does not 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 is locked with two padlocks. Alice puts a message inside the box, locks it with her lock, and ships it to Bob. Bob puts his own lock on the box and ships it back to Alice. What is special about commutative encryption is that Bob's lock does not 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 $\mathbb{F}_p^\times$, 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.
4. Bob takes a 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 $x_1, \ldots, x_n$, 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:

$$\mathrm{Enc}_a(x_1), \ldots, \mathrm{Enc}_a(x_n).$$

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

At this point, Bob cannot read any of the messages, because he does not know the keys. No problem! Bob just picks out the i-th ciphertext $\mathrm{Enc}_a(x_i)$, adds his own layer of encryption onto it, and sends the resulting doubly-encrypted message back to Alice:

$$\mathrm{Enc}_b(\mathrm{Enc}_a(x_i)).$$

Alice does not know Bob's key b, so she cannot learn anything about the message he encrypted – even though it originally came from her. Nonetheless she can apply her own decryption method Dec_a to it. Since the encryption scheme is commutative, the result of Alice's decryption is simply

$$\mathrm{Enc}_b(x_i),$$

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: In total, Alice sent two messages to Bob (Step 1 and 3), and Bob sent one message to Alice (Step 2).

We can do better, using public-key cryptography.

Let us 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:

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

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

Is Bob happy with this protocol? Yes. Alice has no way of learning the value of i, as long as she cannot 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 cannot tell random garbage from a true public key! And then Bob would be able to decrypt all n messages $x_1, \ldots, x_n$.

But there is a simple trick to fix it. Bob chooses some "verifiably random" value r; to fix ideas, we could agree to use $r = \text{sha}(1)$. Then we require that the numbers $b_1, \ldots, b_n$ 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 $b_1, \ldots, b_n$ 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 is a standard assumption in public-key cryptography that Bob cannot do this: There is 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 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 the discrete logarithm of the public key.) 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 is 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 is *garbled gates*, which are gates whose functionality is hidden. This can be done 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, it is also possible for a group of people to compute whatever secret function they want, which is the field of *multi-party computation (MPC)*.

3 SNARKs Prelude: Elliptic Curves and Polynomial Commitments

Evan Chen

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.[5]
- 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,[6] which is secure because it is hard to factor a composite number (like 6887) into prime factors ($6887 = 71 \cdot 97$).

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

[5] https://en.wikipedia.org/wiki/EdDSA
[6] https://en.wikipedia.org/wiki/RSA_(cryptosystem)

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[7] 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 (Jonathan Wang has a blog post[8] about the properties of this curve). The name BN stands for Barreto-Naehrig, two mathematicians who proposed a family of such curves in 2006.[9]

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 is a white lie: Conventionally, there is one additional point $O = (0, \infty)$ called the "point at infinity" added in (whose purpose we describe in the next section).

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

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*[10]). 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 is possible to define an **addition** operation on the curve; this is called the group law on the elliptic curve.[11] This

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

$$\begin{aligned} x :=\ & 4965661367192848881 \\ p :=\ & 36x^4 + 36x^3 + 24x^2 + 6x + 1 \\ =\ & 21888242871839275222246405745257275088 \\ & 696311157297823662689037894645226208583 \\ q :=\ & 36x^4 + 36x^3 + 18x^2 + 6x + 1 \\ =\ & 21888242871839275222246405745257275088 \\ & 548364400416034343698204186575808495617. \end{aligned}$$

[8] https://hackmd.io/@jpw/bn254

[9] https://link.springer.com/content/pdf/10.1007/11693383_22.pdf

[10] https://en.wikipedia.org/wiki/The_Hunting_of_the_Snark

[11] https://en.wikipedia.org/wiki/Elliptic_curve#The_group_law

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*.

This group law involves some heavy algebra. It is 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 for the interested reader.

> So, let us 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 are 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. We will 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) three roots. In fact if it has two roots, it is guaranteed to have a third (because you can factor out the first two roots, and then you are left with a linear factor).
>
> Now 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 are solving
>
> $$Y^2 = x_R^3 + 3,$$

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

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 will not 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 $-P = (x_P, -y_P)$.
- 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
$$P + Q = -R.$$

We 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 will not 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 will 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 will be working with points on E as black boxes. We will be able to add them, subtract them, and multiply them by arbitrary scalars from $\mathbb{F}_q$.

Consequently — and this is important — **one should 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 "scalar" 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 does not 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 $g, g' \in E$, the* discrete logarithm problem *asks you to find an integer n such that $n \cdot g = g'$.*

Experience suggests that the discrete logarithm problem is hard: In general, we do not know a fast algorithm to solve it. The discrete logarithm assumption *says that no such algorithm exists.*

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 uses 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.[12] For example, to compute $400g$, one only

[12] https://en.wikipedia.org/wiki/Exponentiation_by_squaring

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 nondegeneracy constraint $4a^3 + 27b^2 \not\equiv 0 \bmod p$ holds. In such a situation, $E(\mathbb{F}_p)$ will indeed be an abelian group once the identity element $O = (0, \infty)$ is added in.

How large is $E(\mathbb{F}_p)$? There is a theorem called Hasse's theorem[13] 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[14] 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 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 on the curve and multiply it by 8.

BN254 is also engineered to have a property called *pairing-friendliness*, which is defined in Section 3.3.2 when we need it later. (In contrast, Curve25519 does not have this property.)

[13] https://en.wikipedia.org/wiki/Hasse's_theorem_on_elliptic_curves
[14] https://en.wikipedia.org/wiki/Curve25519

3.2.2 Example application: EdDSA signature scheme

We will show how Assumption 3.6 can be used to construct a signature scheme that replaces RSA. This scheme is called EdDSA,[15] and it is 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 the 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 msg, given her published public key $[d]$.

Algorithm 3.10 (EdDSA signature generation)**.** Suppose Alice wants to sign a message msg.

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

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

[15] https://en.wikipedia.org/wiki/EdDSA

Algorithm 3.11 (EdDSA signature verification)**.** For Bob to verify a signature $([r], s)$ for msg:

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

An adversary cannot forge the signature even if they know r and n. Indeed, such an adversary can compute what the point $[s] = [r] + n[d]$ should be, but without knowledge of d they cannot get the integer s, due to Assumption 3.6.

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 is therefore imperative that r is not known to Bob nor reused between signatures, and so on. One way to do this would be to pick $r = \text{hash}(d, \text{msg})$; this has the bonus that it is 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" $\text{Com}(x)$ 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 $\text{Com}(x)$). 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 $g_1, \dots, g_n \in E$ are a bunch of randomly chosen points of E with order q, then it is computationally infeasible to find $(a_1, \dots, a_n) \neq (b_1, \dots, b_n) \in \mathbb{F}_q^n$ such that

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

(Remember that $q \approx 2^{256}$ is very large.)

Definition 3.12. In these notes, if there is a globally known elliptic curve E and points $g_1, \dots, g_n$ have order q and no known nontrivial linear dependencies between them, we will say they are a *computational basis over* $\mathbb{F}_q$.

Remark 3.13. This may horrify pure mathematicians because we are pretending the map

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

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 cannot 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 cannot find a collision"). To spell this out:

Definition 3.14. Let $g_1, \ldots, g_n \in E$ be a computational basis over $\mathbb{F}_q$. Given a vector

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

of scalars, the group element

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

is called the *Pedersen commitment* to 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 will not 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 cannot 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 this modest book. Fortunately, we will not need the details of how it works; but we will 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 will see, the pairing is only used by the verifier Victor, never by the prover Peggy.

Remark 3.18. On the other hand, we currently do not seem to know a good way to do *multilinear* pairings. For example, we do not 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 ζ is some fixed q-th root of unity. The construction of the pairing is based on the Weil pairing[16] in algebraic geometry.

[16] https://en.wikipedia.org/wiki/Weil_pairing

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.[17])

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, no matter how many coefficients the polynomial has.

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 publish

$$[s^0], [s^1], \ldots, [s^M]$$

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.

[17] https://crypto.stanford.edu/pbc/notes/ep/pairing.html

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 $[s^0] = [1], [s^1], \ldots, [s^M]$. 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 the value of 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 $[s_1^0], \ldots, [s_1^M]$, and then discards s_1. The second party chooses s_2 and computes $[(s_1 s_2)^0], \ldots, [(s_1 s_2)^M]$. And so forth. In the end, the value of s with be the product of the secrets s_i chosen by the parties... so the only way they can break secrecy is if all the "trusted parties" collude.

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
$$\text{pair}([Q(s)], [s] - [z]) = \text{pair}([P(s)] - [y], [1]) \tag{2}$$
and accepts if and only if Equation (2) is true.

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

If $y \neq P(z)$, then Peggy cannot 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 is not 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 is 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 $P(1) = 100$, $P(2) = 400$, $\dots$, $P(9) = 8100$. (We chose these numbers so that $P(X) = 100X^2$ for $X = 1, \dots, 9$.)

Evaluating a polynomial at $1, 2, \dots, 9$ is essentially the same as dividing by $(X-1)(X-2)\cdots(X-9)$ and taking the remainder. In other words, if Peggy does a polynomial long division, she will find that

$$P(X) = Q(X)((X-1)(X-2)\dots(X-9)) + 100X^{2.}$$

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

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

The full generality just replaces the $100X^2$ with the polynomial obtained from Lagrange interpolation[18] (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 $1 \le i \le n$.

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
$$P(X) - f(X) = (X - z_1)(X - z_2)\cdots(X - z_n) \cdot Q(X).$$
3. Peggy sends the single element $[Q(s)]$ as her proof.
4. Victor verifies
$$\text{pair}([Q(s)], [(s - z_1)(s - z_2)\dots(s - z_n)]) = \text{pair}([P(s)] - [f(s)], [1]).$$

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

3.4.4 Root check

To make PLONK work, we are 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 is the problem statement:

[18] https://en.wikipedia.org/wiki/Lagrange_polynomial

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 $P_1(z) = P_2(z)$ holds for all z in some large finite set S.*

Peggy just needs to show that $P_1 - P_2$ is divisible by $Z(X) := \prod_{z \in S}(X - z)$. This can be done by committing the quotient

$$H(X) := \frac{P_1(X) - P_2(X)}{Z(X)}.$$

Victor then gives a random challenge $\lambda \in \mathbb{F}_q$, and then Peggy opens $\mathrm{Com}(P_1)$, $\mathrm{Com}(P_2)$, and $\mathrm{Com}(H)$ at λ.

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

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

she could define

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

and run the same protocol with this F. This means she does not 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. If she has not already done so, Peggy sends to Victor a commitment $\mathrm{Com}(F)$ to F.[19]
2. Both parties compute the polynomial
$$Z(X) := \prod_{z \in S}(X - z) \in \mathbb{F}_q[X].$$
3. Peggy does polynomial long division to compute $H(X) = \frac{F(X)}{Z(X)}$.
4. Peggy sends $\mathrm{Com}(H)$.
5. Victor picks a random challenge $\lambda \in \mathbb{F}_q$ and asks Peggy to open $\mathrm{Com}(H)$ at λ, as well as the value of F at λ.
6. Victor verifies $F(\lambda) = Z(\lambda)H(\lambda)$.

[19] In fact, it is enough for Peggy to have some way to prove to Victor the values of F.

So for example, if F is a product of two polynomials $F = F_1F_2$, and Peggy has already sent commitments to F_1 and F_2, then there is no need for Peggy to commit to F.

Instead, in Step 5 below, Peggy opens $\mathrm{Com}(F_1)$ and $\mathrm{Com}(F_2)$ at λ, and that proves to Victor the value of $F(\lambda) = F_1(\lambda)F_2(\lambda)$.

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, the discrete logarithm assumption 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

Evan Chen

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 is just a really long, annoying calculation, and Victor does not 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 be 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.)

The name stands for:

- *Succinct*: the proof length is short (in fact, it is a constant length, independent of how long the problem is).
- *Non-interactive*: the protocol does not require back-and-forth communication.
- *Argument*: basically a proof. There is a technical difference, but we will not worry about it.
- *of Knowledge*: the proof does not just show the system of equations has a solution; it also shows that the prover knows one.

We will not 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 are 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 $\text{sha}(M) = 0xa91af3ac...$, but you do not 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.[20] 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 proofs for arbitrary functions. That means we need a "programming language" that we will write our function in. For PLONK, the choice that is 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 quadratic equations.

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

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

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

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

This leads to the natural question of how a function like SHA-256 can be encoded into a system of quadratic equations. This process of encoding a problem into algebra is called *arithmetization*. It turns out that quadratic equations over $\mathbb{F}_q$, viewed as an NP problem called Quad-SAT, is *NP-complete*; in other words, any NP problem can be rewritten as a system of quadratic equations. If you are not familiar with this concept, the upshot is that Quad-SAT being NP-complete means it can serve as a reasonable arithmetization that can express most reasonable (NP) problems.

[20] https://tlsnotary.org

Remark 4.2 (Example of Quad-SAT encoding 3-SAT)**.** We assume knowledge of 3-SAT and it being NP-complete. The following example instance illustrates how to convert any instance of 3-SAT into a Quad-SAT problem:

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

(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 codeable. This is the goal of projects like Circom[21], which gives a high-level language that compiles a function like SHA-256 into a system of equations over $\mathbb{F}_q$ that can be used in practice. Systems like this are called *arithmetic circuits*, and Circom is appropriately short for "circuit compiler". If you are curious, you can see how SHA-256 is implemented in Circom on GitHub.[22]

So, the first step in proving a claim like "I have a message M such that $\text{sha}(M) = 0xa91af3ac...$" 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 will not get into any more details of arithmetization here.

4.2.2 An instance of PLONK

PLONK is 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*.

[21] https://docs.circom.io/

[22] https://github.com/iden3/circomlib/blob/master/circuits/sha256/sha256.circom

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 $i = 1, \dots, n$, 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, 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

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

we get a "multiplication" gate $a_ib_i = c_i$. Finally, if κ is any constant, then

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

gives the constraint $a_i = \kappa$.

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

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

We will 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 will 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 us now explain how each step works.

4.2.3 Step 1: the commitment

In PLONK, we will assume that $q \equiv 1 (\mathrm{mod}\, n)$, 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

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

We specifically choose ω^i because that way, if we use Algorithm 3.26 on the set $\{\omega, \omega^2, \ldots, \omega^n\}$, then the polynomial called Z is just

$$Z(X) = (X - \omega) \cdots (X - \omega^n) = X^n - 1,$$

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 are working in a finite field, the fast Fourier transform is sometimes called the "number theoretic transform" (NTT) even though it is 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

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

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

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 indeed divisible by $Z(X) := X^n - 1$).

4.2.5 Step 3: proving the copy constraints

The copy constraints are the trickiest step. There are a few moving parts to this idea, so we skip it for now and dedicate Section 4.3 to it.

4.2.6 Step 4: public and private witnesses

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 us 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 will just do the same thing for them.) The prover can interpolate another polynomial, A^{public}, such that $A^{\text{public}}(\omega^i) = a_i$ if $i \in S$, and $A^{\text{public}}(\omega^i) = 0$ if $i \notin S$. Actually, both the prover and the verifier can compute A^{public}, since all the values a_i are publicly known!

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

4.3 Copy constraints in PLONK

Now we elaborate on Step 3 which we deferred back in Section 4.2.5. As an example, the constraints might be:

$$a_1 = a_4 = c_4 \quad \text{and} \quad b_2 = c_1.$$

Before we show how to check this, we provide a solution to a "simpler" problem called "permutation check". Then we explain how to deal with the full copy check.

4.3.1 Easier case: permutation check

Problem 4.7. *Suppose we have polynomials $P, Q \in \mathbb{F}_q[X]$ which encode two vectors of values*

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

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

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

to be true, in the sense that both sides are the same polynomial in a single formal variable T. And for that, it is sufficient that a single random challenge $T = \lambda$ passes Equation (5): If the two sides of Equation (5) are not the same polynomial, then the two sides can have at most $n - 1$ common values. So for a randomly chosen λ (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 4.8 (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 $X = 1$.

4.3.2 Copy check

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

$$(\boldsymbol{a_1}) = (\boldsymbol{a_4}) = (\boldsymbol{c_4}) \quad \text{and} \quad (\boldsymbol{b_2}) = (\boldsymbol{c_1}).$$

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

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

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

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

Now how can the prover establish Equation (7) succinctly? The answer is to run a permutation check on the $3n$ entries of Equation (7)! 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 do not satisfy all the copy constraints, then a randomly selected μ 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 (8)

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

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:

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

Then the prover can start defining accumulator polynomials, after re-introducing the random challenge λ from permutation check. We are going to need six in all, three for each side of Equation (8): We call them F_a, F_b, F_c, F'_a, F'_b, F'_c. The ones on the left-hand side are interpolated so that

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

while the ones on the right have the extra permutation polynomials

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

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

initialization conditions

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

and six accumulation conditions

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

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 4.9 (Copy check)**.**

0. Peggy has already sent the three commitments $\mathrm{Com}(A), \mathrm{Com}(B), \mathrm{Com}(C)$ 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 $F_a, \ldots, F'_c$ 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 $X \in \{\omega, \omega^2, \ldots, \omega^{n-1}\}$.
6. Peggy uses Algorithm 3.26 to prove Equation (13) holds.

4.4 Making it non-interactive: Fiat-Shamir

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

Fiat-Shamir turns it into the following noninteractive protocol.

Algorithm 4.11. 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 λ.

4. Victor verifies that $\lambda = \text{hash}(\text{Com}(F), \text{Com}(H))$ 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.

4.5 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 $\mathbb{F}_q$, meaning that what PLONK does under the hood is prove that a system of these equations are satisfied.

3. The work under the hood of PLONK comes down to polynomial commitments (specifically KZG). KZG allows PLONK's gate checks and copy checks.

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

Brian Lawrence and Yan X Zhang

5.1 FHE and leveled 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 does not want to tell him x. What Alice wants is a *fully homomorphic encryption (FHE)* protocol, meaning:

1. Alice encrypts x and sends Bob $\mathrm{Enc}(x)$.
2. Bob then "applies f to the ciphertext" and obtains $\mathrm{Enc}\,(f(x))$, sending it to Alice.
3. Alice decrypts $\mathrm{Enc}\,(f(x))$ to learn $f(x)$.

Leveled FHE is a weaker version of FHE. Like FHE, leveled 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.

Why is there a limit? 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 is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10), and your "encryption scheme" encrypts the message as a real number that is very close to the message, and decrypts a real number by "round to the nearest integer." So, the message 2 might be encrypted as 1.999832.

(You might be thinking: This is some pretty terrible cryptography, because the message is not secure. Anyone can figure out how to round a number, no secret key required. Yep, you are right. The full scheme (Section 5.3) is more complicated. But it still has this "rounding-off-errors" feature, and that is what we want to focus on right now.)

Now imagine that the main operation you want to perform is addition (optionally, say, modulo 11). 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 will not give the

right answer anymore. But as long as you are careful not to go over the error limit, you can add ciphertexts with confidence.

For our leveled 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 will 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; it would make more sense to call it "approximate linear algebra modulo q." Anyway, we will 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 leveled FHE.

5.2 A hard problem: learning with errors

As we have 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 assumption.

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

5.2.1 A small example of an LWE problem

Here is a concrete example of an 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 $a = (a_1, \ldots, a_4)$. There are two sets of claims. Each claim "$(x_1, \ldots, x_4) : y$" purports the relationship*

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

(The ε 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 $(a_1, \ldots, a_4)$.

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

(**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 $S \subset \mathbb{F}_{11}$, 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 $\alpha S = \{\alpha s|s \in S\}$ and $S + T = \{s + t|s \in S, t \in T\}$.
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 exact 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 one or two 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 $4a_1 + 7a_2 \in \{7, 8\}$. Also, 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 $(6, 0, 0, 0|6|\{0, -1, -2\})$, 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.

5.2.2 General problem

The LWE problem (Problem 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

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

given a bunch of **approximate** equations of the form

$$y = a_1x_1 + \cdots + a_nx_n + \epsilon,$$

where each ϵ is a "small" error (for simplicity, say in $\{0, 1\}$).

5.3 Public-key cryptography from LWE

In Section 5.2.1 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.

As an example of how LWE can be used, let us see how to turn LWE into a public-key cryptosystem. We will use the same numbers from the "blue set" in Section 5.2.1. In fact, that "blue set" will be exactly the public key.

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

The private key is simply the vector a.

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

Since the LWE problem is hard, we can release the public key to everybody, and they will not be able to determine the private key.

5.3.1 Encryption

Suppose you have a message $m \in \{0, 5\}$. (You will see in a moment why we insist that m 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 $\mathbf{x} \cdot \mathbf{a} + \epsilon = y + m$, where ϵ is "small".

How to do the encryption? If you are 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 would not be very secure: If I see your ciphertext, I can find that $\mathbf{x}$ in the table and use it to decrypt m.

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 are guaranteed that each row of the table has ϵ either 0 or 1. So if you add at most four rows, then the total ϵ will be at most 4. Since m is either 0 or 5 (and we are working modulo $q = 11$), that is just enough to determine m uniquely.

So, here is the method. You choose at random four (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 $y = y_0 - m \pmod{q = 11}$ of course), and send the message $(\mathbf{x} : y)$.

5.3.2 An example

Let us say you randomly choose the four rows:

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

Now you add them up to get the following.

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

(For reference, the actual value is 4, so our accumulated error is 2.)

Finally, let us 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

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

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

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

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

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 are going to want to take q pretty big, so you should imagine that $q \approx 2^{\sqrt{n}}$.

For security, instead of adding 4 rows of the public key, we want to add at least $\log(q^n) = n \log q$ rows. To be safe, maybe a little bigger, say $N = 2n \log q$ (of course, for this to work, the public key has to have at least N rows). The encryption algorithm will be "select some subset of the rows at random, and add them up".

Combining N rows will have the effect of multiplying the error by N, so if the initial ϵ was bounded by 1, then the error in the ciphertext will be at most N. But remember that q is exponentially large compared to N and n anyway, so a mere factor of N should not scare us!

To generalize our choice of m in $\{0, 5\}$, we could encode a single bit by using either 0 or $\lfloor \frac{q}{2} \rfloor$ to obtain maximum separation and thus tolerance to error. Alternatively, we could allow the message to be any multiple of some constant r, where r is bigger than the error bound, which allows you to encode a message space of size q/r rather than just a single bit.

When we do FHE, we are going to apply many operations to a ciphertext, and each is going to cause the error to grow. We are 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 Leveled FHE from LWE

5.4.1 The main idea: approximate eigenvalues

Now we want to turn the public-key encryption from Section 5.3 into a leveled 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 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 is easier to do algebra with $+$ and $\times$.

Recall the setup from Section 5.3: We are 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 will do linear algebra, where all our calculations are done modulo q – but we will also allow the calculations to have a small "error" ϵ, which will typically be much, much smaller than q.

As before, our *secret key* will be a vector of length n:

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

Suppose we want to encode a message μ that is just a single bit, let us 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.

Remark 5.2. With a bit of effort, it is possible to make this into a public-key cryptosystem too. 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 μ 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. This gives a C such that

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

Problem 5.3. *How do we build such a C_0? (One possible direction is to build it row-by-row.)*

5.4.2 Operations on encrypted data

To make homomorphic encryption work, we need to explain how to operate on μ. We will 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 is not a great operation, because if you add $1 + 1$, you get 2, and 2 is not a legitimate bit anymore. So we will not really use this.

Negation of a bit (NOT) is equally simple. If $\mu \in \{0, 1\}$ is a bit, then its negation is simply $1 - \mu$. And if C is a ciphertext for μ, 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 is just AND. To multiply two bits, you just multiply (matrix multiplication) the ciphertexts:

$$C_1C_2\mathbf{v} \approx C_1(\mu_2\mathbf{v}) = \mu_2C_1\mathbf{v} \approx \mu_2\mu_1\mathbf{v} = \mu_1\mu_2\mathbf{v}.$$

At this point you might be concerned about this symbol $\approx$ and what happens to the size of the error. That is an important issue, and we will resolve it with the help of a special operation called "Flatten."

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.

5.4.3 The "Flatten" operation

In order to make the error estimates work out, we are 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

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

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 do not know the specific value of a_1.

Now suppose we give you the vector

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

We 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.

Problem 5.4. *How would you flatten a different vector, like*

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

As a hint, remember we are working with numbers modulo 11: So if you come across a number that is bigger than 11 in your calculation, it is safe to reduce it mod 11.

In general, if you know that $\mathbf{v}$ 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 Flatten(C) such that:

- Flatten$(C)\mathbf{v} = C\mathbf{v}$, and
- All the entries of 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 μ_1 and μ_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.

5.4.4 Error analysis

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

$$C_1\mathbf{v} = \mu_1\mathbf{v} + \varepsilon_1,$$

where ε_1 is some vector with all its entries bounded by some B. (And similarly for C_2 and μ_2.)

When we add two ciphertexts, the errors add:

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

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

Negation is similar to addition – in fact, the error will not change at all.

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

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

Now since μ_2 is either 0 or 1, we know that $\mu_2\varepsilon_1$ is a vector with all entries bounded by B. What about $C_1\varepsilon_2$? Here we have to think carefully 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 are assuming that C_1 is a 0-1 matrix, and all entries of ε_2 are bounded by $B\ldots$ so the product has all entries bounded by nB. Adding this to the error for $\mu_2\varepsilon_1$, 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 cannot 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 is 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 *leveled* FHE 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 large circuits, it is the only viable option. Bootstrapping is beyond the scope of this book.

5.5 FHE takeaways

1. A *fully homomorphic encryption* protocol allows Bob to compute some function $f(x)$ for Alice in a way that Bob does not get to know x or $f(x)$.

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 $\mathbb{F}_q$ is 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.

6 Oblivious RAM

Elaine Shi

To motivate Oblivious RAM, let us think of Signal's usage scenario. Signal is an encrypted messenger app with billions of users. They want to support a private contact discovery application. In contact discovery, a user Alice sends her address book to Signal, and Signal will look up its user database and return Alice the information about her friends. The problem is that many users want to keep their address book private, and Signal wants to provide contact discovery without learning the users' contacts.

A naive solution is to rely on trusted hardware. Suppose Signal has a secure processor (like Intel SGX) on its server. One can think of the secure processor as providing a hardware sandbox (often referred to as an *enclave*). Now, Alice sends her address book in encrypted format to the server's enclave; further, the server's database is also encrypted as it is stored in memory and on disk. Now, the enclave has a secret key that can decrypt the data and perform computation inside. At first sight, this seems to solve the privacy problem, since data is always encrypted in transit and at rest, and the server cannot see the contents.

Unfortunately, encryption alone provides little privacy in such scenarios. The enclave will need to access encrypted entries stored on disk, and the server's operating system can easily observe the *access patterns*, i.e., which memory pages are being fetched by the enclave. The access patterns leak exactly who Alice's friends are even if the data is encrypted!

In general, access patterns of a program leak sensitive information about your private data. As a simpler example, if you are performing binary search over a sorted array, the entries accessed during the search would leak your private query.

We can also think of access pattern leakage through a programming language perspective. For example, the following program has an `if`-branch dependent on secret inputs. (Maybe the secret input is the last bit of a secret key). By observing whether memory location `x` or `y` is accessed, one can infer which branch is taken.

```
if (s) {
  mem[x]
} else {
  mem[y]
}
```

Therefore, we want to solve the following challenge:

How can we provably hide access patterns while preserving efficiency?

The solution Signal eventually deployed is an algorithmic technique called Oblivious RAM (ORAM).

6.1 Oblivious RAM: problem definitions

Oblivious RAM (ORAM) is a powerful cryptographic protocol that *provably* hides access patterns to sensitive data.

We would like to ensure a very strong notion of security. In particular, no information should be leaked about: 1) which data block is being accessed; 2) the age of the data block (when it was last accessed); 3) whether a single block is being requested repeatedly (frequency); 4) whether data blocks are often being accessed together (co-occurrence); or 5) whether each access is a read or a write.

Let us explain the parts of an ORAM system.

An ORAM algorithm (the *client*) sits between a *user* who wants to access memory and a *server* that has memory capabilities. At the server-ORAM interface, the server simply acts as a memory: The ORAM client sends read and write requests to the server, and the server responds. Between the ORAM and the user, the user submits *logical* read and write requests to the ORAM client, and the client will reply to each (after interaction with the server).

We will call *physical* memory the memory that the server manages, and *logical* memory the memory the user wants to access. Memory, both physical and logical, will be made up of "blocks"; our ORAM algorithm will support N blocks of logical memory.

Formally, the user sends to the algorithm a sequence of *logical* requests, where each logical request is of the form $(\texttt{read}, \texttt{addr})$ or $(\texttt{write}, \texttt{addr}, \texttt{data})$.

After each user request, the ORAM algorithm interacts with the server to make a sequence of *physical* memory accesses, where each physical access either reads or writes a block to a physical location.

And finally, the ORAM algorithm turns back to the user and returns an answer to the logical request.

For example, in Signal's scenario, the "user" and the "ORAM client" are both in the hardware enclave, and the "ORAM server" is the untrusted memory and disk on Signal's server.

The security requirement for ORAM is that the server should learn nothing about the user's logical memory requests from observing the sequence of physical memory requests. For any two *logical* request sequences, the ORAM's resulting *physical* access sequences will be indistinguishable.

Remark 6.1. In this security requirement, we require that the server learn nothing from observing only the list of physical addresses, and whether each physical access is a read or write. We do not say anything about the data that is written to physical memory.

In practice, we need to use encryption to hide the contents of the blocks. If we read a block and then write it back, we should re-encrypt it with a different ciphertext, or else the server would recognize it as the same block.

From now on we will simply assume secure encryption as given, and focus on hiding the access patterns.

6.2 Naive solutions

6.2.1 Naive solution 1

One trivial solution is for the client to read all blocks from the server upon every logical request. Obviously this scheme leaks nothing but would be prohibitively expensive.

6.2.2 Naive solution 2

Another trivial solution is for the client to store all blocks, and thus the client need not access the server to answer any memory request. But this defeats the numerous advantages of cloud outsourcing in the first place. *Henceforth, we require that client store only a small amount of blocks* (maybe constant or polylogarithmic in N).

6.2.3 Naive solution 3

Another naive idea is to randomly permute all memory blocks through a secret permutation known only to the client. Whenever the client wishes to access a block, it will appear to the server to reside at a random location.

Indeed, this scheme gives a secure *one-time* ORAM scheme: It provides security if every block is accessed only once. However, if the client needs to access each block multiple times, then the access patterns will leak statistical information such as frequency (how often the same block is accessed) and co-occurrence (how likely two blocks are accessed together). As mentioned earlier, one can leverage[23] such statistical information to infer sensitive secrets.

6.2.4 Important observation

The above naive solution 3 gives us the following useful insight: Informally, if we want a "non-trivial" ORAM scheme, it appears that we may have to relocate a block after it is accessed — otherwise, if the next access to the same block goes back to the same location, we can thus leak statistical information. It helps to keep this observation in mind when we describe our ORAM scheme later.

6.3 Binary-tree ORAM: data structure

We will learn about tree-based ORAMs. Then, we will mention an improvement called Path ORAM,[24] which is the scheme that Signal has deployed.

[23] https://www.ndss-symposium.org/wp-content/uploads/2017/09/06_1.pdf
[24] https://eprint.iacr.org/2013/280.pdf

6.3.1 Server data structure

The server stores a binary tree with N leaves. (If necessary, replace N with the next larger power of 2.)

Each node is called a *bucket*, and each bucket is a finite array that can hold up to Z number of blocks — for now, think of Z as being relatively small (maybe polylogarithmic in N); we will describe how to parametrize Z later. Some of the blocks stored by the server are *real*; other blocks are *dummy*. As will be clear later, these dummy blocks are introduced for security: We do not want the server to learn which buckets hold real blocks.

6.3.2 Main path invariant

The most important invariant is that at any point of time, each block is mapped to a random path in the tree (also referred to as the block's designated path), where a path begins from the root and ends at some leaf node — and thus a path can be specified by the corresponding leaf node's identifier. When a block is mapped to a path, it means that the block can legitimately reside anywhere along the path.

6.3.3 Imaginary position map

For the time being, we will rely on the following cheat (an assumption that we can get rid of later). We assume that the client can store a somewhat large position map that records the designated path of every block. In general, such a position map would require roughly $\Theta(N \log N)$ bits to store — but later we can recursively outsource the storage of the position map to the server by placing position maps in progressively smaller ORAMs.

6.4 Binary-tree ORAM: operations

We now describe how to access blocks in our ORAM scheme.

6.4.1 Fetching a block

Given how our data structures are set up, accessing a block is very easy: the client simply looks up its local position map, finds out on which path the block is residing, and then reads each and every block on the path. As long as the main invariant is respected, the client is guaranteed to find the desired block.

6.4.2 Remapping a block

Recall that earlier, we have gained the informal insight that whenever a block is accessed, it should relocate. Here, whenever we access a block, we must remap it to a randomly chosen new path — otherwise, we would end up going back to the same path if the block is requested again, thus leaking statistical information.

To remap the block, we choose a fresh new path, and we update the client's position map to associate the new path with the block. We now would like to write this block back to the tree, to somewhere on the new path (and if the request is a `write` request, the block's contents are updated before being written back to the server). But doing this is tricky! It turns out that we cannot write the block back directly to the leaf bucket of the new path, since doing so would reveal which new path the block got assigned to. For the same reason, we cannot write this block back to any internal nodes of the new path either, since writing to any internal node on the new path also leaks partial information about the new path.

It turns out that the only safe location to write the block back is to the root bucket! The root bucket resides on every path, and thus writing the block back to the root does not violate the main path invariant; and further, it does not leak any information about the new path.

Now this is great. Our idea thus is to write this block back to the root bucket. However, there is also an obvious problem. The root bucket has a capacity of Z, and if we keep writing blocks back to the root, soon enough the root bucket will overflow! Therefore, we now introduce a new procedure called *eviction* to cope with this problem.

6.4.3 Eviction

Eviction is a maintenance operation performed upon every data access to ensure that none of the buckets in the ORAM tree will ever overflow except with negligible in N failure probability. Note that if an overflow does happen, the block that leads to the overflow can get lost since there is no space to hold it on the server, and this can affect the correctness of our ORAM scheme. However, we will guarantee that such correctness failure happens only with negligible probability.

The high-level idea is very simple: Whenever we can, we will try to move blocks in the tree closer to the leaves, to allow space to free up in smaller levels of the tree (levels closer to the root). There are a few important considerations when performing such eviction:

- Data movement during eviction must respect the main path invariant: Each block can only be moved into buckets in which it can legitimately reside.
- Data movement during eviction must retain *obliviousness*: The physical locations accessed during eviction should be independent of the input requests to the ORAM.
- As we perform eviction, we pay a cost for this maintenance operation and the cost is charged to each data access. Obviously, if we are willing to pay more such cost, we can pack blocks closer to the leaves, thus leaving more room in smaller levels. In this way, overflows are less likely to happen.

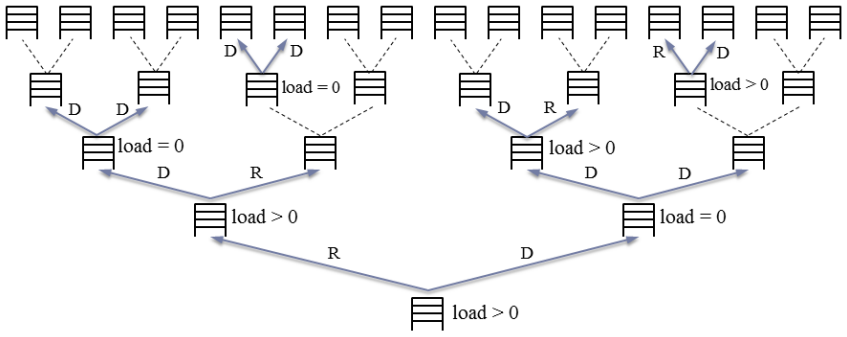

Figure 1: The `Evict` algorithm. Upon every data access operation, two buckets are chosen at every level of the tree for eviction during which one data block will be evicted to one of its children. To ensure security, a dummy eviction is performed for the child that does not receive a block; further, if the bucket chosen for eviction is empty, dummy evictions are performed on both child buckets. In this figure, R denotes a real eviction and D denotes a dummy eviction.

> On the other hand, we also do not want the eviction cost to be too expensive. Therefore, another tricky issue is how we can design an eviction algorithm that achieves the best of both worlds: With a small number of eviction operations, we want to keep the probability of overflow very small (technically: negligible in N).

We describe a simple candidate eviction scheme, and we will give an informal analysis of the scheme later:

- *[An eviction algorithm]* Upon every data access, we choose at random 2 buckets in every level of the tree for eviction (for the root level, pick one bucket). If a bucket is chosen for eviction, we will pop an arbitrary block (if one exists) from the bucket, and write the block to one of its children.

Note that depending on the chosen block's designated path, there is only one child where the block can legitimately go. We must take precautions to hide where this block is going: Thus for the remaining child that does not receive a block, we can perform a "dummy" eviction. Additionally, if the bucket chosen for eviction is empty (does not contain any real blocks), then we make a dummy eviction for both children — this way we avoid leaking the information that the chosen bucket is empty.

More specifically, to write an intended block to a child bucket, we sequentially scan through the child bucket. If the slot is occupied with a real block, we simply write the block back. If the slot is empty, we write the intended block into that slot. A dummy eviction therefore is basically reading every block sequentially and writing the original contents back.

So far, we have not argued why any bucket that receives a block always has space for this block — we will give an informal analysis later to show that this is indeed the case.

6.4.4 Algorithm pseudo-code

We present the algorithm's pseudo-code in Algorithms 6.2 and 6.3.

Algorithm 6.2. `The procedure Access`$(\texttt{op}, \texttt{addr}, \texttt{data}^*)$ where $\texttt{op} = \texttt{read}$ or $\texttt{op} = \texttt{write}$

Assume: Each block is of the form $(\texttt{addr}, \texttt{data}, l)$ where l denotes the block's current designated path.

1: Let l^* be a random value from 1 to N. Assign $l \leftarrow \texttt{position}[\texttt{addr}]$, $\texttt{position}[\texttt{addr}] \leftarrow l^*$.
2: **for** each bucket from leaf l to root **do**
3: Scan bucket, and if $(\texttt{addr}, \texttt{data}_0, \_) \in$ bucket then let $\texttt{data}^* \leftarrow \texttt{data}_0$ and remove this block from bucket.
4: **end for**
5: if $\texttt{op} = \texttt{read}$ then add $(\texttt{addr}, \texttt{data}^*, l^*)$ to the root bucket; else add $(\texttt{addr}, \texttt{data}, l^*)$ to the root bucket.
6: Call the `Evict` subroutine.
7: **return** $\texttt{data}^*$.

Algorithm 6.3. The procedure `Evict`

1: **for** each level d from root to the level of leaves -1 **do**
2: bucket_0, $\text{bucket}_1 \leftarrow$ randomly choose 2 distinct buckets in the level d (for the root level, pick one bucket).
3: **for** bucket $\in \{\text{bucket}_0, \text{bucket}_1\}$ **do**
4: block := pop a real block from bucket if one exists; else block := $(\perp, \perp, \perp)$.
5: for each of the two children of bucket in a fixed order: scan the child bucket reading and writing every block. If block is real and wants to go to the child, write block to an empty slot in the child bucket.
6: **end for**
7: **end for**

Remark 6.4. Note that in a full-fledged implementation, all blocks are typically encrypted to hide the contents of the block. Whenever reading and writing back a block, the block must be re-encrypted prior to being written back. If the encryption scheme is secure, the server should not be able to tell whether the block's content has changed upon seeing the new ciphertext.

6.5 Analysis

We will now discuss why the aforementioned binary-tree ORAM construction
1) preserves obliviousness, and
2) is correct (except with negligible probability).

6.5.1 Obliviousness

Obliviousness is easy to see. First, whenever a block is accessed, it is assigned to a new path and the choice of the new path is kept secret from the server. So whenever the block is accessed again, the server simply sees a random path being accessed. Second, the entire eviction process does not depend on the input requests at all.

6.5.2 Correctness

Correctness is somewhat trickier to argue. As mentioned earlier, to argue correctness, we must argue why no overflow will ever occur except with negligible probability — as long as the bucket size Z is set appropriately.

Claim 6.5. *Bucket size and overflow probability: If the bucket size Z is super-logarithmic in N, then over any polynomially many accesses, no bucket overflows except with negligible in N probability.*

Proof. The full proof uses results from queueing theory,[25] in particular Burke's theorem.[26] We will give a heuristic argument that does not require any specialized knowledge.

- The root bucket (level 0 of the ORAM tree) receives exactly 1 incoming block with every access, and it is evicted on every access, so the root bucket always ends up empty.
- There are N leaf buckets, and N blocks are assigned to them at random. We leave it as an exercise to check that the probability that more than $(\log N)^2$ blocks are assigned to any one leaf is negligible than N (in other words: decays faster than N^k, for every k).
- Now consider a bucket, neither root nor leaf, at level $i \geq 1$ of the ORAM tree. A block will enqueue on one out of every 2^i accesses, and with probability $\frac{1}{2^{i-1}}$, the bucket is chosen for eviction.

So for every non-leaf and non-root level of the tree, with every ORAM access, the dequeue probability is twice as large as the enqueue probability. This situation is well-known in queueing theory as the "M/M/1 queue":

- Every time step, with probability p, an item enqueues;
- Every time step, with probability $2p$, an item dequeues if the queue is non-empty.

Since the bucket is drained (on average) twice as fast as it is filled, we expect that it is very unlikely for a lot of blocks to accumulate in any one bucket.

[25] https://en.wikipedia.org/wiki/Queueing_theory
[26] https://en.wikipedia.org/wiki/Burke%27s_theorem

Indeed, one can prove an exponential bound on the probability that any one bucket gets too full:

$$\Pr[\text{number of items in queue} > R] \leq \exp(\Omega(-R)).$$

Unfortunately, this does not quite finish our analysis of the ORAM tree. The reason is that the buckets in the ORAM tree are not independent, and our informal argument above ignored possible dependence between buckets. It turns out that this gap can be filled using Burke's theorem. But at this point the reader should already be convinced that the result is at least quite plausible. □

6.6 Binary-tree ORAM: recursion

So far, we have cheated and pretended that the client can store a large position map. We now describe how to get rid of this position map. The idea is simple: Instead of storing the position map on the client side, we simply store it in a smaller ORAM denoted $\mathsf{posORAM}_1$ on the server. The position map of $\mathsf{posORAM}_1$ will then be stored in an even smaller ORAM denoted $\mathsf{posORAM}_2$ on the server, and so on. As long as the block size is at least $\Omega(\log N)$ bits, every time we recurse, the ORAM's size reduces by a constant factor; and thus $O(\log N)$ levels of recursion would suffice.

We can thus conclude with the following theorem.

Theorem 6.6 (Binary-tree ORAM[27]). *For any super-constant function $\alpha(\cdot)$, there is an ORAM scheme that achieves $O(\alpha \log^3 N)$ cost for each access: each logical request will translate to $O(\alpha \log^3 N)$ physical accesses; and moreover, the client is required to store only $O(1)$ number of blocks.*

Note that in the total cost $O(\alpha \log^3 N)$, an $\alpha \log N$ factor comes from the bucket size; another $\log N$ factor comes from the total height of the tree; and the remaining $\log N$ factor comes from the recursion.

6.7 Path ORAM

The design of the above binary-tree ORAM is a little silly: Whenever we visit a triplet of buckets for eviction, we only evict one block. For this reason, the bucket size needs to be super-logarithmic to get negligible failure probability. It turns out that if we instead use a more aggressive eviction algorithm, and with a more sophisticated proof, the bucket size can be made constant. At this point, we are ready to introduce an improved version called Path ORAM.

Unlike the above binary-tree ORAM, in Path ORAM, every bucket has constant size (maybe 4 or 5), except the root bucket which is super-logarithmic in size. Every time we access some path to fetch a block, we also perform eviction on the same path. In particular, we will rearrange the blocks on the path in the most aggressive manner possible: We want to move the blocks as close to the leaf level as possible, but *without violating the path invariant*. With Path ORAM,

[27] https://eprint.iacr.org/2011/407.pdf

every access operation touches a single path, hence the name Path ORAM. The cost of each access is $O(\alpha \log^2 N)$ for an arbitarily small superconstant function α.

6.7.1 Other applications of ORAM

ORAM promises many potential applications. For instance, in Large Language Models (LLMs), a commonly used technique is called Retrieval Augmented Generation (RAG). RAG parses the user's query and looks up the relevant locations in a large knowledge base to fetch the relevant entries. If we want to protect the privacy of users' queries in LLMs, it is also crucial to hide the access patterns, and this would be a great application of ORAM.

Besides its usage in secure processors, ORAM is critical for scaling cryptographic multi-party computation (MPC) to big data. Traditional MPC techniques require us to express the desired computation as a circuit. However, in the real-world, we program assuming the Random Access Machine (RAM) model where a CPU can dynamically read and write a memory array. Translating a RAM program to a circuit brute-force would incur a linear (in the memory size) cost for each memory access! For example, a binary search in a sorted database requires only logarithmic time on a RAM, but it requires linear cost when expressed as a circuit. Fortunately, ORAM again comes to our rescue. There is a line of work on RAM-model MPC, and the idea is that we first translate the RAM to an Oblivious RAM, and at this point all the memory accesses are safe to reveal. At this moment, we can use MPC to securely emulate a "secure processor" that performs computation while accessing memory obliviously.

6.8 ORAM takeaways

1. *Oblivious RAM* is a system to hide memory access patterns from a server.
2. The server stores encrypted data blocks in a binary tree, and it does not learn which blocks correspond to which memory items.
3. Every time the ORAM client accesses a block, it writes that block back to the root.
4. A randomized eviction procedure moves blocks away from the root, so individual nodes of the tree do not overflow.