Notes From a Student

Jonathan Weiss

CHAPTER 1

Intro-and todos

- break computer-science file into part files
- These notes are a collection of information I've gathered regarding distributed systems, cryptography, and math. Mainly intended to be a source of knowledge to look back at and get quickly into the notions. Some of the items will have sources, and some will simply be proved here.

Contents

Part 1

Algebraic structures

This section was motivated by a theorem for polynomial identity, which is used in many cryptography applications:

LEMMA 1.0.1 (DeMillo-Lipton-Schwartz-Zippel lemma). *Let $p \in \mathbb{F}[x_1, x_2, x_3, \ldots, x_n]$ be a non-zero polynomial of a total degree $d \geq 0$ over a field $\mathbb{F}$. Let S be a finite set of $\mathbb{F}$ and let $(r_1, r_2, \ldots, r_n)$ be selected at random,* independently and uniformly *from S. Then*

$$\Pr\left[p(r_1, r_2, \ldots, r_n) = 0\right] \leq \frac{d}{|S|}$$

Unfortunately, this doesn't work over quotient polynomial rings (e.g., instead of $\mathbb{F}[x]$, we have $\mathbb{F}[x]/(x^n + 1)$).

We'll start learning Algebraic structures in hopes we gain enough insight regarding **??**.

CHAPTER 2

Group theory

2.1. Group

DEFINITION 2.1.1. *A* group *is a set G with a binary operation $\cdot : G \times G \to G$ that satisfies the following constraints:*

- *the operation holds closure: if $x, y \in G$ so is $x \cdot y \in G$.*
- *the operation is associative: $\forall x, y, z \in G$ it holds that $(x \cdot y) \cdot z = x \cdot (y \cdot z)$.*
- *has an identity element: there exists $e \in G$ such that $\forall x \in G x \cdot e = x = e \cdot x$*
- *Every element $x \in G$ has an inverse $x^{-1} \in G$ such that $x \cdot x^{-1} = e = x^{-1} \cdot x$.*

REMARK 2.1.2. *Note that a group can be an additive group or a multiplicative group.*

EXAMPLE 2.1.3. *Assume $\mathbb{F}* = F$
0, $(\mathbb{F}*, \cdot, 1)$ is a valid multiplicative group but, $(\mathbb{F}, \cdot, 1)$ is not because there is no inverse to 0 in $\mathbb{F}$.*

DEFINITION 2.1.4. *group G is called* commutative*, or* abelian *if for every $x, y \in G$ $x \cdot y = y \cdot x$ holds.*

2.1.1. traits of groups. let $(G, \cdot, e)$ be a group, thus

- e is unique (like in fields).
- the inverse of an element is unique.
- $\forall x \in G (x^{-1})^{-1} = x$.
- $\forall x, y \in G (x \cdot y)^{-1} = y^{-1} \cdot x^{-1}$

DEFINITION 2.1.5. *denote $|G|$ as the* order *of a group - its size.*

DEFINITION 2.1.6. *denote the order of $x \in G$ to be the minimal $m! = 0 \in \mathbb{N}$ such that $x^m = e$. if there is no such m, then the order of x is infinite.*

THEOREM 2.1.6.1. *Let there a finite group G, thus $\forall x \in G$ has a finite order.*

THEOREM 2.1.6.2. *Let G be a group, and let $x \in G$ such that $order(x) = n$, thus $x^m = e$ iff $n|m$.*

THEOREM 2.1.6.3 (Fermat's little theorem). *let there be x and p prime, thus $x^p \equiv x \mod p$ (and as a result $x^{p-1} \equiv emodp$).*

2.1.2. Subgroups.

DEFINITION 2.1.7. *Let there be a group G, $\emptyset \neq H \subseteq G$ is a* subgroup *if it is a group with respect to the same operation of G.*

usually, we denote a subgroup with $H \leq G$.

2.1.3. cosets.

Part 2

Computer-Science

CHAPTER 3

Distributed Algorithms

Many thanks to the Decentralized thoughts blog.

cheat sheet: https://decentralizedthoughts.github.io/2021-10-29-consensus-cheat-sheet/.

3.1. Basic Foundations and Classics

3.1.1. Consensus and Agreement. In this section, we define the consensus problem and discuss some variants and their differences.

Let us begin with the most straightforward consensus problem: agreement.

3.1.1.1. *The agreement problem.*

We assume n nodes, where each node i has some input $v_i \in V$ where V is the set of all possible values. A protocol that solves Agreement must have the following properties:

(1) (*Agreement*): no two honest nodes **decide** on different values.
(2) (*validity*): if all honest nodes have the same input value v, then v must be the decision value.
(3) (*termination*): all honest nodes must decide on a value from V and terminate.

- This is a strong property. that is, validity guarantees that a decisional value must be held by all honest nodes.
- We can make the agreement property even stronger - no two nodes (notice I've omitted the honest?) decide on different values.

We continue to another consensus problem tied strongly with the agreement problem.

3.1.1.2. *Broadcast problem.* We assume a designated node, the leader, holds some value $v \in V$. A protocol that solves Broadcast must comply with the following properties:

(1) (*Agreement*): no two honest nodes **decide** on different values.
(2) (*validity*): if the leader is honest then v must be the decision value.
(3) (*termination*): all honest nodes must decide on a value from V and terminate.

Exercise 3.1.1. *Implement Broadcast given a black box Agreement protocol.*

Exercise 3.1.2.

My Assumptions: the leader is known. Can communicate privately on a secure channel with each server.

(1) *first round; the leader chose a value and send it to all.*
    (a) *an honest machine would only accept a value from the leader, and once.*

(2) *second round; every honest node should hold the same value as the leader, so they can now start to run the agreement protocol.*
(3) *end of the agreement should mean all had a decision, and because all held the value v that the leader sent, this is the agreement value.*

proving the properties is immediate from definitions. Termination from the agreement protocol. Agreement from agreement protocol. Validity is because the leader is honest and did not send different values to the servers, and then the agreement kicked off with the same input values.

EXERCISE 3.1.3. *Implement Agreement using broadcast.*
Note, in this exercise you start with n leaders. how would you make an agreement protocol?

3.2. Synchrony, Asynchrony, and Partial synchrony

In the standard model, communication uncertainty is captured by an adversary that can delay messages. The three basic communication models are:

synchronous: There exists some known finite time bound Δ. For any message sent, the adversary can delay its delivery by at most Δ.

Asynchronous: for any message sent, the adversary can delay its delivery by any finite amount of time. So, on the one hand, there is no bound on the time to deliver a message, but each message must eventually be delivered.

partial–synchrony: This is a combination of the two. In this model, we assume that there exists some known finite time Δ, and a special event called GST (global stabilization time) such that:

- the adversary must cause the GST event to happen after some unknown finite time.
- any message that is sent at time x must be delivered by time $\Delta + \max(x, GST)$.

Another definition: Partial synchrony is to assume that there is some finite unknown upper bound Δ on message delivery. This bound is not known in advance and can be chosen by the adversary.

Partial synchrony captures the intuition that we would like to design protocols for systems that are usually synchronous but have reasonable guarantees, even if the synchrony assumptions become temporarily violated by some extreme event (like a denial of service attack). In particular, a recurring theme in the Partial synchrony model is to design protocols that are always safe (even when the system is asynchronous) but provide liveness and termination guarantees only after GST (only when the system is synchronous).

3.3. The Threshold Adversary

If the adversary has no limits, then there is very little we can do. The simplest model is that of a threshold adversary. Given n nodes, the adversary controls some f nodes.

(1) $n > f$, the adversary controls all but one.
(2) $n > 2f$, the adversary controls a minority of nodes, often called the dishonest minority.
(3) $n > 3f$, where the adversary controls less than a third of the nodes.

3.4. Generalized bounded resource threshold adversary

The adversary control some fraction of the resources.

(1) the adversary controls any fraction of the resource, but not all.
(2) the adversary controls $\frac{1}{2} - \epsilon$ fraction of the resource.
(3) the adversary controls less than $\frac{1}{3}$ fraction of the resource.

Some common examples:

(1) in Nakamoto consensus, the adversary has access to $\frac{1}{2} - \epsilon$ fraction of CPU power.
(2) in systems that use proof-of-stake, the assumption is that the bounded resource is some finite set of coins. It is then natural to assume that the adversary controls a threshold of the total coins. can often map voting power based on the relative amount of coins at a given time. For example, Algorand mentions the total voting power of the adversary is bounded by a third.

3.4.1. A note on stake-based bounded resources.

(1) On the one hand, using a resource controlled by the platform allows controlling the resource allocation in ways that are not possible when the resource is external. In particular, the platform can create punishment mechanisms to incentivize honest behavior better. For example, Buterin suggests conditions to detect malicious behavior and punish the adversary by reducing (slashing) the offender's stake.
(2) n the other hand, if the value of the stake depends on the platform, this may cause some circular reasoning about security and in particular a bootstrapping problem. If the value of the total coins is too high, then it may happen that not enough honest entities will sign up and the system will lose liveness. If the value of the total coins is initially too low, then an attacker can gain early monopoly power. One option is to bootstrap proof-of-stake using an existing decentralized high-value Proof-of-work coin or high-value traditional fiat (see here). This type of solution assumes that the adversary does not already have monopoly power on the bootstrapping resource. Another option is to bootstrap a Proof-of-stake system from an existing high-value Proof-of-work system (for example, see eth2.0).

3.5. The power of the adversary

We still need to make important modeling decisions about the adversary power. In addition to the size of the threshold ($n > f, n > 2f, \text{or } n > 3f$), there are 4 more important parameters:

(1) The type of corruption.
(2) The computational power of the adversary.
(3) The visibility of the adversary. (TBD).
(4) The adaptivity of the adversary.

3.5.1. Type of corruption.

There are four classic adversaries: Passive, Crash, Omission, and Byzantine.

Passive: Also known as Honest-but-curios. Will follow the protocol in earnest, but allows the adversary to learn any information in its *view*.

Crash: Once corrupted, it'll stop sending or receiving messages.

Omission: Does not deviate from the protocol, but will lose or wont send messages according to the adversaries will.

Byzantine: this gives the adversary full power to control the party and take any (arbitrary) action on the corrupted party.

REMARK 3.5.1. *Each level of corruption, subsumes the previous level.*

REMARK 3.5.2. *There are more types of corruptions, which we'll investigate later.*

3.5.2. Computational power.

Unbounded: the adversary has unbounded computational power. This model often leads to notions of perfect security or statistical security.

Computationally-bounded: the adversary is at most a polynomial advantage in computational power over the honest parties. Typically this means that the adversary cannot (except with negligible probability) break the cryptographic primitives being used. For example, typically assume the adversary cannot forge signatures of parties not in its control.

Fine-grained computationally–bounded: there is some concrete measure of computational power and the adversary is limited in a concrete manner. This model is used in proof-of-work based protocols.

3.5.3. Visibility.

The visibility is the power of the adversary to see the messages and the states of the non-corrupted parties. Again, there are two basic variant:

Full information: here we assume the adversary sees the internal state of all parties and the content of all message sent. This often limits the protocol designer. See for example: Feige's selection protocols, or Ben-Or et al's Byzantine agreement.

Private channels: in this model, we assume the adversary cannot see the internal state of honest parties and cannot see the internal content of messages between honest parties. The adversary does know when a message is being sent and depending on the communication model can decide to delay it by any value that is allowed by the communication model.

In round based protocols:

Rush: the adversary can see all messages in a round before choosing what to send.

non-Rush: the adversary must commit to the round i messages it sends before it receives any round i messages from non-faulty parties.

3.5.4. Adaptivity.

Adaptivity is the ability of the adversary to corrupt dynamically based on information the adversary learns during the execution. There are three basic variants: static, adaptive, and mobile. The adaptive model has several sub-variant. We will cover here only the simplest one.

Static: the adversary has to decide which f parties to corrupt before executing the protocol.

Adaptive: the adversary can decide dynamically as the protocol progresses who to corrupt based on what the adversary learns over time. The main parameter still needs to be decided on how long it takes between the adversary's decision to corrupt and the event that the control is passed to the adversary. One standard assumption is that this is instantaneous. Another is that it takes an additional round.

Mobile: the adversary can decide dynamically who to corrupt and who to un-corrupt. The number of corrupted parties at any given time is at most f, but the set of corrupted parties may change over time. It is often required that there is a gap between the time the adversary corrupts one party and the time it is allowed to corrupt another.

3.6. The Trusted Setup Phase

When you want to understand a decentralized system, you first need to ask whether it has a trusted setup phase.

Many distributed computing and cryptography protocols require a trusted setup. A trusted setup is a special case of a multi-phase protocol. We call the first **phase** the setup phase and the second phase the main phase. Two properties often distinguish a setup phase from the **main phase**:

- Typically, the main phase implements some repeated task. The setup phase is done once and enables repeating many instances of the main phase.
- The setup phase is often input-independent. Namely, it does not use the private inputs of the parties. Furthermore, sometimes the setup phase is even function-independent, meaning that the specific function that the parties wish to compute is irrelevant; the parties at this phase only know that they want to compute some function. As such, the setup and main phases are often called offline (or preprocessing) and online respectively (i.e., parties may run the offline phase when they realize that at some later point in time, they will want to run some function on inputs they do not know yet).

think of a setup phase as an ideal functionality run by a wholly trusted entity, denoted T. For instance, assuming Public-Key Infrastructure (PKI) means that we assume there is a wholly trusted entity to which every party submits its public encryption (and verification) key, and that entity broadcasts those keys to all parties. In this chapter, we will review some of the common types of trusted setup assumptions by looking at their ideal functionalities.

One way to model that trusted entity follows: There is an initial set of parties $p_1, \ldots, p_n$ who interact with the trusted entity T. The parties may send inputs $x_1, \ldots, x_n$ to T (where x_i is p_i's input), who in turn, runs some function $f(r, x_1, \ldots, x_n)$ where r is a uniformly random string, obtains outputs $y_1, \ldots, y_n$ and hands y_i to p_i. This process may be "reactive", namely, it may repeat multiple times. As this already describes an idealized world, we always assume that the communication channels between the parties and the trusted entity are secure.

we argue that most of those functionalities fall into one of out of the five categories below:

(1) No Setup: This is the simplest case, in which we don't really use any trusted entity or any trusted setup. The minimal communication assumption is that parties have access to some type of communication medium. Often, though, "no setup" also refers to a setting where parties' identities are globally known.
(2) Pairwise setup: Here we assume there is some set of initial parties $p_1, \ldots, p_n$ and each two parties have a reliable communication channel between them. In particular, in the simplest pairwise setup assumption when party p_i receives a message on the (i, j) channel it knows that party p_j sent this message.
(3) Broadcast setup: We assume a setup whose implementation requires no secrets. The canonical example is a PKI setup that requires broadcast only to relay the public keys.
(4) Partially public setup: often called the Common Reference String (CRS) model. Many cryptographic protocols leverage this setup for improved efficiency. A special case of this setup is a randomness beacon.
(5) Fully private setup: often called the offline phase in the context of secure multiparty computation (MPC) protocols. Here, the setup phase computes a rather complex output that is party-dependant.

Let us detail these five setup variants.

3.6.1. No Setup. If a protocol has no setup, then there is nothing to worry about. It's easier to trust such protocols. On the other hand, there are inherent limitations.

A setting with no setup has two main flavors. The first assumes really nothing on the knowledge of the parties and is sometimes called anonymous channel model The second assumes global knowledge of the identities of all parties, which is quite acceptable in many real world applications. Both flavors suffer from non-authenticated channels, meaning that the adversary can launch man-in-the-middle attacks arbitrarily.

In traditional cryptography (where the adversary is polynomially-bounded), this type of model was first studied by Dolev, Dwork and Naor, for specific tasks like non-malleable encryption and zero-knowledge and later generalized to arbitrary computations by Barak et al. (The latter assumes global identities.)

Another line of research, in the anonymous model, is based on a more refined assumption on the adversarial power. Namely, the assumption limits the computational power of the adversary (e.g., hash rate) compared to the computational power of the honest parties. It was shown possible to construct a limited notion of PKI from scratch even in this slim model.

3.6.2. Pairwise Setup. Here, we assume that the communication channel between every pair of parties is authenticated. This is a classic assumption in distributed cryptography and distributed computing. The Fisher, Lynch and Merritt 1985 lower bounds show that even weak forms of Byzantine Agreement are impossible when $n \leq 3 \cdot f$ even given this setup, and even against a traditional polynomially-bounded adversary.

For $n > 3 \cdot f$, on the other hand, this setup allows perfect implementation of any functionality. This is the celebrated result of Ben-Or, Goldwasser and Widgerson 1988 (see Asharov and Lindell for a full proof).

3.6.3. TODO: complete from this blogm page: https://decentralizedthoughts.github.io/2019-07-19-setup-assumptions/

3.7. Broadcast

todo: https://decentralizedthoughts.github.io/2019-10-22-flavours-of-broadcast/

3.8. Sync HotStuff

A Simple and Practical State Machine Replication.

CHAPTER 4

Elliptic curves

Taken from *A Graduate Course in Applied Cryptography, Dan Boneh and Victor Shoup*

The best-known discrete log algorithm in an elliptic curve group of size q runs in time $O(\sqrt{q})$. This means that to provide security comparable to AES-128, it suffices to use a group of size $q \approx 2^{256}$ so that the time to compute discrete log is $\sqrt{q} \approx 2^{128}$.

4.1. a bit of history on elliptic curves

Elliptic curves come up naturally in several branches of mathematics. Here we will follow their development as a branch of arithmetic (the study of rational numbers). Diophantus, a greek mathematician who lived in Alexandria in the third century, was interested in the following problem: Given a bivariate polynomial equation, $f(x, y) = 0$ find rational points satisfying the equation. That is, find $P = (x, y)$ such that $x, y \in \mathbb{Q}$. Diophantus wrote a series of influential books on this subject, called the *Arithmetica*, of which six survived. Much of Arithmathica studies integer and rational solutions of quadratic equations. For example, one of the first problems in the books regarding elliptic curves is to find $(x, y) \in \mathbb{Q}$ that satisfy the equation

$$y^2 = x^3 - x + 9$$

The method invented to answer this question secures most Internet traffic worldwide. One can easily verify that the following points are on the curve $(0, \pm 3), (1, \pm 3), (-1, \pm 3)$. Diophantus developed a method similar to chord method: Given two rational points U and V on the curve, where $U \neq -V$, we can pass a line through them, and this line must intersect the curve at a third rational point W. The chord method was rediscovered several times over the centuries, but it finally stuck with the work of Poincare.

Poincare defines the sum of U, V, denoted $U \boxplus V$ as $U \boxplus V := -W$ Diophantus' method, the *tangent method* is another way to build a new rational point from a given rational point. We should later see that the method corresponds to adding the point P to itself: $P \boxplus P$.

4.2. elliptic curves over finite fields

We are primarily interested in elliptic curves over finite fields for cryptographic applications.

DEFINITION 4.2.1. *Let $p > 3$ be a prime. An **elliptic curve** E defined over $\mathbb{F}_p$ is an equation*

$$y^2 = x^3 + ax + b$$

where $a, b \in \mathbb{F}_p$ satisfy $4a^3 + 27b^2 \neq 0$. We write $E/\mathbb{F}_p$ to denote the fact that E is defined over $\mathbb{F}_p$.

The weird condition $4a^3 + 27b^2 \neq 0$ ensures that the equation $x^3 + ax + b = 0$ does not have a double root (needed to avoid degeneracies).

DEFINITION 4.2.2. *We say that point $P = (x, y)$, where $x, y \in \mathbb{F}_p$, is on curve E if it satisfies the equation of E.*

DEFINITION 4.2.3. *$E(\mathbb{F})$ denotes the set of all points on the curve E and are defined over $\mathbb{F}$.*

The addition law, and adding to the set $E(F_p)$ the member $\mathcal{O}$, which servers as the identity member, turns it into a group. I'm not getting into the addition laws.

4.3. Elliptic curve cryptography

Given two points P and $\alpha \cdot P$, where $\alpha \in \mathbb{Z}_q$, it is hard to compute α. The best known algorithm to compute α is $\Omega(\sqrt{q})$. There are a few exceptions to these rules, and thus one should be careful when defining a new curve. So it's better to use some specific already implemented and used curve rather than generating a random prime p and a random curve over $\mathbb{F}_p$.

4.4. Pairing based cryptography

We now show that certain elliptic curves have an additional structure, called a pairing, which enables a world of new schemes that could not be built from discrete log groups without this other structure.

DEFINITION 4.4.1. *Let $\mathbb{G}_0, \mathbb{G}_1, \mathbb{G}_T$ be three cyclic groups of prime order q where $g_0 \in \mathbb{G}_0$ and $g_1 \in \mathbb{G}_1$ are generators. A **pairing** is an efficiently computable function $e : \mathbb{G}_0 \times \mathbb{G}_1 \to \mathbb{G}_T$ satisfying the following properties:*

- *bilinear: $\forall u, u' \in \mathbb{G}_0, v, v' \in \mathbb{G}_1$ we have*

$$e(u \cdot u', v) = e(u, v) \cdot e(u', v) and e(u, v \cdot v') = e(u, v) \cdot e(u, v')$$

- *non-degenerate: $g_T := e(g_0, g_1)$ is a generator of $\mathbb{G}_T$.*

We refer to $\mathbb{G}_0, \mathbb{G}_1$ as the pairing groups and $\mathbb{G}_T$ as the target group.

Bilinearity implies the following central property of pairing that will be used in all our constructions: for all $a, b \in \mathbb{Z}_q$ we have

$$e(g_0^a, g_1^b) = e(g_0, g_1)^{a \cdot b} = e(g_0^b, g_1^a)$$

This equality follows from the equalities $e(g_0^a, g_1^b) = e(g_0, g_1^b)^a = e(g_0^a, g_1^b) = e(g_0^a, g_1)^b$ which are themselves a direct consequence of bilinearity (Note that in cyclic groups such as $\mathbb{G}_0 and \mathbb{G}_1$ this definition is equivalent to the definition of bilinearity above). [if I understand correctly, due to $\mathbb{G}_0 and \mathbb{G}_1$ having generators, we can represent every number as a power of the generators themselves, and using bilinearity we can reach the above equations]

In a symmetric pairing group, where $G_0 = G_1$ schemes like ElGamal aren't secure anymore, that is, given two public keys, it is to break the DDH assumption:

$$e(g^x, g^y) = e(g, g)^{x \cdot y}$$

Hence, it is possible to extract information.

In asymmetric pairing, where $G_0 \neq G_1$, it may still be possible that the DDH assumption holds in both G_0 and G_1.

Furthermore, if the discrete log in G_T is easy, it is easy in G_0 (and G_1).

REMARK 4.4.2. *The pairing function e on an elliptic curve comes from an algebraic pairing called the* ***Weil pairing****. Which can be efficiently evaluated. In practice, one uses variants of the weil pairing, called the Tate and Ate pairings.*

We'll avoid $bn256$ curve as it isn't secure.

optimisations.

- When the first value of the pairing is known, we can speed things up by calculating a table of the other possible values. (though it seems very inefficient, and memory intensive.)

The BLS signature scheme

Let $e : F_0 \times G_1 \to G_T$ be a pairing where G_0, G_1, G_T are cyclic groups of prime order q, and where $g_0 \in G_0, g_1 \in G_1$ are generators. We'll also require a hash function $H : \mathcal{M} \to G_0$.

The BLS signature scheme denoted $\mathcal{S}_B LS = (G, S, V)$, has message space $\mathcal{M}$ and works as follows:

- $G()$: The key generation algorithm runs as follows

  $$\alpha \leftarrow \mathbb{Z}_q, u \leftarrow g_1^\alpha \in G_1$$

  The public key is $pk := u$, and the secret key is α.
- $S(sk, m)$: To sign a message $m \in \mathcal{M}$ using a secret key $sk = \alpha \in \mathbb{Z}_q$ do:

  $$\sigma \leftarrow H(m)^\alpha \text{ output } \sigma$$

- $V(pk, m, \sigma)$: To verify a signature

  $$e(H(m), u) = e(\sigma, g_1)$$

  that is, the signature is accepted due to the following calculations

  $$e(H(m), g_1^\alpha) = e(H(m), g_1)^\alpha = e(H(m)^\alpha, g_1) = e(\sigma, g_1)$$

Advanced encryption schemes from pairings

Identity based encryption. Using Identity-Based Encryption (IBE from now), Alice can send an encrypted message to Bob without asking for bobs key beforehand. Beyond key exchange, IBE can also be used to construct CPA-secure public key encryption and even search encrypted data. In an IBE scheme, a trusted entity, Trudy, generates a master key pair (mpk, msk). The key mpk is the *master public key* and is known to everyone. Trudy keeps the *master secret key* msk to herself. When Bob wants to use his email address as his public key, he must somehow obtain the private key derived from msk and Bob's public key. We denote Bob's email address as his (public) identity id and denote the corresponding private key by sk_{id}. He receives sk_{id} by contacting Trudy, she'll use msk and id to generate sk_{id}, and send it back to Bob. Now anyone, including Alice, can send bob messages without looking up his public key. Assuming, of course, that Alice knows mpk.

DEFINITION 4.4.3. *An **Identiy based encryption scheme** $\mathcal{E}_id = (S, G, E, D)$ is a tuple of four efficient algorithms: a **setup algorithm** S, a **Key generation algorithm** G, an **encryption algorithm** E and a **decryption algorithm** D.*

- *S is a probabilistic algorithm which outputs $(mpk, msk) \xleftarrow{R} S()$.*
- *G is a probabilistic algorithm invoked as $sk_{id} \xleftarrow{R} G(msk, id)$.*
- *E is a* probabilistic *algorithm invoked as $c \xleftarrow{R} E(mpk, id, m)$.*
- *D is a deterministic algorithm invoked as $m \xleftarrow{R} D(sk_{id}, c)$. Where m is either a message or a **reject** value.*
- *We require that decryption undoes encryption:*

$$\Pr[D(G(msk, id), E(mpk, id, m)) = m] = 1$$

- *identities, messages, and ciphertexts are in their respective finite spaces, and $\mathcal{E}_id$ is defined over $(\mathcal{ID}, \mathcal{M}, \mathcal{C})$.*

We can view IBE as a particular public key encryption scheme where the messages to be encrypted pairs $(id, m) \in \mathcal{ID} \times \mathcal{M}$. The master secret key can decrypt any well-formed ciphertext. However, there are weaker secret keys, such as sk_{id} that can decrypt a ciphertext $c \xleftarrow{R} E(mpk, id', m)$ only if $id' = id$.

Bob won't use his email address as a public key if a company uses IBE. Otherwise, if Bob's secret key is compromised, bob would need to change his email. Instead, Bob's public key should bob could use the identity string $(id, date)$. Bob will need to request a new secret key from Trudy every day. Now Alice can even encrypt emails for specific dates, and bob cannot access it before that, unless Trudy gives bob any key he requests.

I will not go into detail about the formal definition of security, but the idea is that the attacker has access to any $sk_{id'}$ it want but should not be able to break semantic security for some other identity with secret key sk_{id} that the adversary does not have.

IBE from pairings. We'll need a few components before being able to construct an IBE:

- a pairing. as defined above, over cyclic groups of prime order q, with generators g_0, g_1.
- a symmetric cipher $\mathcal{E} = (E_s, D_s)$.
- hash functions $H_0 : \mathcal{ID} \to \mathbb{G}_0$ and $H_1 : \mathbb{G}_1 \times \mathbb{G}_T \to \|$, where k is the key space.

Construction 1.

- $S()$: runs as follows:

$$\alpha \xleftarrow{R} \mathbb{Z}_q, u_1 \leftarrow g^\alpha, mpk \leftarrow u_1, msk \leftarrow \alpha \text{ output } (mpk, msk)$$

- $G(msk, id)$: key generation using $msk = \alpha$:

$$sk_id \leftarrow H_0(id)^\alpha \in \mathbb{G}_0 \text{ output } sk_id$$

- $E(mpk, id, m)$: encryption using the public parameters $mpk = u_1$:

$$\beta \xleftarrow{R} \mathbb{Z}_q,\ z \leftarrow e(H_0(id), mpk^\beta)$$
$$k \leftarrow H_1(g^\beta, z), c \xleftarrow{R} E_s(k, m),\ \text{output } (g^\beta, c)$$

The encryption generates a new secret key k at random. This private key relies on the pairing with the hashed public id. Once we've generated the ephemeral secret key k, we encrypt the message m.

- $D(sk_id, (w_1, c))$: decryption using secret key sk_id of ciphertext (w_1, c):

  $$z \leftarrow e(sk_{id}, w_1), k \leftarrow H_1(w_1, z), m \leftarrow D_s(k, c), \text{ output } m$$

  The decryption relies on $w_1 = g^\beta$ to figure out the symmetric encryption key k. With k, the message can be decrypted.
  To generate k correctly, we first remind the reader that

  $$sk = H(id)^\alpha, mpk^\beta = g^{\alpha\beta}, k = H(g^\beta, z)$$

  now, let's compute z, our missing component:

$$z = \underbrace{e(sk_{id}, g^\beta)}_{\text{z in decryption}} = e(H(id)^\alpha, g^\beta) = e(H(id), g^{\alpha\beta}) = \underbrace{e(H(id), mpk^\beta)}_{\text{z in encryption}}$$

CHAPTER 5

Fully Homomorphic Encryption

TODO; add homomorphic encryption definition. NOTE: they can support multiple operations.

We can have both a symmetric key HE, or an asymmetric key HE.

5.1. operations

Some homomorphic encryption schemes, such as BGV, BFV, and CKKS, support "packing" – or "batching" – many plain-texts into a single cipher-text.

- we can do logical ops; like AND, OR, XOR
- addition and multiplication
- ops over fixed point numbers
- rotations on packed cipher-texts

5.2. Hardness

Most HE schemes are using different Hardness assumptions than DDH.

In plain language, the basic hardness is called Learning with Errors, and it assumes that no adversary can infer whether $Ax + e = c$ or $Ax = c$ just from seeing A, c. the formal definition:

DEFINITION 5.2.1. *Learning With Errors (LWE) Problem. The LWE problem is parametrized by four parameters:* $(n, m, q, \mathcal{X})$ *where* n *is a positive integer referred to as the "dimension parameter",* m *is "the number of samples",* q *is a positive integer referred to as the "modulus parameter" and* $\mathcal{X}$ *is a probability distribution over rational integers referred to as the "error distribution". The LWE assumption requires that the following two probability distributions are computationally indistinguishable:*

Distribution 1.: *Choose a uniformly random matrix* $m \times n$ *matrix* A*, a uniformly random vector* s *from the vector space* $\mathbb{Z}_q^n$*, and a vector* e *from* $\mathbb{Z}^m$ *where each coordinate is chosen from the error distribution* $\mathcal{X}$*. Compute* $c := As + e$*, where all computations are carried out modulo* q*. Output* (A, c)*.*

Distribution 2.: *Choose a uniformly random* $m \times n$ *matrix* A*, and a uniformly random vector* c *from* $\mathbb{Z}_q^m$ *. Output* (A, c)*.*

In plain language, the basic hardness is called Learning with Errors, and it assumes that no adversary can infer whether $Ax + e = c$ or $Ax = c$ just from seeing A, c. the formal definition:

5.3. The BGV and BFV Encryption schemes

BFV is instantiated over two rings:

(1) The plaintext ring which includes encodings of unencrypted or intelligible messages;
(2) The ciphertext ring which includes encrypted messages.

Similar to any other FHE scheme, BFV allows an untrusted party to induce meaningful computation over en- crypted data without access to the decryption key.

5.3.1. BFV Primitives. The scheme consists of numerous algorithms:

- $ParamGen(\lambda) \rightarrow Params$: Parameter generator (ParamGen) takes as input the security parameter λ, which is a number used to define the security level of BFV, and returns a set of encryption parameters used in BFV. One can view λ as the computational cost of successful attacks on the scheme. In order for these attacks to succeed with probability 1, they would require 2^{λ} basic computational operations.
- $KeyGen\,(Params) \rightarrow (sk, pk, ek)$: Key generation (KeyGen) takes as input the encryption parameters and a secret key, a public key and evaluation key.
- $Encrypt\,(pk, m) \rightarrow c$: Encrypt takes as input pk and a plaintext message m in the plaintext space P, and returns a valid ciphertext c from the ciphertext space $\mathcal{C}$.
- $Decrypt\,(sk, c) \rightarrow m$: Decrypt takes as input sk and a valid ciphertext c in $\mathcal{C}$, which encrypts message m in P, and returns m.
- $EvalAdd\,(params, ek, c_1, c_2) \rightarrow c$: takes two ciphertexts and adds their underlying value m_1,m_2 respectively. outputs the encryption of $m_1 + m_2$
- $evalMult$ same thing...
- more ops...

5.4. Plaintext and Ciphertext Spaces

DEFINITION 5.4.1. *The notation $\mathbb{Z}_a\,[x]\,/\,(x^n+1)$ can be viewed as the set of polynomials with integer coefficients modulo both a and (x^n+1), i.e., with coefficients in $\left\{\lceil -\frac{a}{2} \rceil, \ldots \lfloor \frac{a-1}{2} \rfloor\right\}$ (from $-a/2$ to $a/2$) and of degree less than n. (i've read another definition, where we can just have coefficients modulo a.*

- *K[x] will meanthat k is the field we choose our coefficients from. and x is the number of variables.*

EXAMPLE 5.4.2. *The following examples are of valid plaintext messages for the parameters $n = 4$ (degree must be less than that) and $t = 5$ (to choose coefficients):*

(1) $m_0 = 1 + 2x + 1 \cdot x^2 - 1 \cdot x^3$
(2) $m_2 = -1 - 2x - 1x^2 + 2x^3$

The plaintext and ciphertext spaces in BFV are defined over two distinct polynomial rings. the plaintext space is denoted by the polynomial ring $\mathcal{P} = R_t =$

$\mathbb{Z}_t[x]/(x^n+1)$, that is, polynomials of degree less than n with coefficients modulo t.
the ciphertext space is denoted by $\mathcal{C} = (\mathbb{Z}_q[x]/(x^n+1)) \times (\mathbb{Z}_q[x]/(x^n+1))$.

5.5. BFV params

basically, some random distributions.

- R_2 used to sample polynomials with coefficients in $\{-1, 0, 1\}$.
- $\mathcal{X}$ is the error distribution, which is discrete Gaussian distribution.
- R_q is a uniform random distribution over R_q.

5.6. Plaintext Encoding and Decoding

Recall that the plaintext space is the polynomial ring R_t. This means that messages need to be converted to polynomials in R_t. Let m denote an integer message we would like to encrypt in FHE. The first encoding scheme (let's call it the naive encoding scheme) composes the plaintext element (polynomial) as $M = m + 0 \cdot x + 0 \cdot x^2 + \cdots + 0 \cdot x^{n-1}$, the constant polynomial. this scheme is extremely naive and inefficient. Let's do better.

another simple solution, but effective, would be to take the message m's binary representation $a_0 a_1 a_2 \ldots$ and convert them to the coefficients of the polynomial. If we have less bits than coefficients, set all coefficients with matching bits as 0. Probably can take the bytes instead of simple bits too.

To ensure that the results of homomorphic evaluation matches the expected results of the computation of interest, we need to ensure that the degree of the plaintext coefficient does not wrap around n and the coefficients do not wrap around t.

5.7. Key generation

The secret key sk is generated as a random ternary polynomial from R_2, a polynomial of degree n with coefficients in $\{-1, 0, 1\}$.

The public key pk is a pair of polynomials (pk_1, pk_2) calculated as follows:

$$pk_1 = [-1 \cdot (a \cdot sk + e)]_q$$
$$pk_2 = a$$

Where a is a random ploynomial in R_q ($R_q = \mathbb{Z}_q[x]/(x^n+1)$). e is a random error polynomial sampled from $\mathcal{X}$. the notation $[\cdot]_q$ means that the polynomial arithemtic should be done modulo q. Note that as ok_2 is in R_q, polynomial arithmetic should also be performed modulo the ring polynomial modulus (x^n+1).

5.8. Encryption and Decryption

encrypting an encoded message m requires 3 small random polynomials¨ $u \in R_2$ and $e_1, e_2 \in \mathcal{X}$. the ciphertext $c = (c_1, c_2)$ is generated as follows:

$$c_1 = \left[pk_1 \cdot u + e_1 + \lfloor \frac{q}{t} \rfloor \cdot m\right] q$$
$$c_2 = [pk_2 \cdot u + e_2]$$

Decryption is performed by evaluating the ciphertext on the secret key as follows and inverting the scaling factor $\left(\lfloor \frac{q}{t} \rfloor\right)$ applied in the encryption:

$$m = \left[\lceil \frac{t \cdot [c_1 + c_2 \cdot sk]\, q}{q} \rceil \right] t$$

let us expand this to gain better understanding.

$$\begin{aligned} c_1 + c_2 \cdot sk &= pk_1 \cdot u + e_1 + \lfloor \frac{q}{t} \rfloor \cdot m + (pk_2 \cdot u + e_2) \cdot sk = \\ &= \underbrace{-(a \cdot sk + e)}_{pk_1} \cdot u + e_1 + \lfloor \frac{q}{t} \rfloor \cdot m + a \cdot u \cdot sk + e_2 \cdot sk = \\ &= \cancel{-a \cdot u \cdot sk} - e \cdot u + e_q + \lfloor \frac{q}{t} \rfloor \cdot m + \cancel{a \cdot u \cdot sk} + e_2 \cdot sk = \\ &= \lfloor \frac{q}{t} \rfloor \cdot m - e \cdot u + e_1 + e_2 \cdot sk \\ &= \lfloor \frac{q}{t} \rfloor \cdot m + \underbrace{e_1 + e_2 \cdot sk - e \cdot u}_{} = \lfloor \frac{q}{t} \rfloor \cdot m + v \end{aligned}$$

the infinity norm of v (largest abs value from the coefficients) is pretty small: all of these: sk, e, e_1, e_2 are all **small** polynomials. İf these polynomials are bounded by some β then $||v|| \leq 2n \cdot \beta^2 + \beta$. (I'm skipping the proof of that).
They ensure that the noise is small, and thus they can recovering it.

5.9. Doing ops:

addition is very simple, and the additional error is not high. (can just add the cipher-texts together). Multiplication adds a lot of noise, in addition to that, from 2 cipher-textṡ we get 3 output cipher-texts (polynomials) which means a different encryption procedure (can be done by using exponents of the secret key). To overcome this one should do relinearization.

5.10. Maintenance Operations

The BFV scheme include operations that do not effect the underlying plaintext, but are needed for implementation reasons:

According to the "Protecting Privacy through Homomorphic Encryption" ciphertext-ciphertext multiplication have a side effect of requiring a different secret–key to decrypt the result than what was needed before the operation. thus, multiplication is followed by a key switching opeation to restore the secret key back to the original one. (relinearization). Avoiding relianirization is possible, but will reduce noise budget quicker. In addition to that, ops on non-relinearized ciphertexts are much slower.

Another such operation is **bootstrapping**, which "refreshes" a ciphertext and reduces the level of noit in it, to support more computations. It is a very expensive operation, and hence it isn't often used/ implemented.

5.11. security of the scheme

Choosing optimal BFV parameters that maximize performance and respect security and functionality constraints is an art that is practiced by expert cryptographers. but there is a standard one can follow.

But mainly, we first start by choosing the max size of integers (q) for the coefficients, and the max polynomial degree (n). but generally, larger n gives more security (but slower ops), larger q means we can do more complex computations.

Ciphertexts in these encryption schemes contain a noise component (which is important for security), and that noise grows with each operation (The encrypted result can only be decrypted if the noise is smaller than q, hence using larger values of q imply that we can do more operations).

5.12. Relinearization

Remeber we generated another key along with the private and public. called $ek = (ek_0, ek_1)$. using this key as randomness source, we can relinearize the result of multiplication: the new ciphertext would be :

$$c_1 = [C_1^* + ek_0 \cdot C_3^*]_q$$
$$c_2 = [C_2^* + ek_1 \cdot C_3^*]_q$$

Remember, decryption is : $c_1 + c_2 \cdot sk$. After doing that, the decryption would result with:

$$C_1^* + C_2^* \cdot sk + C_3^* \cdot sk^2 + C_3^* \cdot e$$

which would decrypt just fine, but with a big error ($C_3^* e$)! We can fix that by using base decomposition.

5.13. Base Decomposition

CHAPTER 6

Secret sharing, distributed key generation and threshold cryptography

6.1. Shamir secret sharing

Given a secret S, generate random polynomial $p = S + a_0 \cdot x + a_1 \cdot x + a_2 \cdot x^2 + \dots$ of degree $f + 1$. create shares $(p(1), p(2), \dots, p(n))$ where $n \geq f + 1$ and send it to nodes $(1, 2, ...n)$. The nodes will publish their shares when they want to reconstruct the secret. Using Lagrange interpolation and $f + 1$ shares, one can reconstruct the polynomial $p(0) = S$ to learn the secret.

6.2. Verifiable secret sharing

The dealer will generate a random polynomial $P(x) = S + a_1 \cdot x + a_2 \cdot x^2 + \dots a_n \cdot x^n$. It'll broadcast the polynomial as follows: $\{g^{a_i} | i \in [n] \cup S\}$. Then it'll send to each node i its share of the polynomial $P(i)$. To verify any share, including shares of other nodes, compute the following:

$$g^{P(i)} = g^S \cdot (g^{a_1})^i \cdot (g^{a_2})^{i^2} \cdots = \prod_{j=0}^{k} (g^{a_j})^{i^j} = g^{\sum_{j=0}^{t} a_j \cdot i^j} = g^{P(i)}$$

Every node can compute for any i the value $g^{P(i)}$ using the hidden coefficient broadcasted by the dealer. The ability to compute $g^{P(i)}$ guarantees that no one can publish a corrupt share and fool other nodes when they reconstruct the secret.

6.3. Distributed key generation

6.3.1. protocol. Assumptions, n players, where each player has a broadcast channel (Might cost n^2 to publish through) and a private channel to the other players. Note: I'll use player/ member interchangeably.

(1) Each member performs VSS. That is, player i generates a random polynomial $p_i(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \dots a_t \cdot x^t$, and broadcasts $\{g^{a_i} | i \in [t]\}$. Then it sends a private share $p_i(j)$ to each player j where $j > 0$.
(2) Each member verifies the share it receives according to the VSS protocol.
(3) Each member j takes the shares it received $\{p_1(j), p_2(j), \dots\}$ and adds them together. Then publish $g^{y_i} = g^{\sum_i p_i(j)}$.
(4) Each member verifies these shares. How? They take all the broadcasted values from the first step as follows:$\{g^{a_{ij}} | i \in [t], j \in Players\}$ and compute:

$$\{\prod_j g^{a_{1j}}, \prod_j g^{a_{2j}}, \dots\} =$$

$$\{g^{\sum_j a_{1j}}, g^{\sum_j a_{2j}}\}$$

Each member should be able to verify the share their peers sent using these values. Denote $\mathcal{Q}$ the distributed polynomial. Example:

$$g^{\mathcal{Q}(k)} = \prod_i (g^{\sum_j a_{ij}})^{k^i} = g^{\sum_i p_i(k)} = g^{y_k}$$

6.3.2. complexity.

Communication complexity. varies with the costs of broadcasting. using reliable broadcast means Each of these broadcasts costs $O(n^2)$. with up to constant rounds of broadcasting, we get $O(n^3)$ total communication costs in the system. Without reliable broadcast - $O(n^2)$ communication costs

Computation complexity. The top computation cost is creating $\mathcal{Q}$ then verifying each value. That is, computing $\{g^{\sum_j a_{1j}}, g^{\sum_j a_{2j}}, \dots\}$ is $O(n^2)$, and computing it for each player: $O(n^3)$

6.4. Largange interpolation in the exponent

We can modify cryptographic schemes using Lagrange interpolation to work with $f < n$ participants.

Given a dataset of coordinate pairs (x_j, y_j) on a polynomial $P(X) = \sum_{i=0}^{k} a_i^i \cdot x^k$ we can interpolate it as follows:

$$P(X) = L(x) = \sum_{j=0}^{k} y_j \ell_j(x)$$

where $\ell_j(x)$ is:

$$\prod_{0 \le m \le k,\ m \ne j} \frac{x - x_m}{x_j - x_m}$$

That is, we ensure $L(x_j) = y_j$.

6.4.1. Using it in crypto. For our convinicece, we publish onlt shares of the form: $\{(i, P(i)) | i > 0\}$.

Denote: $i_j \in T \subseteq [n]$, means index of player j. in order to reconstruct we'll need enough indices such that $|T| = deg(P) + 1$.

So to evaluate $L(0)$ we'll need to compute for each $i_j \in T$:

$$\ell_{i_j}(0) = \frac{\prod_{k \in T \setminus \{i_j\}} \mathbf{0} - k}{\prod_{k \in T \setminus \{i_j\}} (i_j - k)}$$

Now, let us finally evaluate the secret:

$$L(0) = \sum_{i_j \in T} \ell_{i_j} \cdot P(i_j)$$

$n - out - of - n$ protocol. We'll take a complex and unfamiliar example to ensure one can apply it to other schemes. NOTE: We rely in the following example on pairing-based cryptography. It'll have slight differences. for example, using the key pair (sk, g^{sk}) one can sign a message m as follows: $H(m)^{sk}$.

Let's begin:

A node can publish a secret tied to a specific ID in the following protocol (Assume we have pairings). Each user has a key pair (sk_i, pk_i), and the shared public key would be $\prod_i pk_i = pk$. that is $\prod_i g^{sk_i} = g^{\sum_i sk_i}$.

to hide a new transaction tx: sample new keys: (sk', pk') and a random number r.

- compute $\gamma = Enc_{pk'}(tx)$
- compute $c = \left(\underbrace{g^r}_{c_L}, \underbrace{e\left(pk, H(\gamma)^r\right) \cdot sk'}_{c_R} \right)$
- send (c, γ, pk').

Now, when every server signs: $\sigma_i = H(\gamma)^{sk_i}$
we can compute the product signature: $\sigma = \prod H(\gamma)^{sk_i}$ which can be used to decrypt the temp secret key:

$$\begin{aligned}
\frac{c_R}{e(c_L, \sigma)} &= \frac{e\left(g^x, H(\gamma)^r\right) sk'}{e\left(g^r, \sigma\right)} = \\
&= \frac{e\left(g^{\sum sk_i}, H(\gamma)^r\right) \cdot sk'}{e\left(g^r, H(\gamma)^{\sum sk_i}\right)} = \\
&= \frac{e\left(g^{\sum sk_i}, H(\gamma)^r\right) \cdot sk'}{e\left(g, H(\gamma)\right)^{r \cdot \sum sk_i}} = \\
&= \frac{e\left(g, H(\gamma)\right)^{r \sum sk_i} \cdot sk'}{e\left(g, H(\gamma)\right)^{r \cdot \sum sk_i}} = \\
&= sk'
\end{aligned}$$

we managed to find out the temp secret key. now we can get the value under $\gamma = Enc_{pk'}(tx)$.

$f - out - of - n$ *protocol.* We assume all nodes have shares of the same polynomial P. This protocol will have slight changes from the one described before. It'll ensure we need only $f - out - of - n$ signatures to reconstruct the secret.

Each player holds a share s_j (and a matching public key g^{s_j}), generated from Polynomial P. The shared secret key is $P(0) = S = \sum_{k \in T} \ell_k(0) \cdot s_k$, and the public key is

$$PK = \prod_{k \in T} \left(g^{s_k}\right)^{\ell_k(0)} = g^{\sum_{k \in T} \ell_k(0) \cdot s_k} = g^{L(0)} = g^{P(0)} = g^S$$

for $|T| = f + 1$ indices.

Creating new secret. For each transaction tx: sample new keys: (sk') and a random number r.

- compute $\gamma = Enc_{sk'}(tx)$
- compute $c = \left(\underbrace{g^r}_{c_L}, \underbrace{e\left(PK, H(\gamma)^r\right) \cdot sk'}_{c_R} \right)$
- send (c, γ).

reconstruction. With $f + 1$ signatures over $H(\gamma)$, we can recompute the secret and extract out of c the decryption key sk'. When a player receives a signature $\sigma_i = H(\gamma)^{s_i}$ It verifies the signature and then stores $(\sigma_i)^{\ell_i(0)}$. With enough signatures, we can compute the product signature:

$$\sigma = \prod_{k \in T} \sigma_k^{\ell_k(0)} = \prod_{k \in T} H(\gamma)^{\ell_k(0) \cdot s_k} = H(\gamma)^{\sum_{k \in T} \ell_k(0) \cdot s_k} = H(\gamma)^{L(0)} = H(\gamma)^S$$

Which can be used to decrypt the temp secret key:

$$\begin{aligned}
&\frac{c_R}{e\left(c_L,\sigma\right)}=\frac{e\left(PK,H\left(\gamma\right)^r\right)\cdot sk'}{e\left(g^r,\sigma\right)}= \\
&=\frac{e\left(g^S,H\left(\gamma\right)^r\right)\cdot sk'}{e\left(g^r,H(\gamma)^S\right)}= \\
&=\frac{e\left(g^S,H\left(\gamma\right)^r\right)}{e\left(g^r,H(\gamma)^S\right)}\cdot sk'= \\
&=\frac{e\left(g,H\left(\gamma\right)\right)^{S\cdot r}}{e\left(g,H(\gamma)\right)^{r\cdot S}}\cdot sk'= \\
&=sk'
\end{aligned}$$

we managed to find out the temp secret key. now we can get the value under $\gamma = Enc_{pk'}\left(tx\right)$. as wanted.

CHAPTER 7

protocols for identification and login

7.1. Identification Protocols

.

7.1.1. Types of adversaries. The identification protocol is a set of a prover P and a verifier V. where the prover wants to gain access to some resource, and to do so, the verifier must give it access. the verifier will only give access to a prover that convinces it that it has the rights to that resource. for example, P wants to enter its house, and to do so it must unlock the door. the convincing method is a key.

Direct attacks: The adversary is not able to eavesdrop. Then using no information other than what is publicly avail- able, the adversary must somehow impersonate the prover to the verifier. A simple password protocol is sufficient to defend against such direct attacks.

Eavesdropping attacks: The adversary can eavesdrop and obtain the transcript of several interactions between the prover and verifier. In this case the simple password protocol is insecure. However, a slightly more sophisticated protocol based on one-time passwords (TBD) is secure.

Active attacks: an active adversary that interacts with the prover. The adversary uses the interaction to try and learn something that will let it later impersonate the prover to the verifier. Identification protocols secure against such active attacks require interaction between the prover and verifier. They use a technique called challenge-response.

7.1.2. Salting.

Storing the password in plaintext is prone to active attacks, once the verifier is compromised, the adversary can gain access easily. A simple solution would be to hash the plain-texts. unfortunately the space of possible passwords being used is not very large. Hence in addition to hashing the plain-texts, add salt!

To salt a plain-text, the verifier should add some random string to it, then store the salt as plain-text, and hash the password and the salt concat to it $(salt, Hash\,(salt||password))$.

Salting ensures a direct attack would need to run an exhaustive search over $D \times S$ where D is the dictionary with all weak passwords and S is the space of random strings the verifier uses, for example salt size can be $|S| = 2^{128}$.

That is not really the case though, it just ensures the attacker cannot use a preprocessing algorithm to have efficient attacks. (?the attacker needs to make a preprocess for all salts it saw in the server along with the dictionary.)
Now its best case would be to run on a dict along hash the plaintext hash with the

word from the dict in the hopes of finding a similar hash in the servers password file.

7.1.2.1. *salt and pepper.* skipped it

7.1.3. eavesdrop safe.

7.1.3.1. *Hotp.* HOTP is a protocol for identification which is safe against eavesdrop attacks.

This attack attempts to change the password after each use. To do so, the prover and verifier should share a secret, which they will then be used to generate some proof of authenticity. Using $HMAC$ (or a PRF as follows $F(k,i)$) the prover generates a new value over a shared counter: $Hmac(i)$ and will send it to the verifier. verifier can authenticate it. and then they both advance i by 1.

REMARK 7.1.1. *We can improve it by sending both i and the output of the $HMAC$ over i, this way the verifier can check if i is greater than its counter i', so they do not have to be matching counters at all times.*

REMARK 7.1.2. *This verification protocol changes state only upon an attempt to login. A user that does not login frequently, might be compromised if an attacker gained the latest token. The attacker can sell the obtain token to anyone which should give access once unless the user logins again.*

7.1.3.2. *TOPT: time base OTP.* Instead of a counter, one can use the current time, and each login attempt is only valid if the timestamp is recent enough, say 10 seconds.

7.1.3.3. *S/key.* TODO.

attempts to solve the leaking key issue:
generate random k, give the verifier $H^{(n+1)}(k)$. the prover stores (k,n) to send identification request:

- P sends $t = H^i(k)$ then sets $i = i - 1$.
- V will inspect that $H(t)$ equals to what it stored, if it does it updates the storage as t.

this can be used for n times, afterwards needs to be regenerated. kinda cool.

basically, the provers sends $H^n(k)$ then $H^{n-1}(k)$ ... $H(k)$.

- Assumes that the key is kept secret by the client
- the resulting password is at least 128bit, which is long to type by humans.

You reached 18.6 in the book, which is talking about security against active attacks.

7.2. Active attack

the attacker can impersonate the verifier for a brief moment, and its goal is to find a secret to fool the actual verifier.

thus HOTP is easy to break, gain a legit one time password and forward it to the verification protocol.

7.2.1. challenge-response protocol: the verifier sends a random challenge c, the prover uses a mac over the challenge c and send it back. the attacker can only interact with the prover, it cannot prepare ask the prover all possible challenges the verifier will chuck at him. and because we use MAC the attacker cannot forge a response from the prover.

7.2.1.1. *improvement.* to ensure the verifier can be compromised and the attacker can't do anything to gain access later using active attack - we use signature scheme instead of mac.

CHAPTER 8

Identification and signatures from sigma protocols

This chapter will be useful to develop zero–knowledge proofs.

8.1. Schnorr's identification protocol

Schnorr's identification protocol is a basic block for building signature schemes with DH assumption as its base. This will be our first proof of knowledge ever to be seen.

This protocol can be proved secure against eavesdropping attacks, assuming the discrete logarithm problem is hard. Let $\mathbb{G}$ be a cyclic group of prime order $q$with a generator $g \in \mathbb{G}$. Suppose the prover has a secret key $\alpha \in \mathbb{Z}_q$ and a corresponding public key $u = g^\alpha$. P wishes to prove to an identifier V that it knows α.

8.1.0.1. *protocol.* both the prover and the verifier generates random numbers respectively k, c. The prover send g^k and once it receives the challenge c from the verifier, it'll need to compute $t = k + \alpha \cdot c$ and send t to the verifier. in turn, the verifier should compute the following:$(u)^c \cdot g^k = g^{k+\alpha \cdot c}$ which should be equal to g^t.

A keen eye could see that this is very similar already to Schnorr's signature scheme.

8.1.0.2. *Honest verifier zero knowledge and security against eavesdropping.* We prover that the protocol is eavesdrop safe by assuming the adversary has access to vk, and that it had seen a conversation between P and V. The idea is to show that these conversations do not help the adversary, because the adversary could have efficiently generated these conversations by himself, given vk (but not sk).

DEFINITION 8.1.1. *Let $\mathcal{I} = (G, P, V)$ be an identification protocol. We say that $\mathcal{I}$ is honest verifier zero knowledge, or HVZK for short, if there exists an efficient probabilistic algorithm Sim (called a simulator) such that for all possible outputs (vk, sk) of G, the output distribution of Sim on input vk is identical to the distribution of a transcript of a conversation between P (on input sk) and V (on input vk).*

Comments on the terminology are in order. The term "zero knowledge" is meant to suggest that an adversary learns nothing from P, because an adversary can simulate conversations on his own (using the algorithm Sim), without knowing sk. The term "honest verifier" conveys the fact this simulation only works for conversations between P and the actual, "honest" verifier V , and not some arbitrary, "dishonest" verifier, such as may arise in an **active attack** on the identification.

THEOREM 8.1.1.1. *If an identification protocol $\mathcal{I}$ is secure against direct attacks, and is HVZK, then it is secure against eavesdropping attacks.*

The main idea here: if I'm secure against someone interacting maliciously with the verifier. and in addition to that, my regular protocol does not provide any useful information someone can use later on: im safe against eavsedrop attacks too.

Schnorr's protocol is HVZK, they prove it using a simulator. In addition to that, they show it's safe against direct attacks by showing that if there was some attacker that can break it, it can generate twice accepting answers to the verifier. then they can use these two answers to BREAK the discrete log assumption.

8.1.1. ECDSA. The ECDSA signature scheme (G, S, V) uses the group of points $\mathbb{G}$ of an elliptic curve over a finite field F_p. Let g be a generator of $\mathbb{G}$ and let q be the order of the group $\mathbb{G}$, which we assume is prime. Im skipping the algorithm, because it isn't as elegant. in addition to that, Schnorr sigs are strongly secure, but ECDSA are not. Given an ECDSA signature $\sigma = (r, s)$ on a message m, anyone can generate more signatures on m.

8.2. Sigma protocols: basic definitions

Schnorr's identification protocol is a special case of an incredibly useful class of protocols called Sigma protocols. We'll soon see the basic concepts associated with Sigma protocols and how they can become helpful to us.
Later we'll see how Sigma protocols are useful to us, even out of the context of identification and signatures.

DEFINITION 8.2.1. *(Effective relation). An effective relation is a binary relation $R \subseteq X \times Y$, where X, Y and R are efficiently recognizable finite sets. Elements of Y are called **statements**. If $(x, y) \in R$, then x is called a witness for y.*

DEFINITION 8.2.2. *Sigma Protocol. Let $R \subseteq X \times Y$ be an effective relation. A **Sigma Protocol** for R is a pair (P, V).*

- *P is an interactive protocol algorithm called the **prover**, which takes as input a witness–statement pair $(x, y) \in R$.*
- *V is an interactive protocol algorithm called the **verifier**, which takes as input a statement $y \in Y$, and which outputs accept or reject.*
- *P and V are structured so that an interaction between them always works as follows:*
  - *To start the protocol, P computes a message t, called the commitment, and sends t to V;*
  - *Upon receiving P's commitment t, V chooses a challenge c at random from a finite challenge space C, and sends c to P;*
  - *Upon receiving V 's challenge c, P computes a response z, and sends z to V;*
  - *Upon receiving P's response z, V outputs either **accept** or **reject**, which must be computed strictly as a function of the statement y and the conversation (t, c, z). In particular, V does not make any random choices other than the selection of the challenge — all other computations are completely deterministic.*

We require that for all $(x, y) \in R$, when $P(x, y)$ and $V(y)$ interact with each other, $V(y)$ always outputs accept.

REMARK 8.2.3. *So a sigma protocol, is any prover verifier challenge where the prover knows some witness to y and want to prove that y is in the language to the verifier?*

If the output is accept we call the conversation (t, c, z) an accepting conversation for y.

EXAMPLE 8.2.4. *It should be clear that for Schnorr's identification protocol (G, P, V), the pair (P, V) is an example of a* ***Sigma protocol*** *for the relation $R \subseteq X \times Y$, where*

$$X = \mathbb{Z}_q, Y = \mathbb{G} \text{ and } R = \{(\alpha, g^\alpha) \in \mathbb{Z} \times \mathbb{G}\}.$$

TODO continue

CHAPTER 9

MPC

We have a function f that takes n inputs and produces m outputs:

$$(y_1, ..., y_m) = f(x_1, ..., x_n).$$

We also have N parties, $P_1, ..., P_N$. Their goal is to run a protocol where each input value x_i is contributed by one of the parties and each output y_j is obtained by one or more of the parties.

We want to design an efficient protocol for this problem that provides *privacy, soundness, and input independence.*

9.1. Wanted properties

- Privacy: No part learns anything about any other part's inputs (except for the information that is inherently revealed by the outputs).
- Soundness/correctness: honest parties compute correct outputs.
- Input independence: All parties must choose their inputs independently of the other parties' inputs (No one can use Alice's inputs to their advantage).
- Guaranteed output delivery: all honest parties are guaranteed to obtain their output.
- Fairness: If any party (including corrupted members) obtains their output, then all honest parties do so.

9.2. Formal Security

The way to define security formally is to say that an attack on the protocol in the "real world" is equivalent to some attack on the protocol in an "ideal world" in which no damage can be done.

In the ideal world, the protocol is implemented using a trusted party to which all parties (both honest and corrupt) submit inputs, and from which all parties (both honest and corrupt) obtain their designated outputs. The trusted party itself cannot be corrupted.

We then describe the details of the ideal-world execution, including the behavior of the trusted party who evaluates the function.

The definition of security then basically says: for every efficient adversary A in the real world, there exists an "equivalent" efficient adversary $\mathcal{S}$ in the ideal world (usually called a simulator). Since there is no possible attack in the ideal world, there is no possible attack in the real world.

the informal notions of *privacy, soundness, and input independence* are implied by this type of definition.

Essentially we'll want to show that the ideal world we describe is "equivalent" to the real world.

9.2.1. A few applications of MPC.

- Elections: Without MPC, parties submit votes to a trusted server T who counts the votes and states the winner.
- Privacy-preserving mining of medical data. Two hospitals want to share data, to perform research, but cannot share it due to regulations.

9.3. Securely evaluating arithmetic circuits

CHAPTER 10

Algorithms

10.1. FFT, DFT and NTT

10.1.1. representations of a polynomial. we can either use *coefficient representation* or *Point-value representation*.

coefficient representation. A coefficient representation of a polynomial $P(x)$ is a vector of coefficients $a = (a_0, a_1, \ldots, a_{n-1})$. In coefficient representation, we can evaluate or add a polynomial in $O(n)$ time. unfortunately multiplying two polynomials takes $O(n^2)$. The operation of multiplying two polynomials a, b is called convolution $c = a \otimes b$.

Point-value representation. A point-value representation of a polynomial $P(x)$ of degree-bound n is a set of n point-value pairs.

$$\{(x_0, P(x_0)), \ldots, (x_{n-1}, P(x_{n-1}))\}$$

THEOREM 10.1.1 (Uniqueness of an interpolating polynomial). *For any set $\{(x_0, P(x_0)), \ldots, (x_{n-1}, P(x_{n-1}))\}$ of n point-value pairs such that all the x_k values are distinct, there is a unique polynomial $P(x)$ of degree-bound n such that $y_k = P(x_k)$. for $k \in [n]$*

Due to the above theorem, we can see that this representation defines a polynomial well. And one can move between both representations.

Because $C(X) = A(X)B(X)$ for any x_j, then $C(x_j) = A(x_j)B(x_j)$. Meaning multiplication and addition in this representation cost $O(n)$.

10.2. FFT

Luckily there is a way to move to and from both representations quickly using the *roots of unity*.

DEFINITION 10.2.1. *An nth root of unity is a complex number ω such that $\omega^n = 1$.*

There are exactly n roots of unity $e^{2\pi ik/n}$ for $k \in [n-1]$ ($e^{iu} = \cos u + i \sin u$).

The value $\omega_n = e^{2\pi i/n}$ is the *principal nth root of unity.* all other roots of unity are powers of ω_n: $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$.

FFT takes special advantage that these values have special properties.

LEMMA 10.2.2 (roots of unity). *If z is an nth root of unity, then $a \equiv b \mod n \Rightarrow z^a = z^b$.*

meaning that $\omega_n^a = \omega_n^b$. one can label specific points on the unit circle over $\mathbb{C}$.

Why is that important? In the simplest of cases, we are going to evaluate the polynomial over the following points $\{\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}\}$. Let us view the polynomial evaluation over n such points as a matrix, this presentation might help the reader see some similarities:

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2, \ldots & \\ 1 & x_1 & x_1^2, \ldots, & \\ \vdots & \vdots & \vdots & \ddots \\ 1 & x_{n-1} & x_{n-1}^2, \ldots, & \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Now, let us assign the values:

$$\begin{pmatrix} p(\omega_n^0) \\ p(\omega_n^1) \\ \vdots \\ \omega_n^2 \end{pmatrix} = \begin{pmatrix} 1 & (\omega_n^0)^1 & (\omega_n^0)^2, \ldots, (\omega_n^0)^{n-1} & \\ 1 & (\omega_n^1)^1 & (\omega_n^1)^2, \ldots, (\omega_n^1)^{n-1} & \\ \vdots & \vdots & \vdots & \ddots \\ 1 & (\omega_n^{n-1}) & (\omega_n^{n-1})^2, \ldots, (\omega_n^{n-1})^{n-1} & \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

The keen looker can see that we can reuse values in this matrix. for example, let us look at $(\omega_n^{n-1})^{n-1}$ using **??** $(\omega_n^{n-1})^{n-1} = (\omega_n)^{n*n-2n+1} = (\omega_n)^{0+0+1}$.

Thus we gain a matrix with heavy symmetry, which the FFT can use.

fft main idea. The FFT method employs a divide-and-conquer strategy, using the even-indexed and odd-indexed coefficients of $A(x)$ separately to define the two new polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ of degree-bound $n/2$.

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} \tag{10.2.1}$$

$$A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}. \tag{10.2.2}$$

It follows that $A(x) = A^{[0]}(x^2) + x \cdot A^{[1]}(x^2)$.

Thus, the problem of evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$ reduces to evaluating the degree-bound $n/2$ polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ at the points

$$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$$

and then combining the results according to $A(x) = A^{[0]}(x^2) + x \cdot A^{[1]}(x^2)$.

Using the properties of the root of unity, the above values consist not of n distinct values but only of the $n/2$ complex $(n/2)$th roots of unity, with each root occurring exactly twice! These subproblems have the same form as the original problem but are half the size. We can continue in this manner until we evaluate the full polynomial on n distinct values.

10.2.1. putting it all together. there might be some miscalculations here, but the idea is solid. As we've seen we can create a matrix with heavy symmetry, called the DFT matrix. now, we'd want to capitalize on that and create a recursive algorithm. To do so, instead of the regular polynomial assignment we've seen above using a matrix, we'll shuffle things a bit according to the divide-and-conquer approach from above

$$F_n \cdot p(x) = \begin{pmatrix} I_{\frac{n}{2}} & D_{\frac{n}{2}} \\ I_{\frac{n}{2}} & -D_{\frac{n}{2}} \end{pmatrix} \cdot \begin{pmatrix} F_{\frac{n}{2}} & 0 \\ 0 & F_{\frac{n}{2}} \end{pmatrix} \cdot \begin{pmatrix} p_{\text{even coefficient}} \\ p_{\text{odd coefficient}} \end{pmatrix}$$

where F_n is the matrix of assignment we've seen in **??**, and D_n is the following diagonal matrix

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & \omega_n^1 & 0 & \dots & 0 \\ 1 & 0 & \omega_n^2 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & 0 & \dots & \omega_n^n - 1 \end{pmatrix}$$

We can continue this process recursively, to get what we want. In each recursive level, the algorithm does at most $O(n)$ work (i.e., multiplying by sparse matrices), and due to the logarithmic depth of the recursion, we get $O(nlogn)$.

```python
import numpy as np

def fft(p: np.ndarray) -> np.ndarray:
    n = len(p)
    if n == 1:
        return p
    omega = np.exp(2 * np.pi * 1j / n)

    p_even, p_odd = p[::2], p[1::2]
    ye, yo = fft(p_even), fft(p_odd)

    # mult the structured left matrix with the recursive result.
    # O(n) work comes from here.
    y = np.zeros(n, dtype=np.complex128)
    for k in range(n // 2):
        y[k] = ye[k] + omega ** k * yo[k]
        y[k + n // 2] = ye[k] - omega ** k * yo[k]
    return y

```

Listing 10.1. fft algorithm in python

Remark 10.2.3 (Summary). *Notice that to achieve this result, we needed to reduce the polynomial into even and odd parts, and then adjust the assignment matrix using the properties of roots of unity. Each part is important in its own way.*

Interpolation. The process is similar, we just need to invert F_n **??**. TODO

Corollary 10.2.3.1. *For FFT, DFT, and NTT we must use specific values to gain speedy interpolation and evaluation of the polynomial.*

Conclusin:

10.2.2. NTT. The NTT is a specialized version of the discrete Fourier transform, in which the coefficient ring is taken to be a finite field (or ring) containing the right roots of unity. It can be viewed as an exact version of the complex DFT, avoiding round-off errors for exact convolutions of integer sequences. While Gauss used similar techniques already in [2], laying the groundwork for modern FFT algorithms to compute the DFT, and therefore the NTT, is usually attributed to Cooley and Tukey's seminal paper [1].

10.2.2.1. *Special Features.* Applying the NTT transform provides a cyclic convolution, computing $c = a \cdot b \mod (X^n + 1)$ with two polynomials a and b would require applying the NTT of length $2n$ and thus n zeros to be appended to each input; this effectively doubles the length of the inputs and also requires the computation of an explicit reduction modulo $X^n + 1$. To avoid these issues, one can exploit the *negative wrapped convolution* [3].

This exploitation allows us to perform NTT without expanding to $2n$ parameters! meaning we can send less bandwidth over the network, and perform fewer computations too. This trick can be seen in [4].

10.3. Reed-Solomon

It is fairly simple, but effective algorithm to encode words and be able to deal with erasurs.

10.3.1. Naive solution. assuming we decide we want our words to be of size $k = 4$, and we add additional two redundancies $n = 6$. Thus to send it we'll do the following: given a word vector, we'll treat it as coefficients values of a polynomial of degree $k-1$ (because we have k coefficients its a k-1 degree poly), then to encode it, we'll evaluate it on the following x values: $(-1, 0, 1, 2, \dots n)$.

Now given the 6 values that we received over the network we can decode it as follows: We go over every group of 4 points, and try to reconstruct the polynomial.

the polynomial we've reconstructed the most is the correct one.

10.3.2. Systematic Reed-Solomon. Assuming we want to be able to correct up to s values, we'd need at least $2s$ redundant symbols. So if $k = 4$, and $n = 6$, we can only recover a single byte.

It won't work for larger values, like $k = 223$, $n = 255$ which is a popular value for reed-solomon. (Meaning we add 32 more symbols to our message, so we can repair 16 symbols in total).

The previous algorithm will work, but we'd need to send different values then the original message. It would be best to send $Message||$addition code word. that way,we might be able to avoid reconstruction if we can find quickly that there was no error.

this is called a Systematic code.

Bibliography

[1] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2003354 (visited on 05/30/2023).

[2] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. "Gauss and the history of the fast Fourier transform". In: *Archive for History of Exact Sciences* 34.3 (1985), pp. 265–277. DOI: 10.1007/BF00348431. URL: https://doi.org/10.1007/BF00348431.

[3] Vadim Lyubashevsky et al. "SWIFFT: A Modest Proposal for FFT Hashing". In: *Fast Software Encryption*. Ed. by Kaisa Nyberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 54–72. ISBN: 978-3-540-71039-4.

[4] *Microsoft SEAL (release 4.1)*. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. Jan. 2023.