Intuitive Advanced Cryptography

Nguyen Thoi Minh Quan [*]

Abstract

Advanced cryptographic protocols such as zero-knowledge proof, homomorphic encryption, commitment schemes, blind signature, lattice based cryptography, secure multi-party computation, oblivious transfer, etc are beautiful but look like magic to general software engineers. They're notoriously difficult to understand. Unfortunately, I dropped out of PhD program before I could understand advanced math and formal cryptographic proofs. Therefore, I had to find ways to work around this challenge. This article shares with you my notes with the hope that you will have an *intuitive understanding* of these fascinating cryptographic protocols. Furthermore, as a security engineer, to bypass cryptographic proofs, I'll use *attack oriented* approach to analyze protocols.

Contents

[*] https://www.linkedin.com/in/quan-nguyen-a3209817, https://scholar.google.com/citations?user=9uUqJ9IAAAAJ, https://github.com/cryptosubtlety, msuntmquan@gmail.com

1 Introduction

To understand this article you must finish reading the following 4 books: [1], [2], [3], [4]. I'm just kidding, don't run away :)

The previous paragraph is a joke but it has certain elements of truth. Advanced cryptographic protocols are built upon various advanced mathematical concepts which I honestly don't fully understand. To overcome this obstacle, we need to find a simple interpretation and representation of mathematical concepts. Our mathematical interpretation and representation may not be accurate, but we're fine with it as long as it helps us move forward in understanding complicated cryptographic protocols. In the same spirit, we'll sacrifice accuracy in describing cryptographic protocols, instead we'll focus on their intuitive understanding. Dear mathematicians and cryptographers, please forgive me for misunderstanding and misrepresenting your true math and cryptographic protocols :)

A final note before we move on to background section. Knowing math is not enough to understand cryptography. In the end, cryptography is a part of security, i.e., designing cryptographic protocols that work is just a start, we have to make sure that they're safe. We're not going to dig into cryptographic proofs, instead we'll take *attack oriented* approach, i.e., we'll look at cryptographic protocols from attacker's point of view.

2 Background

2.1 Group

A group $(G, +)$ is a set of elements G, together with group operation $+$ between any two elements of G.

If it sounds abstract to you then imagine the set of integers where you only define addition operation $+$ and ignore multiplication operation $.$, i.e., $(\mathbb{Z}, +)$ forms a group. Another example is the following: given an integer number p, consider the set $\mathbb{Z}_p = \{0, 1, ..., p-1\}$ and the group operation $+$ is addition mod p then you have a finite group $(\mathbb{Z}_p, +)$. This group is finite because the set $\mathbb{Z}_p$ has finite p elements. The number of elements in a finite group is called the *order* of a group, denoted as $|G|$.

Now, let's look closer at our group $(\mathbb{Z}_p, +)$. The number 0 is called the *identity* of the group because $\forall x \in \mathbb{Z}_p : x + 0 = 0 + x = x$. Let's see what we can do by using only number 1 and addition $\mod$ p operation:

$$\begin{aligned}
&1 = 1 \mod p \\
&1 + 1 = 2 \mod p \\
&1 + 1 + 1 = 3 \mod p \\
&\underbrace{1 + \cdots + 1}_{p-1 \text{ times}} = p - 1 \mod p \\
&\underbrace{1 + \cdots + 1}_{\text{p times}} = 0 \mod p
\end{aligned}$$

We notice that with 1 and addition $\mod p$ operation we can generate all elements of $\mathbb{Z}_p$ including identity element 0. An element (in this case 1) that generates all elements of the group using group operation is called a *generator*.

For concreteness, let's say $p = 6$. The group $(\mathbb{Z}_6, +)$ has 6 elements $\{0, 1, 2, 3, 4, 5\}$. The element 1 generates all elements in $(\mathbb{Z}_6, +)$, so it's a generator of $(\mathbb{Z}_6, +)$. How's about number 2? We have: 2, $2 + 2 = 4 \mod 6$, $2 + 2 + 2 = 0 \mod 6$, i.e., 2 only generates three elements in the group $\mathbb{Z}_6 : \{0, 2, 4\}$. If we only look at the set $\{0, 2, 4\}$ with addition $\mod 6$ operation then we notice that it by itself is another group and it's contained in a larger group $(\mathbb{Z}_6, +)$. The group $(\{0, 2, 4\}, + \mod 6)$ is called a *subgroup* of the group $(\mathbb{Z}_6, +)$. A classic Larange's theorem tells us that order of a subgroup divides the order of the group. In this case, the order of the subgroup $(\{0, 2, 4\}, + \mod 6)$ is 3 which divides 6 - the order of the group $(\mathbb{Z}_6, +)$.

Before ending this section about group, let's introduce a convenient shortcut notation. If X is an element of group G, instead of writing $\underbrace{X + \cdots + X}_{\text{k times}}$, we write kX. The smallest number n for which $nX = 0$ is called the *order* of X. In our previous example, in $\mathbb{Z}_6$, the order of number 1 is 6, while the order of number 2 is 3 because

$$\underbrace{1 + \cdots + 1}_{\text{6 times}} = 0 \mod 6 \qquad \underbrace{2 + \cdots + 2}_{\text{3 times}} = 0 \mod 6$$

In summary, on one side, while group is an abstract mathematical concept, it's enough to think of group as $(\mathbb{Z}_p, +)$. On the other side, don't limit your imagination of group element as only number, a group element can be anything, for instance, a matrix, a polynomial, a point on curve, etc.

2.2 Field

A field is a set F, together with two operations addition and multiplication. You deal with field all the time even though you aren't aware of it. The real numbers $\mathbb{R}$, the rational numbers $\mathbb{Q}$, the complex numbers $\mathbb{C}$ with regular addition and multiplication operations are all classical fields.

In cryptography, we often use the following finite field: given a prime number p, consider the set $\{0, 1, ..., p-1\}$ together with addition and multiplication operations $\mod p$ then we have a finite field denoted as $\mathbb{F}_p$. For instance, the finite field $\mathbb{F}_5$ has 5 elements $\{0, 1, 2, 3, 4\}$ and its operations work as follow: $2 + 4 = 1 \mod 5$, $3 * 4 = 12 = 2 \mod 5$. Finally, $\mathbb{F}_p$ is a special case of $\mathbb{F}_{p^k}, k \geq 1$ that is also often used in practice.

2.3 Elliptic curve

An elliptic curve E is a set of points $P(x, y)$ where their coordinates x, y satisfy the equation $y^2 = x^3 + ax + b\,(E)$ where $x, y \in \mathbb{F}_p$, p is a prime number. For instance, let's look at the following curve:

$$\begin{aligned} p &= 17 \\ a &= 14 \\ b &= 4 \\ y^2 &= x^3 + ax + b\,(E) \end{aligned}$$

The coordinates x, y are defined over $\mathbb{F}_{17}$ and they satisfy the equation $y^2 = x^3 + 14x + 4 \mod 17$. The point $P(8, 13)$ is on the curve E because $13^2 = 8^3 + 14 * 8 + 4 \mod 17$.

An extraordinary property of elliptic curve is that we can define addition operation $+$ between its points, i.e., given 2 points $P, Q \in E$, the operation $P + Q$ makes sense and the result is another point $R \in E$. How points addition operation $+$ is defined is not our concern, what we care is that $(E, +)$ *forms a group*. In practice, we don't work directly with the group $(E, +)$, instead we choose a *base point* G and works with the *subgroup* generated by G, i.e., the group $(\{0, G, \cdots (q-1)G\}, +)$ where q is G's order. By Larange's theorem $|G|$ divides $|E|$ and we call $|E|/|G|$ *cofactor*. The value of cofactor plays a significant role in the security of cryptographic protocols.

Why is elliptic curve widely used in cryptography? From *mathematical* point of view, it's because elliptic curve has nice mathematical property as its points form group structure. However, having group structure is not enough to be useful in cryptography. From a *security* point of view, elliptic curve is popular because the following *discrete log problem* is extremely hard: Given point X, base point G, find x such that $X = xG$. We often write $x = \log_G(X)$.

2.4 Polynomial modulus

I guess you're familiar with polynomial, for instance, let $P(x)$ be a polynomial of degree k over $\mathbb{F}_q$, i.e., $P(x) = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_0$ where $a_0, \cdots, a_{k-1} \in \mathbb{F}_q$. The operation $\mod n$ where n is a number is also a familiar operation. How's about their combination, i.e., $\mod P(x)$ operation?

It's best to take a look at a specific example. Let's assume $P(x) = x^5 + x + 1$. What does $\mod x^5 + x + 1$ look like? Let's go back to see what $\mod 7$ means. In $\mod 7$ world, 7 is equal to 0. For instance: $8 + 4 = 7 + 1 + 4 = 1 + 4 = 5 \mod 7$. Similarly, in $\mod x^5 + x + 1$ world, $x^5 + x + 1$ equals to 0 and you can replace $x^5 + x + 1$ with 0 or you can replace x^5 with $-x - 1$. For instance: let's say $P_1(x) = x^4, P_2(x) = x^6 + 1$ and if we add and multiply them $\mod x^5 + x + 1$, we have:

$$\begin{aligned}
&P_2(x) = x^6 + 1 = x * (x^5) + 1 = x * (-x - 1) + 1 = -x^2 - x + 1\\
&P_1(x) + P_2(x) = x^4 - x^2 - x + 1\\
&P_1(x) * P_2(x) = x^4 * (-x^2 - x + 1) = -x^6 - x^5 + x^4\\
&= -x * x^5 - (-x - 1) + x^4 = -x * (-x - 1) + x + 1 + x^4 = x^4 + x^2 + 2x + 1
\end{aligned}$$

In this article, we'll use polynomials in $\mathbb{Z}_q[x]/(P(x))$, i.e., polynomials whose coefficients are in $\mathbb{Z}_q$ and every operation is $\mod P(x)$.

2.5 Alice, Bob and Eve

In cryptographic protocols, we often have 3 characters: Alice, Bob and Eve. Typically, Alice wants to interact with Bob while Eve tries to attack Alice, Bob and their communication. The nonobvious question in the previous description is whether Alice and Bob trust each other. A common mistake in cryptographic implementation is to assume that Alice and Bob are both trustworthy. It's better to assume that Alice and Bob may be malicious, i.e., Alice and Bob must protect themselves while interacting with each other. Life is hard. Wait a second, are you saying that we shouldn't trust anyone at all? No, I didn't say that but it's your right to interpret it in any way you want :)

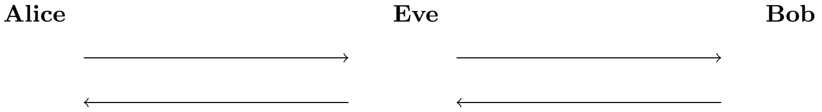
Note that, in practice, the characters Alice, Bob and Eve are not human, they're computer programs, so don't be surprised if I say Alice's memory, Alice's state, Alice's program, etc.

2.6 Quantum computer

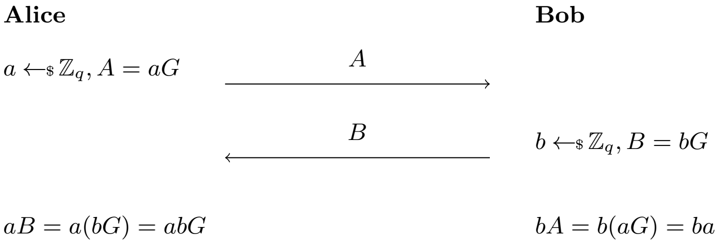
In classical computers, the basic unit is a *bit* that is either a 0 or 1 at any moment. In quantum computers, the basic unit is a *qubit* that can exist at both states 0 and 1 at the same time. This gives quantum computer more computing power than classical one. That's it for quantum computer section. You may wonder why this section is so short. If I'm smart enough to understand quantum computer then I'm not here to write this article about intuitive understanding of cryptographic protocols :)

From cryptographic point of view, the reason we care about quantum computer is because Peter Shor [5] showed us that quantum computer can solve *discrete log problem* and *factoring problem*, i.e., it can break our most popular *public key* algorithms. I didn't tell you that quantum computer can break *symmetric key* encryption or AES, did I? No, quantum computer can not break AES. The only known quantum attack against encryption is to use Grover's quantum search algorithm [6] that can search N entries in $\sqrt{N}$ time. Therefore, if you're worried that quantum computer can break your symmetric key encryption, then increase your AES's key from 128 bits to 256 bits. Don't waste money for snake oil salesman that sells you quantum-safe symmetric key encryption.

3 Elliptic curve Diffie-Hellman (ECDH) key exchange



Imagine that Alice wants to talk to Bob in a secure way so that Eve who eavesdrops on Alice and Bob's communication can't understand Alice and Bob's messages. Easy. Alice encrypts her plaintexts and sends ciphertexts to Bob. Not so fast. You said Alice encrypts message, but what key does she use for encryption? How could Alice securely send her encryption key to Bob in the first place? Remember, Eve always listens. Sounds difficult? It's indeed a really hard problem. Fortunately, Diffie and Hellman [7] invented an elegant protocol that fundamentally changed cryptography forever.

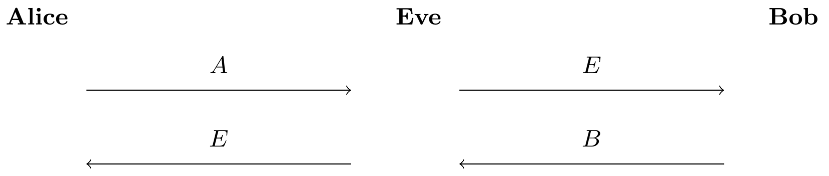

Alice
Bob
$a \leftarrow_{\$} \mathbb{Z}_q, A = aG$
A
B
$b \leftarrow_{\$} \mathbb{Z}_q, B = bG$
$aB = a(bG) = abG$
$bA = b(aG) = baG$

Let's say Alice and Bob agree in advance that they'll use some standard elliptic curve E with base point G of order q. Alice generates a random private key $a \leftarrow_{\$} \mathbb{Z}_q$, computes her public key $A = aG$ and sends A to Bob. Bob generates a random private key $b \leftarrow_{\$} \mathbb{Z}_q$, computes his public key $B = bG$ and sends B to Alice. Now Alice computes $aB = a(bG) = abG$ and Bob computes $bA = b(aG) = baG$. We notice that Alice and Bob compute the same result abG and it's their shared key that can be used for encryption.

All right, math works. Let's convince ourselves that this protocol is secure. What does Eve know? By eavesdropping the communication between Alice and Bob, Eve knows A (aka aG) and B (aka bG). As the discrete log problem in elliptic curve is hard, knowing aG and bG doesn't help finding a and b and hence Eve can't compute abG. The shared key abG between Alice and Bob is protected from Eve. You know what, there was a flaw in my argument. Eve's goal is to compute abG assuming she knows aG and bG. In my argument, I assumed that she has to find a, b (i.e. to solve discrete log problem) first before she can compute abG, but my assumption is wrong because no one knows for sure it's the only way to compute abG. There may exist other ways to compute abG without computing a, b. Therefore, strictly speaking, we have to make a new security assumption to guarantee that the protocol is safe. It's called Computational Diffie-Hellman (CDH) assumption: given G, aG, bG, it's infeasible to compute abG. Cryptography is subtle! A closely related assumption is Decisional Diffie-Hellman (DDH): given G, aG, bG, it's infeasible to distinguish abG from a random point in E.

3.1 Active attacker and authenticated key exchange

In the previous section, we assume that Eve only eavesdrops on the communication between Alice and Bob, i.e., we assume that Eve is passive. How's about the case when Eve actively interferes with the communication between Alice and Bob?



The attack is instead of forwarding Alice's public key A to Bob, Eve sends her public key E to Bob. Similarly, instead of forwarding Bob's public key B to Alice, Eve sends her public key E to Alice. In the end, Eve establishes a shared key aeG with Alice and a shared key beG with Bob. I.e., Eve can decrypt the ciphertexts sent from Alice or Bob. The fundamental issue is that when Alice (Bob) receives public key from the other party, she (he) has no way to know whether the public key actually comes from Bob (Alice).

To prevent the previous attack, Alice and Bob use digital signature to sign their ECDH's public keys A and B. As Eve can't generate valid signature on behalf of Alice or Bob, the above attack doesn't work anymore.

3.2 ElGamal encryption

A straightforward application of ECDH key exchange is ElGamal encryption[8][9]. The basic idea is to use ECDH to establish shared secret and use the shared secret to mask out the message.

To encrypt message m, we create a point P_m on E that has x coordinate as m. When Bob receives Alice's public key A, instead of sending his public key B to Alice as in ECDH key exchange, he sends both B and $bA + P_m$ to Alice, i.e., $(c_1, c_2) = (B, bA + P_m)$. When Alice receives (c_1, c_2), she computes $c_2 - ac_1 = bA + P_m - aB = baG + P_m - abG = P_m$. The message m is the x coordinate of point P_m.

4 Interactive zero-knowledge proof

Have you ever heard of zero-knowledge proof? No? You don't read news, do you? It's a buzzword in every cryptocurrency protocol and you've never heard of it? I didn't ask you whether you know what it is, I only asked you whether you've heard of it :)

Informally, zero-knowledge proof of knowledge is a proof where you prove to other people that you know something without revealing it. Does it sound like magic? It's indeed magic. The technicality behind zero-knowledge proof is complicated, so we'll take a look at concrete protocols and analyze them. All protocols in this section require interaction between parties, which is why this section is called *interactive* zero-knowledge proof.

4.1 Schnorr identification protocol

In this section, we'll analyze Schnorr identification protocol[10]. It's a cool protocol that serves as basis for digital signatures, so pay attention to this section.

Suppose E is an elliptic curve with base point G of order q, Alice's private key is x, her public key is $X = xG$. Alice wants to prove to Bob that she knows x without revealing to Bob what x is. Designing a protocol that reveals nothing about x is easy: Alice does nothing. On the hand, designing a protocol where Alice convinces Bob that she knows x is even easier: she sends x to Bob. The hard part is how to achieve two properties at the same time. Schnorr [10] invented the following protocol:

$$\begin{array}{lcl}
\textbf{Alice} & & \textbf{Bob} \\
r \leftarrow_{\$} \mathbb{Z}_q, R = rG & \xrightarrow{\quad\quad R \quad\quad} & \\
 & \xleftarrow{\quad\quad c \quad\quad} & c \leftarrow_{\$} C \\
e = r + cx \mod q & \xrightarrow{\quad\quad e \quad\quad} & \\
 & & eG \overset{?}{=} R + cX
\end{array}$$

In the above protocol, C is called challenge space and c is randomly generated from C $c \leftarrow_{\$} C$. In the end of the protocol, Bob verifies whether eG equals to $R + cX$ and if it does, then Bob is convinced that Alice knows x. Note that if Alice is honest, i.e., e is indeed equal to $r + cx$ then multiplying both sides of equation $e = r + cx$ with G, we get $eG = rG + cxG = R + cX$.

Why does Bob know nothing about x? The only information related to x that Bob receives is $e = r + cx \mod q$. No matter what x or cx is, adding a random value $r \mod q$ to it makes the result indistinguishable from the random number, i.e., r completely masks out the value of cx.

Another question is why c needs to be unpredictable? If Alice can predict c in advance then she can generate a random e, compute eG and $R = eG - cX$, i.e., she can generate R and e without knowing r and x but Bob's verification equation is still satisfied.

All right, we've convinced ourselves that this protocol doesn't leak information about x. The hard part is how Bob is convinced that Alice knows x. If Alice does not know x then no matter what we do with Alice, we can't get x. Now, if we're allowed to control the execution environment in which Alice is running and if by clever manipulating that execution environment, we somehow can get x then Alice definitely knows x (otherwise, where does x

come from so that we can get it?). Are you confused now? No worries, I was even more confused than you're :) For this protocol, what we're going to do is to wait until Alice finishes generating r and $R = rG$ and then we clone Alice, i.e., make 2 copies of Alice with *the same value* r. In one copy, Alice receives c, sends e and in another copy, Alice receives c', sends e'. We have $eG = R + cX, e'G = R + c'X \to (e' - e)G = (c' - c)X \to (e' - e)/(c' - c)G = X$ and hence $x = (e' - e)/(c - c')$. An obvious question is whether Bob can use the same trick to extract x. To do this, Bob must have the capability to force 2 copies of Alice to use the same r. In theory, Bob doesn't have that capability. In practice, if you don't implement Alice's multithreading program carefully, you may be vulnerable to this attack. Don't say I didn't warn you :)

4.2 Chaum-Pedersen protocol

Let's say E is an elliptic curve and Alice's private key is x. Let G and H be two points on E. Alice publishes $U = xG$, $V = xH$. She wants to convince Bob that $log_G(U) = log_H(V)$ and to prove that she knows x without revealing what x is[11].

Alice		**Bob**
$r \leftarrow_{\$} \mathbb{Z}_q, P = rG, Q = rH$	$\xrightarrow{\quad P, Q \quad}$	
	$\xleftarrow{\quad c \quad}$	$c \leftarrow_{\$} C$
$e = r + cx \mod q$	$\xrightarrow{\quad e \quad}$	
		$eG \stackrel{?}{=} P + cU$ $eH \stackrel{?}{=} Q + cV$

Adding random value $r \mod q$ to cx completely masks out its value and leaks zero information about x to Bob.

Similar to Schnorr identification protocol, to prove that Alice knows x and $x = log_G(U) = log_H(V)$, we wait until Alice finishes generating r and clone her. In one copy, Alice receives c, sends e and in another copy, Alice receives c', sends e'. We have:

$$\begin{align} eG &= P + cU \tag{1} \\ e'G &= P + c'U \tag{2} \\ eH &= Q + cU \tag{3} \\ e'H &= Q + c'U \tag{4} \end{align}$$

From equations (1) and (2), we deduce $(e - e')/(c - c')G = U$, i.e., $x = (e - e')/(c - c')$. From equations (3) and (4), we deduce $(e - e')/(c - c')H = Q$, i.e., $x = (e - e')/(c - c')$. We observe that we extract the same x in two set of equations.

5 Fiat-Shamir heuristics and noninteractive zero-knowledge proof

In the previous section, we looked at interactive protocols. In practice, interactive protocols have a few disadvantages: they cost bandwidth and they require all parties to be online and synced. Fiat and Shamir showed that we can transform an interactive protocol to a noninteractive one using cryptographic hash function such as SHA256.

5.1 Schnorr signature

To transform Schnorr interactive identification protocol to noninteractive protocol, Alice uses cryptographic hash function to compute $c = \text{hash}(G, X, R)$, instead of waiting for Bob to generate random c. It's important to hash *everything* in the transcript (see [12]).

$$\begin{array}{lcl}
\textbf{Alice} & & \textbf{Bob} \\
r \leftarrow_{\$} \mathbb{Z}_q, R = rG & & \\
c = \text{hash}(G, X, R) & & \\
e = r + cx \mod q & \xrightarrow{\quad R, e \quad} & \\
 & & c = \text{hash}(G, X, R) \\
 & & eG \overset{?}{=} R + cX
\end{array}$$

Let's take a closer look to see whether this noninteractive protocols is as safe as the interactive one. In Schnorr identification protocol section, we noticed that if Alice can predict c in advance then the protocol is not safe anymore. Does Fiat-Shamir's transform make the protocol insecure? The answer is no, but to convince ourselves that it's the case is nontrivial. Let's recall a property of cryptographic hash function. For cryptographic hash function, you can't control both the input and output at the same time, i.e., if you choose your input first then the output looks like random number and if you choose your output first, you can't find the input that hashes to your chosen output. Therefore, Alice can't choose c first and then compute R such that $\text{hash}(R) = c$. In other words, Alice has to generate R first and then compute $c = \text{hash}(R)$. But once R is determined, the output $c = \text{hash}(R)$ looks like random number and is out of Alice's control. This is exactly the property that we needed for Schnorr identification protocol's security. In general, the order of cryptographic operations is critical to protocols' security, so watch out your program's state machine.

Schnorr signature [13] is a small modification of the above protocol. To sign message m, instead of computing $c = \text{hash}(R)$, we compute $c = \text{hash}(R||m)$ and the signature is simply $(R, e = r + cx \mod q)$.

5.2 Noninteractive Chaum-Pedersen protocol

After reading the previous section, it's easy to guess that $c = \text{hash}(G, H, U, V, P, Q)$. The complete protocol is shown below:

Alice		**Bob**
$r \leftarrow_{\$} \mathbb{Z}_q, P = rG, Q = rH$		
$c = \text{hash}(G, H, U, V, P, Q)$		
$e = r + cx \mod q$	$\xrightarrow{P, Q, e}$	
		$c = \text{hash}(G, H, U, V, P, Q)$
		$eG \stackrel{?}{=} P + cU$
		$eH \stackrel{?}{=} Q + cV$

6 Ring signature

To verify a signature, the verifier has to know the signer's public key. Therefore, if you use digital signature such as Schnorr signature to sign a message, then everyone knows that it's you that signed the message. This is, in fact, an expected security property. However, there are certain use cases where you don't want to be tracked and held responsibility for what you signed. Ring signature [14] allows a signer to blend himself into a set of n users, i.e., if there are n users and only one of them signs the message, the signature doesn't leak who was the signer, but the verifier can verify that the signature is signed by 1 of n users.

We'll describe a protocol based on Schnorr signature and OR-proof technique[15], [16]. Let's recall ourselves how Schnorr signature looks like:

User		**Verifier**
$r \leftarrow_{\$} \mathbb{Z}_q, R = rG$		
$c = \text{hash}(R, m)$		
$e = r + cx \mod q$	$\xrightarrow{R, e}$	
		$c = \text{hash}(R, m), eG \stackrel{?}{=} R + cX$

Suppose that we have n users user$_1, \cdots,$ user$_n$ and user$_i$ is the signer. User$_i$ uses his private key x_i to sign the message m as usual. However, user$_j, j \neq i$ doesn't use his private key but still has to produce something that looks like signature so that the verifier can verify. You may wonder whether the previous sentence makes any sense at all and you're right, it doesn't. If a user can produce a signature without using his private key, do we have a vulnerability in signature scheme? The only way to overcome this seemingly paradox is to give user$_j$ more capability and freedom so that he can cheat. The extra capability that we give user$_j$ is to let him choose c_j in advance himself. In particular, user$_j, j \neq i$ first chooses c_j himself, randomly generates e_j and computes $R_j = e_jG - c_jX_j$. Now the verifier's equation $e_jG_j \stackrel{?}{=} R_j + c_jX_j$ is still satisfied. What's happening here? The trick was that user$_j, j \neq i$ not only chooses c_j himself but also computes everything in an unusual order $c_j \to e_j \to R_j$ while the regular order is $R_j \to c_j \to e_j$. This section once again stresses how important the order of cryptographic operations is to protocols' security, so watch out your program's state machine. Using the above trick, user$_j$ can produce signature without knowing or using private key x_j.

We're close but we're not done yet. One important detail that I left out was what user$_i$'s c_i is. At high level, there must be some c that is computed by using hash of something as in Fiat-Shamir heuristics. How is that c related to c_i and $c_j, j \neq i$? The paper [15], [16] proposed a beautiful solution.

Users		**Verifier**
User$_j$ $j \neq i : c_j \leftarrow_{\$} C$		
$e_j \leftarrow_{\$} \mathbb{Z}_q, R_j = e_jG - c_jX_j$		
User$_i$ $: r_i \leftarrow_{\$} \mathbb{Z}_q, R_i = r_iG$		
$c = \text{hash}(m, R_1, \cdots, R_n)$		
$c_i = \oplus_{j \neq i} c_j \oplus c$		
$e_i = r_i + c_ix_i \mod q$	$\xrightarrow{R_1, \cdots, R_n, c_1, \cdots, c_n, e_1, \cdots, e_n}$	
		$c = \text{hash}(m, R_1, \cdots, R_n)$
		$c \overset{?}{=} c_1 \oplus \cdots \oplus c_n$
		$e_kG \overset{?}{=} R_k + c_kX_k, k = 1 \cdots n$

User$_j, j \neq i$ randomly generates c_j, e_j, computes $R_j = e_jG - c_jX_j$ as mentioned above. User$_i$ on the other hand, generates random r_i, computes $R_i = r_iG$ as usual. Now c is computed as hash$(m, R_1, \cdots, R_n)$ which binds all $R_1, \cdots, R_n$ to m. The trick is to use $c_i = \oplus_{j \neq i} c_j \oplus c$ such that $c = c_1 \oplus \cdots \oplus c_n$. The core idea is that there is only 1 constraint $c = c_1 \oplus \cdots \oplus c_n$ while there are n variables $c_1, \cdots, c_n$. I.e., the verifier forces n users to have 1 and only 1 choice, which corresponds to the fact that 1 user signs the message m using his real key while the remaining $n-1$ users are free to cheat in producing signatures. Finally, it's obvious from the protocol that the verifier can't distinguish c_i, e_i, R_i from the remaining values and hence the verifier doesn't know who user$_i$ is.

7 Shamir's secret sharing

"Don't put all your eggs in one basket". In this section, we will follow this wise advice to protect our secret.

Let's say we have a valuable secret s to protect. Storing s in one system doesn't sound a like a good strategy. A better approach is to break s into pieces and store different pieces in different systems. This is to increase the attack cost because the attacker has to compromise multiple systems to get s. We'll break s into n pieces $s_1, \cdots, s_n$ so that with any k pieces we can reconstruct s, but with $k-1$ pieces, it's not enough to determine s.

Let $P(x)$ be a polynomial of degree k over $\mathbb{F}_p$, i.e., $P(x) = a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_0$ where $a_0, \cdots, a_{k-1} \in \mathbb{F}_p$. We will use the following mathematical property of polynomial $P(x)$: if we know the values $P(x_1), \cdots, P(x_k)$ at k different $x_1, \cdots, x_k$ then we can uniquely reconstruct $P(x)$, but knowing the values $P(x_1), \cdots, P(x_{k-1})$ at any $k-1$ different $x_1, \cdots, x_{k-1}$ isn't enough to determine $P(x)$. Why's that? Let's look at a small example and convince ourselves that this is true. Let's say $k = 2$, $P(x) = a_1x + a_0$. The graph $(x, y = a_1x + a_0)$ is a straight line. Knowing any 2 points $(x_0, y_0 = a_1x_0 + a_0)$ and $(x_1, y_1 = a_1x_1 + a_1)$ uniquely determines our straight line. On the other hand, if we only know 1 point $(x_0, y_0 = a_1x_0 + a_0)$, we can't determine our straight line because there are so many lines that go through that point.

Using the above property, Shamir's protocol[17] is simple yet elegant: set $a_0 = s$ and randomly generate $a_1, \cdots, a_{k-1}$. The n pieces that are stored at n different systems are $s_1 = P(1), s_2 = P(2), \cdots, s_n = P(n)$. Knowing any k of $\{s_1, \cdots, s_n\}$ uniquely identify $P(x)$ and hence $s = a_0 = P(0)$, while knowing $k - 1$ of $\{s_1, \cdots, s_n\}$ doesn't determine $P(x)$ and hence doesn't determine $s = P(0)$.

8 Secure multi-party signature computation

Let's say we have a private key x that is used to produce digital signature. Following beautiful Shamir secret sharing, to protect x, we'll break x into pieces and store them at different systems. Is our job of managing secret done? If we put more thought on it, we'll see that Shamir secret sharing has a weakness when deployed in practice. As an attacker, I don't care how x is broken into pieces and stored at multiple systems, I wait until x is reconstructed at some system and only attack that single system.

To mitigate the previous attack, we need a stronger security requirement. We'll break the secret x into pieces $x_1, \cdots, x_n$ as before and store them at multiple systems. These systems only use their own secrets $x_1, \cdots, x_n$. We let these systems interact with each other and in the end, they collectively produce a digital signature corresponding to $\mathbf{x}$, but they keep $x_1, \cdots, x_n$ for themselves, i.e., they don't leak $x_1, \cdots, x_n$ to other systems or anyone else. In such a design, the attack cost significantly increases because to get x, the only way is to compromise all systems.

8.1 Secure multi-party Schnorr signature computation

We'll consider a special case when there are only $n = 2$ systems (for Schnorr signature, it's easy to extend to arbitrary n systems). The secure two-party Schnorr signature computation is pretty straightforward. Split secret key x into 2 keys x_1, x_2 such that $x = x_1 + x_2 \mod q$. Similarly, when we generate r, we split r into two parts such that $r = r_1 + r_2 \mod q$.

Party 1		**Party 2**
Private keyx_1		Private keyx_2
$r_1 \leftarrow_\$ \mathbb{Z}_q, R_1 = r_1G$	$\xrightarrow{\qquad R_1 \qquad}$	
	$\xleftarrow{\qquad R_2 \qquad}$	$r_2 \leftarrow_\$ \mathbb{Z}_q, R_2 = r_2G$
$R = R_1 + R_2$		$R = R_1 + R_2$
$c = \text{hash}(R\|\|m)$		$c = \text{hash}(R\|\|m)$
$e_1 = r_1 + cx_1 \mod q$		$e_2 = r_2 + cx_2 \mod q$

In the above protocol, each party computes its own R_1, R_2 locally and exchange R_1, R_2 to each other. R is the sum of R_1 and R_2. Each party again computes c, e_1, e_2 locally. The signature of message m is the sum $(R = R_1 + R_2, e = e_1 + e_2)$. Why's that? We have:

$$
\begin{aligned}
R &= R_1 + R_2 = r_1G + r_2G = (r_1 + r_2)G = rG \\
e &= e_1 + e_2 = r_1 + cx_1 + r_2 + cx_2 = (r_1 + r_2) + c(x_1 + x_2) \\
&= r + cx \mod q
\end{aligned}
$$

We see that (R, e) is a valid m's signature signed by x. In summary, each party protects its own key x_1, x_2, generates its signature locally and the final signature is the sum of the generated signatures.

9 Pairing based cryptography

If you reach this section, you have seen enough protocols that are based on elliptic curve. All of our protocols so far use only 1 elliptic curve. However, cryptographers are not satisfied yet. They want to use 2 elliptic curves at the same time. They drive me crazy :) Do you know how miserable an incompetent security engineer's life like mine is? I have to deal with more and more and more complicated math and it never stops. After finishing this article, I'll retire from cryptography, but you, my fellow, should continue the mission to make the world a safer place by using amazing cryptographic protocols :)

Pairing[18] is defined as a map e: $E_1 \times E_2 \rightarrow F$ where E_1, E_2 are 2 elliptic curves [1] and F is a field like F_{p^k}. An important observation before we continue is that e maps 2 *elliptic curves* to *a field*. This has a profound impact on security as we'll see shortly.

The pairing that we use has a few nice properties such as: $e(P + Q, R) = e(P, R)e(Q, R)$ and $e(aP, bQ) = e(P, Q)^{ab}$ where $a, b \in Z$. One trick to understand math is to see its implications. So, let's play with this formula a little bit. We have: $e(aP, bQ) = e(P, Q)^{ab} = e(abP, Q) = e(P, Q)^{ab} = e(bP, aQ)$. What we've just done is to move "coefficients" a, b around in 2 curves but keep the mapping result equal to $e(P, Q)^{ab}$. If you look at pairing based cryptography, you'll see that this trick is used over and over again. A cool consequence of the above trick is the following: in special case where $E_1 = E_2 = E$ and $P = Q = G$ is the base point of elliptic curve E, we have $e(aG, bG) = e(G, G)^{ab} = e(abG, G)$. I.e., given G, aG, bG we can distinguish abG from random number because we know $e(abG, G) = e(aG, bG)$ but $e(\text{random number}, G) \neq e(aG, bG)$. In other words, once we have pairing, Decisional Diffie-Hellman (DDH) problem becomes easy!

9.1 MOV attack

For elliptic curve that has pairing, Menezes, Okamoto and Vanstone (MOV) [19] found a beautiful attack against discrete log problem in elliptic curve. Recall

[1] Depending on protocols, E_1 may equal to E_2.

the discrete problem in elliptic curve E: given base point G and point A, find a such that $A = aG$.

The attack works as follows: let's assume B is another point in E that we can pair up with A. We have: $e(A, B) = e(aG, B) = e(G, B)^a$. Note that $e(A, B)$ and $e(G, B)$ are in F. Instead of solving discrete log problem in elliptic curve, we'll solve the same problem over *a field*: given $u = e(G, B)$ and $v = e(A, B)$, find a such that $v = u^a$. In general, discrete log problem over a field is easier than discrete log problem in elliptic curve, so for elliptic curve that has pairing, the security of the system reduces. I'm not saying you shouldn't use pairing, but be aware of the consequence that comes with it. There is no free lunch.

9.2 BLS signature

In 2001, Boneh, Lynn and Shacham (BLS) [20] invented a cool signature scheme based on pairing. Let's assume Alice's private key is x, her public key is $X = xG$, H is a hash function that maps messages to points on elliptic curve E. The signature is simply $\sigma = xH(m)$. To verify signature σ, we check whether $e(\sigma, G) \overset{?}{=} e(H(m), X)$. Why's that? We have $e(\sigma, G) = e(xH(m), G) = e(H(m), G)^x = e(H(m), xG) = e(H(m), X)$.

9.3 BLS signatures aggregation

Traditionally, BLS signature scheme was attractive because it's short. However, with the advance of cryptanalysis of discrete log problem in a field, cryptographers had to increase security parameters and signature's size to achieve reasonable security level. On the other hand, this signature scheme has a nice property that is still applicable today. It allows signature aggregation [21].

The basic goal of signature aggregation is the following. Let's assume we have n users, each has private key x_i, public key $X_i = x_iG$. Each user signs its own message m_i as $\sigma_i = x_iH(m_i)$. Now, in verification, instead of checking n signatures σ_i individually, we want to verify a single aggregated signature. This not only reduces CPU cycles but also saves bandwidth in transferring signatures over the network.

To achieve the previous goal, we compute an aggregated signature σ as follow: $\sigma = \sigma_1 + \cdots + \sigma_n$. To verify σ, we check whether $e(\sigma, G)) \overset{?}{=} e(H(m_1), X_1) \cdots e(H(m_n), X_n)$. Why's that? We have:

$$\begin{aligned}
e(\sigma, G) &= e(\sigma_1 + \cdots + \sigma_n, G) \\
&= e(x_1H(m_1) + \cdots + x_nH(m_n), G) \\
&= e(x_1H(m_1), G) \cdots e(x_nH(m_n), G) \\
&= e(H(m_1), G)^{x_1} \cdots e(H(m_n), G)^{x_n} \\
&= e(H(m_1), x_1G) \cdots e(H(m_n), x_nG) \\
&= e(H(m_1), X_1) \cdots e(H(m_n), X_n)
\end{aligned}$$

10 Blind signature

In 1983, when electronic payment was only in early stage, there was a great man named David Chaum who worried about users being tracked by banks, so he invented blind signature [22]. It's such a futuristic vision. Hats down to David Chaum.

Let's look at the following scenario. Assuming a bank has a private key x and public key $X = xG$ in elliptic curve E. The bank only issues tokens of fixed value, e.g. \$1 dollar. A token is a serial number together with bank's signature over that serial number. Using an issued token, a user can go to any store and show the serial number and its corresponding signature. The store who knows the bank's public key X can verify the bank's signature.

Let's pause for a moment to see what's wrong with this protocol. A user can reuse the same token with the same serial number at different stores. This is called *double-spending attack*. To mitigate this attack, the stores have to contact the bank to make sure that each serial number is only used once and the bank has a database to keep track of used serial numbers. As a consequence, the bank knows which stores users use their tokens and hence can track users' locations and spending time.

To protect users' privacy, what we need to do is to let users choose their serial numbers, mask them out, but somehow the bank can still sign them. Let m be the message/serial number to sign. We will describe a blind signature based on BLS signature by Alexandra Boldyreva [23]. Recall that in BLS signature, the signature of a message m is simply $xH(m)$.

User		**Bank**
		Private key x
		Public key $X = xG$
$r \leftarrow_{\$} \mathbb{Z}_q, R = rG, \bar{m} = H(m) + R$	$\xrightarrow{\bar{m}}$	
	$\xleftarrow{\sigma'}$	$\sigma' = x\bar{m}$
$\sigma = \sigma' - rX$		

In the above protocol, the bank knows $\bar{m}$ but has zero information about serial number m because adding a random point R to $H(m)$ completely masks out its value. The bank signs the blinded value $\bar{m} = H(m) + R$ by computing $\sigma' = x\bar{m}$. The user can extract the serial number m's signature by computing $\sigma' - rX = x(H(m) + rG) - rxG = xH(m) = \sigma$.

11 Oblivious Transfer

We'll take a look at a simple oblivious transfer (OT) protocol based on elliptic curve Diffie-Hellman protocol [24]. The problem is the following: Alice has 2 messages m_0, m_1, Bob chooses a random bit c (i.e. c is either a 0 or 1). The goal is for Bob to receive m_c without knowing what the remaining message m_{1-c} is.

Did you find the protocol? It's easy: Bob sends c to Alice and Alice sends m_c to Bob :) I intentionally left out an important security property in the problem's requirements: Alice must not know c. Now, the problem becomes difficult and challenging. Be careful when reading problem's requirements!

Let's recall our familiar Diffie-Hellman protocol. Note that in the last step, we use the shared secret k to encrypt Alice's messages m_0, m_1.

$$\begin{array}{lcl}
\textbf{Alice} & & \textbf{Bob} \\
a \hookleftarrow\!\!\scriptstyle{\$}\ \mathbb{Z}_q, A = aG & \xrightarrow{\qquad\qquad A \qquad\qquad} & \\
 & \xleftarrow{\qquad\qquad B \qquad\qquad} & b \hookleftarrow\!\!\scriptstyle{\$}\ \mathbb{Z}_q, B = bG \\
k = aB = abG & & k = bA = baG \\
 & \xrightarrow{\quad Enc_k(m_0), Enc_k(m_1) \quad} &
\end{array}$$

In this form of basic ECDH, Bob can decrypt both $Enc_k(m_0), Enc_k(m_1)$. The trick to solve OT problem is that if $c = 0$, Bob sends Alice his real public key $B = bG$, otherwise Bob sends Alice a masked key using Alice public key $B = bG + A$. Alice can't differentiate these two cases because both bG and $bG + A$ looks random to Alice, so Alice doesn't know c. When receiving B from Bob, Alice's job is to make sure that Bob can only decrypt one of two messages. Alice derives 2 keys: $k_0 = aB$ and $k_1 = aB - aA$ and sends Bob $Enc_{k_0}(m_0), Enc_{k_1}(m_1)$. The basic observation is that only one of two keys k_0, k_1 is the shared secret, while the other looks like a random number to Bob.

$$\begin{array}{lcl}
\textbf{Alice} & & \textbf{Bob} \\
a \hookleftarrow\!\!\scriptstyle{\$}\ \mathbb{Z}_q, A = aG & \xrightarrow{\qquad\qquad A \qquad\qquad} & \\
 & & b \hookleftarrow\!\!\scriptstyle{\$}\ \mathbb{Z}_q. \\
 & & \text{If } c = 0, \text{set } B = bG \\
 & & \text{If } c = 1, \text{set } B = bG + A \\
 & \xleftarrow{\qquad\qquad B \qquad\qquad} & \\
k_0 = aB, k_1 = a(B - A) & & \\
 & \xrightarrow{\quad Enc_{k_0}(m_0), Enc_{k_1}(m_1) \quad} &
\end{array}$$

If $c = 0$, $B = bG$, then $k_0 = aB = abG$ is the shared secret while $k_1 = a(B - A) = abG - aA$ looks like a random number to Bob because Bob doesn't know aA.

If $c = 1$, $B = bG + A$, then $k_1 = a(B - A) = abG$ is the shared secret while $k_0 = aB = abG + aA$ looks like a random number to Bob because Bob doesn't know aA.

To recap, Bob only knows key k_c which is the shared secret and hence can only decrypt one of two ciphertexts $Enc_{k_0}(m_0), Enc_{k_1}(m_1)$ to get the message m_c. Problem is solved.

12 Commitment schemes

Are you committed to read this article to the end? You said yes, didn't you? I don't believe you because even if you said yes, you can easily make an excuse and break your commitment. In this section, we'll take a look at cryptographic commitment schemes where it's impossible to break commitment.

A commitment scheme often consists of two phases. In the commit phase, the committer Alice commits to a value but hides the committed value. In the reveal phase, Alice reveals the committed value and we expect that Alice doesn't have capability to cheat by revealing a value that she hasn't committed to.

12.1 Hash based commitment scheme

We'll use a cryptographic hash function h (e.g. SHA256) for our commitment scheme. To commitment to a value v, Alice randomly generates a large fixed size number r(e.g. 128-bits) and publishes $c = h(r||v)$. To open the commitment, Alice reveals r and v and everyone can verify whether $h(r||v) \overset{?}{=} c$.

Why can't we use $c = h(v)$? The main issue is that this leaks v because an attacker can brute-force v, computes $h(v)$ and stops until the result matches c. The role of r is to increase randomness so that guessing $r||v$ is an impossible task.

Why can't Alice cheat by opening to a different value rather than v? To open to a different value v', Alice has to find v', r' such that $h(r'||v') = c = h(r||v)$. In other words, Alice has to find a collision for h. For standard cryptographic hash function such as SHA256, this is an infeasible task.

12.2 Pedersen commitment scheme

Pedersen commitment scheme [25] is based on discrete log problem. Let G and H be two points in elliptic curve E and no one knows $\log_G(H)$. To commit to a value v, Alice generates a random r and publishes $c = vG + rH$. To open the commitment, Alice reveals r and v.

Why does c hide v? The reason is that rH is random point in E, so no matter what vG is, adding rH to it completely masks out its value.

Why can't Alice cheat by revealing a different value v'? To achieve that goal Alice has to find v', r' such that $c = vG + rH = v'G + r'H$. This implies $(v - v')G = (r' - r)H$ or $(v - v')/(r' - r)G = H$. This means that the committer knows $\log_G(H) = (v - v')/(r' - r)$. This contradicts with the assumption that no one knows $\log_G(H)$. In other words, don't use any Pedersen's commitment scheme if you aren't convinced that no one knows $\log_G(H)$ because knowing $\log_G(H)$ allows the committer to cheat.

12.2.1 Homomorphic property

Pedersen commitment scheme has a nice homomorphic property: given 2 commitments c_1, c_2, even though we don't know their committed values, we still can compute another commitment $c = c_1 + c_2$ that corresponds to committed values' sum. It's straightforward to check whether the previous statement is true: if $c_1 = v_1G + r_1H, c_2 = v_2G + r_2H$ are commitments of v_1, v_2 then $c = c_1 + c_2 = (v_1 + v_2)G + (r_1 + r_2)H$ is the commitment of $v = v_1 + v_2$.

13 Lattice based cryptography

In the last ten years, lattice based cryptography[26], [4] blossoms. There are two main reasons. The first reason is that lattice based cryptography is assumed to be safe against quantum computer, while ECDH key exchange and RSA aren't. As quantum computer is more powerful than classical computer, can we deduce that lattice based cryptography is safe against classical computer? No, we can't. It's because we start with a security assumption (not fact) which might turn out to be wrong. Like many other things in cryptography, lattice based cryptography's security is an assumption, no one knows for sure. Furthermore, there is no evidence that lattice based cryptography is safer than ECDH against classical computer. Therefore, it's not wise to rush into lattice based cryptography and abandon classical protocols such as ECDH key exchange. The second reason is that with lattice based cryptography, cryptographers can design cryptographic protocols with extraordinary properties that have never been achieved. The most famous application of lattice based cryptography is fully homomorphic encryption (FHE).

We'll describe lattice based cryptography without even defining what lattice is because the math behind lattice theory is way over my head. Instead, we'll talk about familiar polynomials which have lattice structure. By the way, do you notice a pattern in this article? Whenever I see a complicated thing, I run away because I can't deal with complexity!

Lattice based cryptography are often rely on the hardness of finding something *small* or solving equations with *error*. Every time, you see the words small and error in the context of polynomials, you know that you land in the realms of lattice based cryptography. In this section, we'll use polynomials in $\mathbb{Z}_q[x]/(P(x))$ or $\mathbb{Z}[x]/(P(x))$, i.e., polynomials whose coefficients are in $\mathbb{Z}_q$ (or $\mathbb{Z}$) and all operations are $\mod P(x)$. There are 2 problems that are infeasible to solve in $\mathbb{Z}_q[x]/(P(x))$. We'll briefly introduce them without further discussion. The main goal is to introduce a few terminologies and to give you high level idea what hard problems that we depend on.

The first problem is called Ring-Short Integer Solution (R-SIS): given m polynomials a_i in $\mathbb{Z}_q[x]/(P(x))$, find m *small* polynomials z_i in $\mathbb{Z}[x]/(P(x))$ such that $\sum_i a_i.z_i = 0$. Note that without "small" requirement, the problem is easy to solve.

The second problem is called Ring-Learning With Error (R-LWE): given a secret s in $\mathbb{Z}_q[x]/(P(x))$, m random polynomials a_i in $\mathbb{Z}_q[x]/(P(x))$ and m small "error" polynomials e_i, distinguish $(a_i, b_i = a_i.s + e_i)$ from random pair of polynomials. Note that without "error", the problem is trivial to solve.

13.1 Lattice based homomorphic encryption

Let's assume we have data that we store in a cloud. To protect our data, we encrypt them and keep the key to ourselves. On the other hand, we want to take advantage of cloud's computing power, so we want the cloud to compute on our ciphertexts without knowing what our plaintexts are. Homomorphic encryption is special type of encryption that achieves the previous goal. In this section, we'll describe a simple lattice based homomorphic encryption [27].

We'll use $\mathbb{Z}_q[x]/(x^{2^k} + 1)$ (q is a prime number), i.e., polynomials whose coefficients are in $\mathbb{Z}_q$ and all operations are $\mod x^{2^k} + 1$. We also use a small modulus t that is much smaller than q. Note that every thing in this section including secret key, message, ciphertext are polynomials.

The secret key is a polynomial s in $\mathbb{Z}_q[x]/x^{2^k} + 1)$.

To encrypt a message polynomial $m \in \mathbb{Z}_t[x]/(x^{2^k}+1)$, we randomly generate polynomial a, small error polynomial e and the ciphertext is simply $c = Enc(m) = (c_0, c_1) = (-a, as + m + et)$.

To decrypt a ciphertext $c = (c_0, c_1)$, we compute $c_1 + c_0 s \mod t = as + m + et - as \mod t = m + et \mod t = m$.

To see how this encryption is additive homomorphic, let's take a look at two encryptions of m and m': $c = Enc(m) = (c_0, c_1) = (-a, as + m + et)$ and $c' = Enc(m') = (c'_0, c'_1) = (-a', a's + m' + e't)$. If we add (c_0, c_1) and (c'_0, c'_1) together, we have:

$$\begin{aligned}(c_0, c_1) + (c'_0, c'_1) &= (c_0 + c'_0, c_1 + c'_1) \\ &= (-a - a', as + m + et + a's + m' + e't) \\ &= (-(a + a'), (a + a')s + (m + m') + (e + e')t\end{aligned}$$

If we denote $a'' = a + a'$, $m'' = m + m'$, $e'' = e + e'$, then we see that $c'' = (c''_0, c''_1) = (c_0, c_1) + (c'_0, c'_1)$ is the encryption of $m'' = m + m'$ with error $e'' = e + e'$. To recap, what we've done is to add 2 ciphertexts together without knowing the messages, but the result corresponds to the sum of the messages. It's pretty cool, right?

Another nice property is that if you multiply a polynomial p to the encryption of m then the result corresponds to encryption of pm. To see why it's the case, let's take a look at polynomial p and encryption of m: $(c_0, c_1) = (-a, as + m + et)$. We have:

$$\begin{aligned}p(c_0, c_1) &= (pc_0, pc_1) \\ &= (p(-a), p(as + m + et)) \\ &= (-pa, pas + pm + pet)\end{aligned}$$

If we denote $a' = -pa$, $m' = pm$, $e' = pe$, then we see that $(c'_0, c'_1) = p(c_0, c_1)$ is the encryption of $m' = pm$ with error $e' = pe$.

The final note is that the error increases in both cases. For lattice based cryptography to work, the error must be small. Therefore, various techniques have been designed to reduce the error over time. We won't discuss error reduction techniques here, instead, we'll take a look at an awesome application of the previous homomorphic encryption.

13.2 Private information retrieval

Let's say a server has a public database (e.g. movies, songs, lyrics, stories, books) with n items $x_1, \cdots, x_n$. A user wants to see a single item x_i at index i from the database without revealing to the server what item has been downloaded. This is to protect user's privacy. An obvious solution is the user downloads all n items from the database. This has perfect privacy, but it costs significant bandwidth and user's local storage. We'll trade CPU with bandwidth and storage using the above homomorphic encryption [28]. The basic protocol works as follows.

The user forms a sequence of 0 and 1 where only at index i, it's 1 while the remaining numbers are 0: $0, \cdots, 0, \underbrace{1}_{\text{index i}}, 0, \cdots, 0$. The user uses homomorphic encryption to encrypt the above sequence, i.e., $c_1 = Enc(0), \cdots, c_{i-1} = Enc(0), c_i = Enc(1), c_{i+1} = Enc(0), \cdots, c_n = Enc(0)$. The user sends $c_1, \cdots, c_n$ to the server.

The server computes $x = x_1 c_1 + \cdots + x_n c_n$ without knowing what i is and sends x to the user.

The user decrypts x and the result is x_i. Why's that? By homomorphic property, $x = x_1 c_1 + \cdots + x_n c_n$ corresponds to the encryption of $x_1.0 + \cdots + x_{i-1}.0 + x_i.1 + x_{i+1}.0 + \cdots + x_n.1 = 0 + \cdots + 0 + x_i + 0 + \cdots + 0 = x_i$.

14 Conclusion

I can't believe that you finish reading this article. I adore your patience, thank you! By the way, if you cheated and jumped directly to the conclusion section, then go back and read the article :)

My hope is you can see the beauty of advanced cryptographic protocols before running away because of their difficulties. And if we're lucky, you will use these fantastic cryptographic protocols in your applications. In that case, don't forget to read nice cryptography engineering blog [29] and serious book about applied cryptographic protocols [15]. On the other hand, if you reach this section and you have no idea what these cryptographic protocols are about then I failed spectacularly :(I'll try better next time :)

References

[1] Victor Shoup. *A Computational Introduction to Number Theory and Algebra.*

[2] Thomas W. Judson. *Abstract Algebra: Theory and Applications.*

[3] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography.*

[4] Chris Peikert. A decade of lattice cryptography.

[5] Peter Shor. Algorithms for quantum computation: discrete logarithms and factoring.

[6] Lov K. Grover. A fast quantum mechanical algorithm for database search.

[7] Whitfield Diffie and Martin E. Hellman. New directions in cryptography.

[8] Taher ElGamal. A public key cryptosystem and signature scheme based on discrete logarithms.

[9] Neal Koblitz. Elliptic curve cryptosystems.

[10] C.P. Schnorr. Efficient signature generation by smart cards.

[11] David Chaum and Torben Pryds Pedersen. Wallet databases with observers.

[12] Jim Miller. Coordinated disclosure of vulnerabilities affecting girault, bulletproofs, and plonk. https://blog.trailofbits.com/2022/04/13/part-1-coordinated-disclosure-of-vulnerabilities-affecting-girault-bulletproofs-and-plonk/.

[13] C.P. Schnorr. Efficient signature generation by smart cards.

[14] Ronald L. Rivest, Adi Shamir, , and Yael Tauman. How to leak a secret.

[15] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography.*

[16] Ronald Cramer, Ivan Damgard, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols.

[17] Adi Shamir. How to share a secret.

[18] Ben Lynn. `https://crypto.stanford.edu/pbc/notes/elliptic/`.

[19] Afred Menezes, Scott Vanstone, and Tatsuaki Okamoto. Reducing elliptic curve logarithms to logarithms in a finite field.

[20] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing.

[21] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps.

[22] David Chaum. Blind signatures for untraceable payments.

[23] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme.

[24] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer.

[25] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing.

[26] The 2nd biu winter school. `https://cyber.biu.ac.il/event/the-2nd-biu-winter-school/`.

[27] Simple homomorphic encryption library with lattices (shell) (https://github.com/google/shell-encryption).

[28] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir : Private information retrieval for everyone.

[29] Matthew Green. A few thoughts on cryptographic engineering (https://blog.cryptographyengineering.com/).