DS 453 Crypto for Data Science

Xiang Fu
xfu@bu.edu
Boston University Faculty of Computing & Data Sciences

Contents

1 Introduction to Crypto for Data Science

> Cryptography is how people get things done when they need one another, don't fully trust one another, and have adversaries actively trying to screw things up.
>
> — Ben Adida

1.1 Introduction to Crypto (Graphy)

This is a course about crypto... which is admittedly a term with many different meanings to different people. It's also an abbreviation with (at least) three different endings.

- Cryptocurrencies
- Cryptography
- Cryptology

We will get to the topic of cryptocurrencies in a few minutes. But first, here is a question to start us off in this course: what is cryptology?

Cryptology comes from two Greek word parts:

- kryptos, meaning secret or hidden
- logy, meaning the study of

So the goal of cryptology is to study the art of keeping secrets.

In its modern form, crypto uses hard math problems as a way to encode data in order to restrict who can use it, how they can use it, and sometimes even when and where they can access it.

Cryptology can be divided into two sub-fields

- Cryptography: the art of making codes.
- Cryptanalysis: the art of breaking codes.

Both cryptography and cryptanalysis are challenging, multi-disciplinary tasks. They also reinforce each other, due to Schneier's law.

> Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break.
>
> — Bruce Schneier

In this course we will almost exclusively focus on the defensive, cryptographic side. Even though I won't say much about cryptanalysis in this course, it is important to know that all of the building blocks I will show you in today's lecture have been vetted through decades of cryptanalysts, collectively spanning many centuries or millenia of people-years. No single one of us can compete with that.

> Don't roll your own crypto.
>
> — (Every cryptographer)

Crypto is a scientific field at the intersection of many disciplines. To accomplish their goals, cryptographers use:

- Algorithms techniques in their protocol designs.
- Engineering techniques to produce vetted, secure software implementations of their protocols.
- Complexity theory to demonstrate security reductions from new protocols to more well-studied ones.
- Mathematics for cryptanalysis.

This class will primarily make use of the first two skills: algorithms and software engineering.

While we won't delve into the specific type of math used in cryptanalysis, nevertheless we will often use concepts from algebra and probability.

1.1.1 Crypto and Data Science

This is also a course about how crypto can be used within data science.

It's likely that you have already used crypto in all of your data science projects, perhaps without explicitly thinking about it.

In this image, the lock icon in the web browser means that my computer has a protected connection to the kaggle.com servers.

At a high level, this means that nobody else on the internet can

- see which dataset I am downloading from kaggle.com; in other words, the data remains confidential, and
- tamper with the dataset in transit; in other words, data integrity is maintained.

(Note though that the kaggle.com servers can see what you are doing on their website. We will see later in the semester how even this can potentially be avoided.)

Using crypto is an essential part of providing security over the Internet, because it has always been designed as an open network. There is no wire that directly connects your computer with kaggle.com. Instead, the data flows through several other computers along the way.

But the design of the Internet has fundamentally remained the same.

> The Internet is just the world passing notes in a classroom.
>
> — Jon Stewart

Getting the crypto right is essential here.

- If the system is vulnerable to decoding or tampering by unauthorized parties, then it can lead to universal, covert breaches of security. Moreover, everyone will have a false sense of security.
- If the system is too onerous or slow to use, then people will ignore it and move to other, less secure options.

1.1.2 Goal of Cryptography

We will see throughout this course that crypto can be used in many ways besides network transmissions on the Internet.

Still though, this example highlights a general theme of all of our applications:

> Crypto is a social science that masquerades as a mathematical science.
>
> — Someone

Our goal is to use the tools from math, computer science, and data science in order to design systems that help people.

At a high level, cryptographic systems attempt to provide (some or all of) the following three objectives.

- C: Confidentiality, meaning to keep data or people's identities hidden from others.
- I: Integrity, which means preventing raw data or the data science results from being altered, or at least allowing for detection of tampering.
- A: Availability, or ensuring that data science is censorship-resistant.

These three security objectives are sometimes referred to as the "**CIA triad**."

Admittedly, these three objectives are both underspecified (lacking mathematical rigor) and overloaded (the terms have many different sub-meanings).

There are many different flavors of confidentiality, integrity, and availability that cryptography can provide… in fact, far more than we have time to cover in this course.

(Note: BU courses CS 538 and CS 558 also cover different aspects of cryptography and its application in modern digital systems. This course has a partial overlap with them, but we will cover many topics that they don't and vice-versa. You are welcome to register for those courses as well, but they are not required knowledge.)

1.1.3 Crypto and Policy

Because crypto is a scientific field that has social impacts, it is probably inevitable that crypto has clashed with law and policy over the past five decades.

- On the one hand, crypto offers ways to upend existing norms, laws, and rules… in both good and bad ways.
- On the other hand, the use of crypto is influenced and regulated by the law and politicians.

> Cryptography rearranges power: it configures who can do what, from what. This makes cryptography an inherently political tool, and it confers on the field an intrinsically moral dimension.
>
> — Phillip Rogaway

We will see aspects of the two-way connection between crypto and policy throughout the semester, and we will spend the last few weeks focusing exclusively on it.

1.2 Introduction to Crypto (Currencies)

[Note: This section of the lecture is based on Johns Hopkins CS 601.641 by Prof. Abishek Jain.]

Pausing the cryptography discussion for a moment, let's provide an overview of cryptocurrencies like Bitcoin and Ethereum. We will spend the next few weeks delving into their design.

From the start, it is worthwhile to state what this course is NOT about.

- This is not a finance course on cryptocurrencies. You should not expect to be taught how to invest in cryptocurrencies or how to become a billionaire overnight.
- In fact, the cryptocurrency market is still in its infancy and is incredibly volatile. Investing in cryptocurrencies is a good way to lose money. So this course doesn't offer any investment advice, except in this sentence: I advise you not to invest in cryptocurrencies.

Still though, cryptocurrencies are a worthy topic of study in order to understand their capabilities, potential, and limitations.

In the first unit of this course, we will explore the following topics about cryptocurrencies.

- Understanding the mechanics of blockchains
- Understanding why current implementations work
- Understanding the necessary cryptographic background
- Exploring applications of blockchains to cryptocurrencies and beyond
- Understanding limitations of current blockchains

Let's start at the top of the list. What is a blockchain?

It is:

- A distributed ledger or database
- Used for building decentralized cryptocurrencies such as Bitcoin
- Capable of improving many other applications such as distributed Domain Name system (DNS), Public-Key Infrastructure (PKI), stock trade databases, etc.
- The basis of lots of exciting academic research and industry startups

1.2.1 Transferring Digital Money

Blockchains are a data structure to keep track of transactions.

Here is the motivating scenario for this unit: there are two people, who (following the typical convention in crypto) we will call Alice and Bob.

Alice would like to transfer 1 to Bob.

Let's think about

- the functionality of how this could work, i.e., how it can work if everyone is honest
- the security concerns, i.e., ways this could go wrong if someone is malicious

Let's start by exploring how money transfers work in the world of physical banking. Suppose Alice writes a check to Bob. What happens?

If everyone is honest, the money is transferred because a bank performs the bookkeeping task of recording every account and its balance.

Now let's think about what happens if each party is malicious. For this single transaction, Alice doesn't really care whether Bob is malicious or not; she isn't expecting to receive anything of value.

In the other direction: even if Bob trusts the bank (for now), he may be suspicious that Alice is trying to deceive him. He needs to be able to verify the following three properties.

- Alice properly signed the check: To accomplish this goal, Bob can verify the signature himself, say by comparing the check against Alice's identification card.
- Alice possesses 1 in her bank account: Bob can ask the bank to verify that Alice's account balance is sufficient for the check to clear. Note that Bob doesn't even need to learn Alice's balance, just the result of the binary predicate.
- Alice does not double spend the money by writing a check to someone else at the same time: This is the toughest property to achieve, because neither Bob nor the bank know yet whether Alice has overdrafted her account. They will only discover this later; if so, Alice and Bob can use the bank and the legal system to settle their dispute.

By contrast, the digital world is diverse and international. There is no single entity that everyone in the world trusts, nor is there a single legal system to identify and prosecute thieves and money launderers.

So, if we want a digital system of money, then we need a different approach altogether.

It will take us a few lectures to build up the tools for a digital system of transferring money. First, we need to understand two essential crypto tools: digital signatures and hash functions.

1.3 Introduction to Digital Signatures

Today, we only focus on property #1: building a digital version of signing a check.

The digital equivalent to "signing a check" is called, appropriately enough, a digital signature scheme. The most common standard is called the digital signature algorithm, or DSA.

A digital signature allows Alice to append a signature σ to a message M

that satisfies two properties.

- Correctness: Everyone in the world can verify whether Alice created the signature σ.
- Unforgeability: Only Alice can produce the signature σ, and it is bound to the message M'. Nobody besides Alice can use σ in order to forge a signature for any other message M'.

Also, there are (at least) two properties that digital signatures do not achieve on their own.

- Receiver authentication: Because a signature can be verified by anyone, on its own it will not tell Bob that he was the intended target of the message. If Alice wants to convey that Bob is the intended target, she should do so in the message contents M'.
- Thwarting replay attacks: If Bob has a digital signature, he can show it to others (e.g., a bank) as many times as he wants. To solve this issue, we will have to be clever in thinking about the format of the message M' – for instance, perhaps it should contain a unique serial number, so the bank can detect if the same check is being cashed twice.

2 Cryptographic Hash Functions

> Whether you're encrypting an email, sending a message on your mobile phone, connecting to an HTTPS website, or connecting to a remote machine through IPSec or SSH, there's a hash function somewhere under the hood. Hash functions are by far the most versatile and ubiquitous of all crypto algorithms.
>
> — Jean-Philippe Aumasson, Serious Cryptography (Chapter 6)

There are five parts to this course.

1. Authenticity without interaction
2. Currencies without centralization
3. Consensus without trust
4. Data analysis without data sharing
5. Crypto and society

Last time, we saw an introduction to cryptography, and an introduction to cryptocurrencies.

Our starting point into cryptocurrencies is to build a digital version of a banking system, where people can transfer money to each other electronically over the Internet without the need for a single trusted banking authority.

In the physical world, when Bob receives a check from Alice, he may be suspicious that Alice is trying to deceive him. He needs to be able to verify the following three properties.

1. Alice properly signed the check
2. Alice possesses $1 in her bank account
3. Alice does not **double spend** the money by writing a check to someone else at the same time

Last week, we started to explore a cryptographic primitive that will address property #1: building a digital version of signing a check. This primitive is (appropriately enough) called a digital signature.

Today, we will study a special function that is needed to build a digital signature and other crypto tools.

This special function is called a **cryptographic hash function**, and we will have many uses for it throughout the semester.

2.1 Representations of Data

Let's start by exploring how computers represent data. Throughout this course, we will typically represent data as abstractly as possible: as a sequence of bits of a particular length.

- A **bit** is a single 0 or 1 value. That is, a bit is an element of the set $\{0, 1\}$
- A **byte** is a sequence of eight bits. In other words, a byte is an element of the set $\{0, 1\}^8$ (Note that there is a one-to-one mapping between bytes and integers in the range 0-255.)
- A **bitstring** is an arbitrary-length sequence of bits. The set of all bitstrings of length n is denoted as $\{0, 1\}^n$. The set of bitstrings of any possible length is denoted as $\{0, 1\}^*$.

(We will typically, though not always, consider bitstrings that are a multiple of 8 bits; i.e., that can be decomposed into bytes with no "leftover" bits.)

Within Python, you can represent a value of type byte using a string with the letter b in front of it. For example, here is a byte that holds the ASCII value of the character A.

The hexadecimal, or base 16, representation of a number uses the symbols 0-9 and a-f to denote the values 0 through 15.

By convention, Python places a 0x prefix in front of a number to remind you that it is written in hex format.

The Python libraries binascii and PyCrypto provide commands to convert between raw bytes and numbers in bin, int, or hex formats.

Note that Python uses the prefix 0b to denote a number in binary format.

The base64 library and base58 library offer additional options.

2.2 Digital Fingerprinting

Let's rewind time and imagine that the date is currently November 30, 2007.

At that time, here were some of the candidates who were considered as strong candidates for the 2008 U.S. presidential election.

Here's the scenario:

- Alice claims to Bob that she can predict the outcome of the upcoming election. Specifically, she has an image file x on her computer containing a picture of the winner.
- Alice and Bob would like to make a $1 bet that she knows the correct answer.
- However, Alice does not want to reveal x to us right now – if she did so, Bob would know the result and could place a bet with other people.

- Instead, Alice would like to make the bet now, and reveal her answer after the 2008 election.

There is a clear problem here: if Alice won't reveal the image x until after November 2008, then how does Bob know that she knew the answer beforehand?

Question. Can we design a way to make this bet that provides:

- Confidentiality for Alice, in the sense that she doesn't yet have to reveal x
- Integrity for Bob, in the sense that he is convinced that Alice cannot cheat later when revealing x?

What would be great is if there was a function H that we could apply to the image x that:

- Can act as a "fingerprint" of the image file, in the sense that it uniquely specifies the image.
- Produces a small output, so that it's easy for Bob to save H(x) on his computer, even if x is large.

If we had such a file, then

- In 2007: Alice can give y=H(x) to Bob
- In 2008: Alice can give x to Bob

Bob can then compute H(x) himself and test whether it equals the fingerprint y that he received back in 2007. If so, then he knows that Alice did have that image in mind back in 2007.

2.3 Cryptographic Hash Functions

> Often called a cryptographer's Swiss Army knife, a hash function can underlie many different cryptographic schemes: aside from producing a document's digest to be digitally signed—one of the most common applications.
>
> Let us give example applications: code-signing systems, computer forensics, [and protecting] passwords.
>
> — Aumasson, Meier, Phan, and Henzen, The Hash Function BLAKE.

2.3.1 High-level Idea

- It should be public and deterministic, so everyone can compute it quickly, and yet
- It should be unpredictable and random-looking, so that nobody can understand how its outputs are connected to its inputs.

You can think of a hash function as a gigantic function that translates between two languages:

- The inputs that have some mathematical or English-language structure that makes sense to us.
- The outputs are a new kind of "foreign language" that is incomprehensible to all of us on Earth.

Moreover, the outputs have a fixed, known length, no matter how short or long the input messages are.

While this toy example used English-language words as the possible inputs, in reality we want a hash function to accept any data type. To be as generic as possible, we will consider the inputs and outputs to be bitstrings.

As a result, we can apply H to Alice's image file from our election example.

2.3.2 A Formal Definition

Using the language of bitstrings, we can now define a hash H as we stated previously: it's a single mathematical function that can accept inputs of any size and that always returns an output of a fixed length.

Definition. A hash function $H : \{0,1\}* \rightarrow \{0,1\}^n$ is an efficiently-computable function that

- Receives an input bitstring of arbitrary length, and
- Provides an output bitstring that contains a fixed number of bits n

Note that there is nothing hidden or secret here: the code of H is public knowledge and anyone can compute it.

The output H(x) is often called the **hash digest** corresponding to the input x.

2.4 Hash Function Security

Because H has infinitely many inputs and only finitely many outputs, it must be the case that there are **collisions** – that is, two inputs that map to the same output string.

But: we are going to insist that **collisions are computationally infeasible to find**, just like they would be in our motivating example of the randomly-generated foreign language.

This guarantee should hold even if we give the code of H to our adversary.

2.4.1 Meet Mallory

Here is our adversary in this course. Her name is Mallory, because she is malicious and doesn't follow the rules of any crypto protocol.

Mallory is intended to be a stand-in for any kind of real-world adversary who you might be concerned about: your internet service provider, cloud provider, government, or anyone else who might want to ruin your day.

Mallory has a substantial (though not infinite) amount of computing power at her disposal.

She's also very smart. Specifically, we must assume that she is a better cryptanalyst than we are. This is to account for **Schneier's law**.

> Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break.
>
> — Bruce Schneier

Even though Mallory is mathematically savvy and computationally powerful, we insist that she must still not be able to "break" our hash function H.

To make this claim mathematically precise, we need to specify precisely the problem that Mallory cannot break.

2.4.2 Formal Definition of Security

In fact we will define multiple such problems. As we go down the list:

- the problems become easier and easier to break,
- so if Mallory still cannot break them, then the hash function H must be really strong.

Definition. A cryptographic hash function H satisfies the following properties.

- **One wayness**, aka **preimage resistance**: if we sample $y \leftarrow \{0,1\}^n$ uniformly at random from the set of all bitstrings of length n, then it is practically infeasible for Mallory to find any preimage x such that $H(x) = y$. (Note: remember that there are infinitely many such preimages, yet Mallory cannot find even a single one!)
- **Second preimage resistance**: given a "randomly" chosen x, it is practically infeasible for Mallory to find another x′ such that H(x)=H(x′) but x≠x′. (Note: this statement is not mathematically precise, since it isn't possible to sample uniformly at random from an infinite space like {0,1}* But I will ignore this issue since we will not focus on second preimage resistance in this course.)
- Collision resistance: it is practically infeasible for Mallory to find two different inputs x1 and x2 such that H(x)=H(x′). (Note: if a function is collision resistant then it must be one way, but the converse is not necessarily true. Do you see why that is?)
- Puzzle friendliness: (this property is complicated to describe; we will come back to it later)
- Random oracle: H acts like a randomly-generated truth table would.

> If an adversary [Mallory] has not explicitly queried the oracle on some point x, then the value of H(x) is completely random… at least as far as [Mallory] is concerned.
>
> — Jon Katz and Yehuda Lindell, Introduction to Modern Cryptography

Note that there is a limit to how infeasible it is to find a collision. After all, we already said that collisions must exist. And the output space is finite, so if we have enough time and computing power, it must be possible to find a collision.

Exactly how difficult can we hope that the problem might be? That is, even with the best possible hash function, what is a blunt and **brute force attack** that will always succeed?

2.5 The Birthday Problem

Let's step away from hash functions for a moment, and look at a seemingly-unrelated question. Let K be the number of people in this classroom.

Question. What is the probability that two people in the room have the same birthday?

(For the purpose of this question, suppose that everyone's birthday is sampled uniformly at random from the set of 366 possible choices. This is not a valid assumption for many reasons, e.g., leap years, but let's go with it for now.)

It might intuitively seem like the answer should be $\frac{K}{366}$.

But that is incorrect! It would be the answer to a different problem: What is the probability that at least one of N students in this classroom has the same birthday as Prof. Varia does?

The distinction between these two questions (and their corresponding answers) is called the **birthday paradox**.

The answer to the original question depends on the number of pairs of people in the classroom.

The answer to the original question depends on the number of pairs of people in the classroom.

- If birthdays are sampled uniformly at random, each pair has a 1366

chance of having the same birthday.

- There are $\binom{K}{2}$ pairs of people in the room.

So the answer to the question is closer to $\frac{\binom{K}{2}}{366} \approx \frac{K^2}{2 \cdot 366}$ than it is to $\frac{K}{366}$.

But $\frac{\binom{K}{2}}{366}$ is not exactly right either.

The reason is the principle of inclusion-exclusion: we would be overcounting in scenarios where many people have the same birthday.

Instead, it is easier to think about the converse problem. Suppose $K = 23$, and let's imagine that the K people walk into the classroom, one at a time.

What is the probability that all of them have different birthdays?

- The first person certainly has a different birthday than everyone else in the room (since there is nobody else in the room yet).
- The second person has probability $\frac{365}{366}$ of having a different birthday than the first person.
- Conditioned on the first two people having distinct birthdays, the third person has probability 364/366 of having a different birthday than both of them.
- ...and so on, until the 23th person to enter the room has a 344/366 probability of having a birthday that is different than the other 22 people already in the room.

As a result, the overall probability that everybody's birthday is distinct is small:

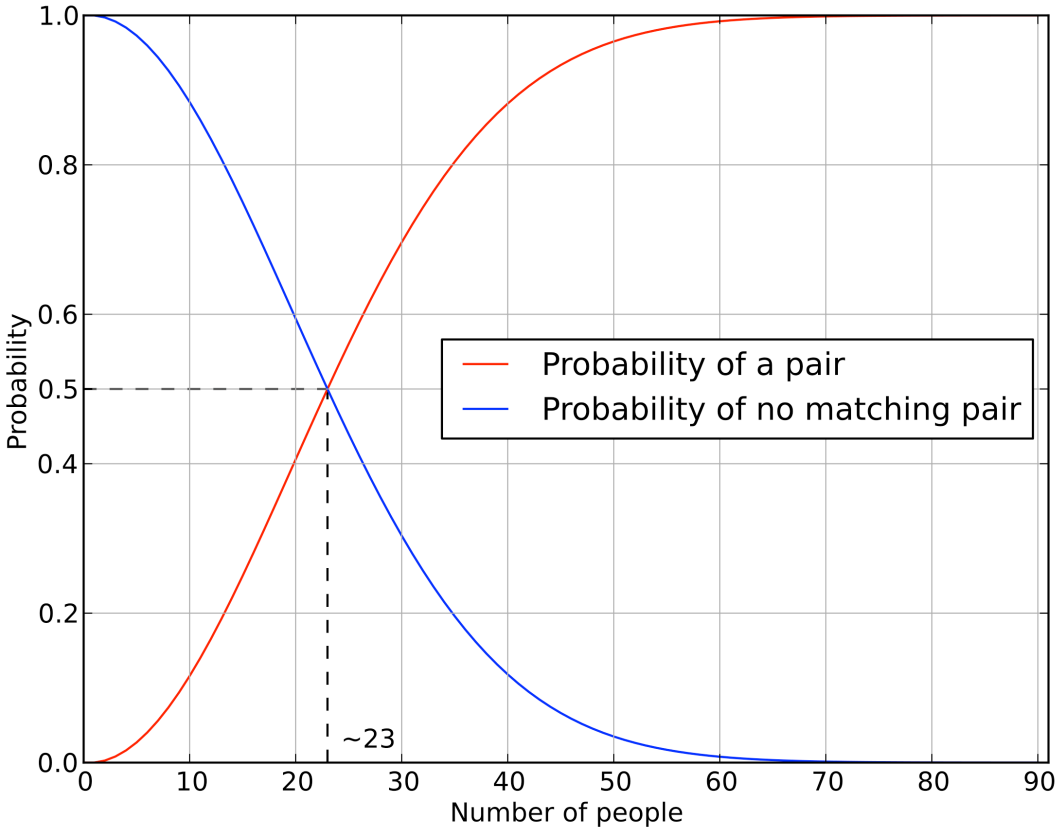
$$1 \cdot \frac{365}{366} \cdot \frac{364}{366} \cdot \ldots \cdot \frac{344}{366}$$

```
import math
math.prod(range(344,367)) / (366**23)

# output: 0.49367698818054007
```

As a result, if there are 23 people in the classroom, then it is more likely than not that two of us have the same birthday.

Here's a graph that shows the probability of two people having the same birthday in a room of N people.



Generalizing from the specific case of birthdays (where there are N=366 options): when drawing with replacement from a set of size N, then:

$$E[\#\text{ of attempts until the first collision}] \approx 1.25\sqrt{N}.$$

This approximation works well even for fairly small choices of N, such as $N = 366$. And it works really well for larger N.

```
import math
1.25 * math.sqrt(366)

# output: 23.91390808713624
```

Moreover, the distribution is tightly correlated around this expected value. There is less than a 15% chance that:

- the first collision has already occurred when drawing $\frac{1}{2} \cdot 1.25\sqrt{N}$

samples.

- the first collision has not yet occurred when drawing $2 \cdot 1.25\sqrt{N}$ samples.

2.6 Comparing Preimage and Collision Resistance

Returning back to hash functions: the key insight from the birthday problem is that:

The best-possible security that we can hope for a hash function $H : \{0,1\}^* \to \{0,1\}^n$ to achieve is related to the **length of the output space** n.

Example. Suppose that we build a hash function whose output length is $n = 3$ bits. Let's try to break one-wayness and collision resistance using a simple guess-and-check approach.

To break one-wayness for (say) $y = 000$, suppose we just select inputs x at random and compute the hash function until we find an input where $H(x) = y$.

- Each sample of x will satisfy the equation with probability 1/8.
- So in expectation, we will find an input x such that $H(x) = y$ after 8 attempts.
- There is a chance that we get unlucky and it could take much longer. Probabilistically: the number of attempts until the first success follows the distribution of a geometric random variable.

Now let's try to break collision resistance. Once again, suppose that we aimlessly select inputs x at random and build a table of all $(x, H(x))$ pairs until we find two inputs with the same digest.

- In the worst case, we can select 9 inputs, and by the pigeonhole principle two of them must map to the same output.
- In expectation, the birthday bound says that the expected number of trials is around $1.25 \cdot \sqrt{2n} \approx 4$.

2.6.1 Choosing Larger n

So in order to have a secure hash function, it is necessary (though not sufficient) to choose a large output length n. A common choice in cryptography is to choose $n = 256$ bits.

Extrapolating from the analysis above, our brute-force attack succeeds at breaking:

- One-wayness with good probability by an attacker who tries around 2^{256} inputs.
- Collision resistance with good probability by an attacker who tries around $\sqrt{2^{256}} = 2^{128}$ inputs.

These are both big numbers. And they're very different scales of big-ness. Remember that $\sqrt{2^{256}}$ is not two times as big as $\sqrt{2^{128}}$. It is instead $\sqrt{2^{128}}$ times as big!

In fact, both of them are so big as to be practically impossible for anyone to execute, with the computing power available today or in the near future.

To give a sense of how big the number $\sqrt{2^{128}}$ is:

- The cryptocurrency Bitcoin gives people money to run the hash function SHA-256 over and over and over again.
- The world has collectively devoted so much computing power to this task that we now run around 1 billion trillion SHA-256 hashes per second (source: blockchain.info). So that's around $10^{21} \approx 2^{70}$ computations per second.

To compute the hash function 2128 times, it would take the entire world (with our current computing power) about 2128/270=258 seconds.

- As a useful heuristic: there are about 2^{25} seconds in one year.
- So this calculation would take $2^{58-25} = 2^{23} \approx 8$ million years

(Admittedly, this calculation discounts the effect of Moore's law. But hopefully it gives you a sense of the vast scale of computing power involved here.)

The number $\sqrt{2^{256}}$ is much, much larger than this. To give you a sense of its scale: suppose that Mallory decided to try running the hash function on $\sqrt{2^{256}}$ inputs, say by running a for loop to try all inputs from

```
from binascii import hexlify
print(hexlify(32 * b'\x00')) # taking 32 copies of the byte value 0 (printed in hexadecimal
format)

# output: b'0000000000000000000000000000000000000000000000000000000000000000'
```

to this:

```
print(hexlify(32 * b'\xff')) # taking 32 copies of the byte value 255 (printed in
hexadecimal format)

# output: b'ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff'
```

Then:

- Running this for loop on a modern computer would require more time than the expected remaining length of the universe.
- And simply flipping all of these bits back and forth in RAM would consume approximately the energy of the sun.

2.7 Feasibility of Building a Hash Function

So far, we have talked about the best-possible security that a hash function can hope to achieve.

But is this hope actually achievable?

The good news: the worldwide crypto community believes the answer is yes!

There is some bad news, however.

- I cannot mathematically and analytically prove to you that any cryptographic hash function actually meets the definition of one-wayness or collision resistance.
- That would require fundamental breakthroughs in computer science like proving $P \neq NP$.

So instead: we evaluate hash functions empirically, based on how well they hold up against the worldwide cryptanalysis community.

Informally, low-level crypto building blocks like hash functions must satisfy three properties:

- **Simple**: It must be implementable in a small number of lines of code, and run quickly on modern computers (e.g., millions, billions, trillions, … of times per second).
- **Makes no sense**: It survives the gauntlet of cryptanalysis attempts by hundreds of cryptanalysts over many years.
- **Simple to see why it makes no sense**: There is some mathematical reason to believe that nobody will come up with a clever cryptanalysis algorithm tomorrow, in order to adhere to Schneier's law.

In particular, we do **not** rely on security by obscurity – that is, simply making the code of a hash function so complicated that nobody can understand it.

Instead, we want the hash function to be hard to break even by people who understand exactly what it is doing.

In more detail, all cryptographic primitives must satisfy **Kerckhoffs's principle**.

- The system must be practically, if not mathematically, indecipherable;
- It should not require secrecy, and it should not be a problem if it falls into enemy hands;
- It must be possible to communicate and remember the key without using written notes, and correspondents must be able to change or modify it at will;
- It must be applicable to telegraph communications;
- It must be portable, and should not require several persons to handle or operate;
- Lastly, given the circumstances in which it is to be used, the system must be easy to use and should not be stressful to use or require its users to know and comply with a long list of rules.

[Source: Auguste Kerckhoffs, La Cryptographie Militaire, 1883 (more details available on Wikipedia)]

2.8 Hash Function Constructions

The U.S. National Institute of Standards and Technology (NIST) develops standards for all cryptographic building blocks: hash functions, digital signatures, encryption schemes, and more.

- Officially, these standards are only required for use within the U.S. federal government.
- Unofficially, their standards are often used in commercial applications around the world.

To date, NIST has developed three standards for hash functions (and also subsequently revoked one of them). These algorithms are called the **Secure Hash Algorithms**, or SHA for short. There have been three iterations of hash functions: SHA-1, SHA-2, and SHA-3.

2.8.1 SHA-1

SHA-1 was initially developed in 1995 by the U.S. National Security Agency. It had an output length of n=160 bits, so by the birthday bound at best we can hope that it takes 280

time to find a collision.

But through a combination of incredibly clever math to design a more clever collision-finding algorithm and a lot of computing power to execute it, researchers have broken SHA-1 and you should never use it.

Example. Here are two PDF files.

- good.pdf
- bad.pdf

It turns out that they have the same hash digest using the SHA-1 hash function.

```
from hashlib import sha1

# read the contents of two files: good.pdf (a check mark) and bad.pdf (an X)
with open('images/good.pdf', 'rb') as file:
    good = file.read()

with open('images/bad.pdf', 'rb') as file:
    bad = file.read()

# verify that the strings are different
assert(good != bad)

# outputs are 40 hex characters, or 160 bits in length
hashGood = sha1(good).hexdigest()
hashBad  = sha1(bad).hexdigest()

# observe that the hash function outputs are the same
print(hashGood)
print(hashBad)
assert(hashGood == hashBad)

# output:
# d00bbe65d80f6d53d5c15da7c6b4f0a655c5a86a
# d00bbe65d80f6d53d5c15da7c6b4f0a655c5a86a
```

2.8.2 SHA-2 and SHA-3

So which hash functions should you use?

- SHA-2 was initially developed in 2001, also by the U.S. National Security Agency. It is the most popular hash function algorithm today. It offers a choice of four possible output lengths: 224, 256, 384, or 512 bits.
- SHA-3 was standardized in 2015 after a multi-year open competition. Its goal was to design a hash function in a fundamentally different way as SHA-2, in order to have a "failsafe" plan in case someone finds a collision in SHA-2.

To the best of our collective knowledge as a crypto community, for both of these functions **are secure**.

In particular, the best known algorithms to break collision resistance require about $2^{\frac{n}{2}}$ work to break, when the hash function is set to have an output length of n bits.

```
from Crypto.Hash import SHA256

# output is 64 hex characters, or 256 bits in length
print(SHA256.new(b'Hello world!').hexdigest())

# output: c0535e4be2b79ffd93291305436bf889314e4a3faec05ecffcbb7df31ad9e51a
```

2.8.3 Application to File Integrity

We will see many uses of hash functions throughout this course. For example, next time we will see how hash functions can be used within the construction of digital signatures.

For today, let's go back to the fingerprinting application from the beginning of the lecture. Suppose that Alice wants to store a large file f on a cloud service like Dropbox or Google Drive. The file is important to Alice, so when she retrieves it later, she wants to be able to check that the file hasn't been altered or corrupted on the cloud.

How can Alice do this?

Here's one common approach:

- Compute the hash of the file h=SHA256(f) before you upload the file f to the cloud. Store h on your own computer. (Remember that h is small, only 32 bytes in size.)
- When you later download a file f', check that $\text{SHA256}(f')?=h$ in order to verify that f' is indeed the same as the file f that you previously uploaded.

This approach is both simple and secure.

- Modern hash functions are very quick to compute, even on large files that are gigabytes in size.
- Due to collision resistance, even if Mallory is running the cloud provider, she cannot find a new file f' that is different from Alice's original file f and yet still hashes to the same string h.

2.9 Merkle Trees

What if Alice wants to upload many files $f_1, f_2, ..., f_k$ to the cloud?

- It would be tedious and space-consuming if she had to store on her computer one hash digest for each file.
- Alternatively she could take the hash of all files $h = H(f_1, f_2, ..., f_k)$. But then she would only be able to verify when she downloaded the entire set of files.

Fortunately, there is another approach that still allows Alice to verify one file at a time. Before uploading the files, Alice must construct a **Merkle tree** as follows.

- Take the hash of each individual file.

- Then, take the hash of each pair of hashes.
- Only store the hash at the root of the tree.

(This data structure is named after cryptographer Ralph Merkle.)

Source: Alin Tomescu, Decentralized Thoughts blog

The nice part about Merkle trees is that:

- Alice only has to store one 32-byte hash digest $h_{1,8}$ that corresponds to, and binds together, the entire set of files $f_1, f_2, ..., f_k$.
- Later, the cloud provider Mallory can send one file and prove to Alice that this file is contained "within" the Merkle tree.

To construct a proof that (for example) a claimed file f_*^3 is in the set, the cloud provider Mallory sends the following $\log(k)$ hash digests to Alice.

Source: Alin Tomescu, Decentralized Thoughts blog

I put an asterisk on the received file f_*^3 and the received hash digests $h_4^*, h_{1,2}^*, h_{5,8}^*$ because Alice does not yet know that the cloud provider Mallory is telling the truth. That is, these might potentially be different than the variables that Alice used when she initially constructed the Merkle tree.

To test whether the received file f_*^3 is correct, Alice can "fill in the blanks" of the path of the Merkle tree from the leaf up to the root.

$$h_3^* = H(f_3^*)$$

3 Digital Signatures

> A major concern in the subject [of cryptography] is that of authentication. How do you know some data is correct (data authentication), and how do you know an entity is who they claim to be (entity authentication).
>
> — Nigel Smart, Cryptography Made Simple

3.1 Overview of Digital Signatures

For the rest of today, we will explore the design of digital signatures. As the name suggests, this primitive is the digital analog to a signed piece of paper. In cryptocurrencies, signatures allow us to know for sure that Alice is the person who signed the message "I Alice hereby send $1 from my account to Bob."

A digital signature allows Alice to append a signature σ to a message M that satisfies two requirements (which we describe informally for now).

- **Correctness**: Everyone in the world can verify whether Alice created the signature σ.
- **Unforgeability**: The signature σ is bound to the message M. Nobody besides Alice could use it to forge a signature for any other message M'.

Also, there are (at least) two properties that digital signatures do not achieve on their own.

- **Receiver authentication**: Because a signature can be verified by anyone, on its own it will not tell Bob that he was the intended target of the message. If Alice wants to convey that Bob is the intended target, she should do so in the message contents M.
- Thwarting replay attacks: If Bob has a digital signature, he can show it to others (e.g., a bank) as many times as he wants. To solve this issue, each message M can contain a unique serial number, so the bank can detect if the same check is being cashed twice.

3.1.1 Cryptographic Keys

To satisfy the unforgeability goal, Alice must have something that the rest of the world does not. We call this special something a "**secret key**" or "**private key**" (the two terms are synonymous).

Informally, a digital signature allows Alice to "lock" her message using her secret key.

To satisfy the correctness goal, there is a corresponding **public key** that anyone can use to verify whether a message was previously "locked" by Alice.

In more detail, a secret key is a long, random sequence of bytes. One common choice is to pick a key that is 32 bytes (aka 256 bits) in length. For example, here is a secret key:

```
# must install PyCryptoDome with, e.g., "pip install pycryptodome"
from Crypto.PublicKey import ECC
secretKey = ECC.generate(curve='P-256')

# warning: this is just an illustrative example
# do not export a secret key that you are using for any legitimate purpose!
print(secretKey.export_key(format='PEM'))

# output:
'''
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgvZ3Kis3judSdWDcK
W/WS2bnHqzi3Ip94rT+Mg3qi13ShRANCAARHh3A+8Kn9OIFwzuC0jhS59o/8sL/3
PAR7PqD2dbTbj4OUOMg5q/R1WB747UEfB+ryO8ajoaY9W6mRJd9urzjV
-----END PRIVATE KEY-----
'''
```

From a secretKey, it should be possible to produce the corresponding publicKey.

But we want this operation to be one way: nobody can go in the other direction of publicKey ⟶ secretKey.

```
publicKey = secretKey.public_key()

# this key is safe to share with the world
print(publicKey.export_key(format='PEM'))
```

```
# output:
'''
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAER4dwPvCp/TiBcM7gtI4UufaP/LC/
9zwEez6g9nW024+DlDjIOav0dVge+O1BHwfq8jvGo6GmPVupkSXfbq841Q==
-----END PUBLIC KEY-----
'''
```

As the name suggests, **only Alice knows the key**; she never shares her secret key with anyone. (Hence my warning in the code snippet above).

This key is long enough that it is incredibly unlikely that anyone else in the world will be able to guess it by chance. Just like with hash functions, you should interpret:

- "long enough" to mean **256 bits** or more in length, and
- "unlikely to collide by chance" to mean that a brute force attack would take the energy of the sun, and even the best known cryptanalytic attacks will take decades, centuries, or longer to complete.

As a result, **knowing the key identifies Alice**, in the sense that:

- if you already know that a particular public key pk belongs to Alice, and
- you interact with a stranger on the Internet who can has the ability to sign a new message that can be verified with Alice's public key, then this Internet stranger must be Alice!

(Well, or someone who has compromised Alice's computer and stolen her keys…)

3.2 Defining Digital Signatures

Remember that a digital signature must satisfy two requirements: correctness and unforgeability.

To understand the unforgeability guarantee better, it helps to introduce our adversary Mallory to the story.

Suppose that Alice is trying to sign a message M. But, there is a meddler-in-the-middle, called **Mallory**, who intercepts the message en route.

We want to ensure that any attempt by Mallory to tamper with the messages from Alice to Bob will be detected.

And we want unforgeability to hold no matter how many messages Alice sends. Even if she sends 1000 messages $M_1, M_2, ..., M_{1000}$, Mallory cannot forge any other message that Alice hasn't sent.

Furthermore, it shouldn't matter which messages Alice sent. That is, for any choice of the 1000 $M_1, M_2, ..., M_{1000}$ that Alice sent, Mallory cannot create a signature for message 1001.

3.2.1 Formal Definitions

We are now ready to define formally the definition of a digital signature scheme.

Definition. A digital signature contains three algorithms:

- **KeyGen**: takes no input, produces a secret key sk and a public key pk
- **Sign(sk, M)**: produces a signature σ
- **Verify(pk, M, σ)**: returns true or false

All three algorithms should be efficiently computable. Moreover, the digital signature must satisfy two requirements: **correctness** and **unforgeability**.

Correctness means that for any secret/public keypair produced by KeyGen, it is the case that any signature made via the Sign algorithm is subsequently accepted by Verify.

We formalize the **unforgeability** guarantee by considering a hypothetical game involving

- Alice, who has run KeyGen to produce a secret/public keypair, and
- Mallory, who only knows Alice's public key pk

This game proceeds in two phases.

- In the first phase, Mallory sends a message M, and Alice responds with σ=Sign(sk,M). Mallory can repeat this step as often as she wants.
- In the second phase, Mallory sends a message M* of her choice along with a claimed signature σ*.

Mallory wins the game if:

1. Her signature verifies, in the sense that Verify(pk,M*)=True.
2. The message is new, meaning that M* is not a message that she previously sent in the first phase for Alice to sign.

Any digital signature scheme that satisfies this definition is said to be "**existentially unforgeable under a chosen message attack**," often abbreviated EU-CMA.

In words, this means that no matter how many signed messages that Mallory has seen Alice make in the past, and no matter what the message contents were, it is practically impossible for Mallory to forge a new message that Alice never signed herself.

3.3 Constructing Digital Signatures

It is helpful to have a definition in hand before we try to construct a digital signature, so that we have a way to test whether a candidate construction is good.

Let's try to construct a digital signature. It turns out that this is challenging to do. Remember that we need the following properties to hold:

- Given a public key, it is difficult to find the signature σ corresponding to a message M.
- Given a public key, it is easy to verify whether the signature σ

corresponding to a message M is correct, if someone else has already found σ for you.

So: we need to find a math problem that is difficult to solve but easy to verify.

Unfortunately, we do not know for certain whether any such math problems exist! If you can find one, the Clay Math Institute will give you a prize of $1 million. (This is called the P vs. NP question.)

So we do the next best thing: build signature schemes from math problems that we believe to have this property of "difficult to solve but easy to check." One such problem is **factoring**.

- Given a composite integer N that is the product of two primes, it is difficult to find the primes p and q such that N=pq.
- If I told you the prime factors, you could easily multiply them together to verify whether $pq? = N$.

3.3.1 RSA Signatures

Rivest, Shamir, and Adleman realized that this math problem can be used as the basis of a digital signature scheme. It is named RSA in their honor. It works using modular (or clock) arithmetic.

Here is a slightly oversimplified (and incorrect) version of their signing algorithm.

KeyGen has no inputs, and does the following:

- Samples two large prime numbers p,q and sets N=pq.
- Chooses two exponents d and e with a special feature we will discuss next.
- Outputs the secret key (p,q,d), and the public key (N,e).

The remaining two operations use modular arithmetic, or clock arithmetic.

As a reminder: when given two integers A and B, the notation AmodB means to calculate the remainder that occurs when you divide A by B.

Here's our first attempt at defining the rest of the RSA algorithm.

- Sign(sk, M) = Ma mod N
- Verify(pk, M, σ) returns true if and only if $o^e = M \bmod N$

Why does this (almost) work? It turns out that:

- If you know the factorization of N , then it is easy to find two exponents d and e that "cancel out" in the sense that (xd)e=(xe)d=x for any value x.
- On the other hand, if you don't know the factorization of N, it is not possible to find the value d that "cancels out" e.

There's a bug in the current construction though. Suppose Alice gives Mallory the signatures of two messages M_1 and M_2:

$$\sigma_1 = M_1^d \bmod N, \sigma_2 = M_2^d \bmod N$$

From this, Mallory could produce the signature corresponding to the message $M^* = M_1 \cdot M_2 \bmod N$ because:

$$\text{Sign}(sk, M^*) = (M_1 \cdot M_2)^d \bmod N$$

$$= M_1^d \cdot M_2^d \bmod N$$

$$= \sigma_1 \cdot \sigma_2 \bmod N$$

Importantly: the last equation is something Mallory can calculate even without knowing sk.

This is our first example of a cryptanalysis attack! Even though a forgery would take a lot of time if you tried to factor N, we found an alternative approach that worked much faster.

To fix this problem, the real RSA signature scheme also uses a hash function H.

- KeyGen: generate two primes p, q and calculate N. Then find two exponents e and d that "cancel out." The secret key is sk=(p,q,d) and the public key is pk=(N,e).
- $\text{Sign}(sk, M) = H(M)^d \bmod N$
- $\text{Verify}(pk, M, \sigma)$ returns true if and only if $\sigma^e = H(M) \bmod N$

In this new construction, the Sign algorithm starts by applying a hash function H to the message M. The hash function breaks the algebraic structure that caused the problem above.

This construction does satisfy existential unforgeability under a chosen message attack, as far as we know. (Explaining why this is the case is out of scope for this class; if you're interested to see the math, it's covered in CS 538.)

As another benefit: the hash function allows us to sign a message M of arbitrary length, since H maps any message onto a fixed-length number that we can use as the base for the subsequent exponentiation.

3.4 A Smaller Signature Scheme

While RSA signatures are sometimes still used in practice, it's rare to see them today.

The reason for this is performance (not security).

- RSA isn't the fastest possible algorithm to complete. Signing and verification take around 1 millisecond on a modern laptop or phone, give or take.
- RSA keys and signatures are somewhat larger than desired – about a few thousand bits long. This is because factoring is a hard problem, but it's not a really hard problem, so we have to pick really big numbers for it to be infeasible by modern computers.

That may not sound big, and indeed a single signature is small. But they add up quickly.

Recall that digital signature schemes have a tricky property: from the public key it is impossible to create a digital signature but easy to verify if someone else has done so.

As a result, we must build signature schemes from math problems that we believe to have this property of "difficult to solve but easy to verify."

Instead, the most common standard is called the **digital signature algorithm**, or DSA. Often this algorithm is instantiated using a mathematical object called an "**elliptic curve**," in which case we give it the name **elliptic curve digital signature algorithm**, or **ECDSA**.

We will discuss how it works next time.

3.5 The Public Key Infrastructure

Remember our original goal: we want Alice to be able to digitally sign the message "I Alice hereby send $1 to Bob", so that:

- Bob himself can become convinced that Alice was the signer, and
- Bob can show it to a bank, who will also be convinced and will credit Bob's account.

We previously said that Alice should broadcast her public key with the world.

Question. How can she securely tell the world "I am Alice and my public key is pk"? Or more precisely: how can she stop Mallory from also broadcasting the message "I am Alice and my public key is pk_*" for a key pk_* chosen by Mallory?

There is a chicken-and-egg issue here:

- In general, Alice can use signatures to convince the world that she's the creator of any message, after people have associated pk with Alice's identity.
- But to start this process, someone needs to go physically check Alice's real-world identity. Bob could do this himself, but if Alice and Bob need to meet in person then they could just exchange a physical check.

Instead, in the Internet we rely on companies called **certificate authorities**. These companies are paid to check and then attest the binding between Alice's identity to her public key.

The resulting set of bindings is often called the **public key infrastructure**, or PKI. You can think of a PKI as similar to a telephone directory, just for cryptographic public keys rather than telephone numbers.

For example, the website bu.edu uses digital signatures so that all visitors know that the data is authentic and comes from BU.

If you click on the 🔒 icon in your web browser, it shows that the company "Sectigo" is asserting that the public key B5 69 6B 98 BA EF EB 09 … belongs to "Trustees of Boston University".

The PKI is a powerful tool for two reasons:

- Now Bob can become convinced that a pk belongs to Alice, without the two of them physically needing to meet.
- The signatures are transferrable. Bob can show the digital check to the bank, and even they will be convinced that Alice was the signer.

However, the system is heavily reliant on the certificate authority companies always telling the truth. This is how the Internet works today: if the certificate authority cheats, it could pretend to be bu.edu and our laptops and phones would be duped.

As a result, some of the work we will do in Parts 2 and 3 of the course involves trying to build systems that do not rely on the binding of someone's public key to their real-world identity, so we don't need to trust a PKI.

3.6 A Smaller Signature Scheme

While RSA signatures are sometimes still used in practice, it's rare to see them today.

The reason for this is performance (not security).

- RSA is fast, but not that fast. Signing and verification take around 1 millisecond on a modern laptop or phone, give or take.
- RSA keys and signatures are small, but not that small. As shown in the image above, the RSA keys and signatures used by bu.edu are 2048 bits (or 256 bytes) long.

This is because factoring is a hard problem, but it's not a really hard problem, so we have to pick very large numbers for it to be infeasible by modern computers.

Recall that digital signature schemes have a tricky property: from the public key it is impossible to create a digital signature but easy to verify if someone else has done so.

As a result, we must build signature schemes from math problems that we believe to have this property of "difficult to solve but easy to verify."

Let's find another math problem that satisfies this property.

3.7 The Discrete Logarithm Assumption

Here is another hard math problem: taking logarithms.

...Wait, hang on, this doesn't sound right. Let's check wheether this is actually a hard math problem.

Question. If I tell you that I have taken the number 5 to some power x and the result is 125, can you determine the power x such that $5^x = 125$?

Taking logarithms to find $x = \log_5(125)$ is easy over the real numbers $\mathbb{R}$.

But: it turns out though that for certain finite spaces, it is believed to be practically infeasible to compute discrete logarithms.

3.7.1 Option 1: Modular arithmetic

If we apply modular arithmetic, then the logarithm question becomes much harder.

Consider the following function: $f(x) = 5^x \bmod 7$.

Question. Can you find a value x such that $5^x = 3 \bmod 7$?

This does not seem as simple to calculate. The good news is that calculating this function in the forward direction is easy: all we have to do is multiply. So, here is the truth table of the function f. (Remember that this took a lot of computational effort to build.)

With this table, now can you solve the question from above?

It turns out the act of exponentiation modulo a prime number is:

- Invertible. In fact it is a permutation, because each number between 1 and 6 occurs exactly once as a possible output.
- Computationally difficult to invert… at least if we choose a prime modulus that is large enough. (There is one additional caveat: for some technical reasons that I will skip, we must take only the subgroup of quadratic residues, i.e., only half of the truth table for even values of x.)

3.7.2 Option 2: Elliptic curves

There is another way to make "exponentiation" easy but "logarithms" hard. It's… admittedly going to look strange.

I'll give you a brief overview of how it works, but fair warning upfront: you shouldn't worry about the math of how it works. It's a bit beyond the scope of this course. You should focus mostly on the fact that it can work at all.

Okay, here's the idea. Rather than multiplying numbers, we're going to make a new kind of arithmetic where we can "multiply" points in space.

- Take a cubic equation $y^2 = x^3 + ax + b \bmod p$ – this is called an elliptic curve.
- Consider the set of points on this degree 3 curve.
- We can "multiply points" by imposing the rule that $P \cdot Q \cdot R = 1$. (Note that this has absolutely nothing to do with taking the exponent of the (x, y) coordinates of any of the points.)
- We can "exponentiate a point" using a tangent line, so $P \cdot Q = 1$.

This is a very strange operation! Don't worry too much about the math details here. The most important thing to remember here is that it also leads to a hard math problem, where exponentiating is easy but the inverse is practically impossible.

- Elliptic curve discrete logarithm assumption: Given two points P and Q such that $P^x = Q$, it is computationally infeasible for anyone to find the discrete logarithm x.

(Note: all of the exponentiations and logarithms in today's lecture are done modulo a prime number, but I'm going to stop writing that explicitly from now onward.)

In fact, this problem is really hard to solve, even harder than factoring (as far as we know). As a result:

- The act of computing a digital signature is faster: closer to microseconds (10^{-6} seconds) than milliseconds (10^{-3} seconds).
- We can get away with smaller keys and signatures: in the tens of bytes rather than hundreds of bytes in size.

This difference may not sound big, and indeed it probably doesn't matter if you only need to perform a single digital signature. But we're going to use signatures a lot in this class, and the performance savings will add up quickly.

3.8 Diffie-Hellman Key Exchange

We have a new tool in our crypto toolbox. What can we do with this?

Here's one option: we can use the hardness of discrete logarithms as the basis for another type of public key cryptography.

Taking a slight detour from digital signatures, let's go back to a question I raised in the first lecture. Suppose you want to send data back and forth to a public website, like kaggle.com, in such a way that nobody else can read or tamper with its contents.

We already know that digital signatures can provide evidence of tampering. But what about message confidentiality?

To solve this problem, you need symmetric key encryption. This is the digital equivalent to a combination safe – it allows for two computers to exchange messages confidentially, as long as they have a shared secret that the two of them know but nobody else in the world does.

It is the basis for end-to-end encrypted secure messaging systems like Signal and WhatsApp.

In order to encrypt data, your computer and the kaggle.com server both need to have a secret key that nobody else in the world does.

But how can they do that? If you could physically meet, then you could exchange secrets. But we want to enable secure communication over the Internet, which is a public network where most pairs of people have never met before.

In the digital world, they need a secure way to exchange secrets in plain view.

> The Internet is just the world passing notes in a classroom."
>
> — Jon Stewart

Amazingly, this can be done!

Based on the hardness of the discrete logarithm problem, here is an approach that you and the kaggle.com server can take.

- You can choose a secret a that you keep to yourself, and send $A = g^a$ to the server. (Remember, nobody can take A and reconstruct a… not the server, or anyone else!)
- The server can choose a secret exponent b and send $B = g^b$ to you.

Now, there is a shared secret that both of you can compute: $s = g^{ab}$.

- You can take the public value B and raise it to your secret exponent a to compute $B^a = \left(g^b\right)^a = s$
- The server can do the opposite: $A^b = g^{ab} = s$

Moreover, nobody else can compute this secret! Other people hear A and B. But they don't know – and cannot compute – either secret exponent. So as far as we know, they have no way to compute s.

(Slight caveat: the last sentence does not actually follow from the one before it. Nevertheless, we do believe that it is a true statement too.)

This ingenious protocol was discovered in the late 1970s by Whitfield Diffie and Martin Hellman, and it is named **Diffie-Hellman key exchange** in their honor. It was based on concepts developed earlier by Ralph Merkle... the same person who invented Merkle trees.

3.9 Constructing a Digital Signature Algorithm

Returning back to our goal: how can we form a digital signature algorithm based on the fact that discrete logarithms are hard?

Remember that a signature scheme needs a secret key and public key, such that it is easy to go from the secret to the public key, but hard to go in the other direction.

Based on the **zero knowledge proof**, here's an idea for $\mathbf{KeyGen}$.

- Secret key: sample an exponent x uniformly at random.
- Public key: choose a base g, and publish $y = g^x$ as the public key.

This satisfies our criteria.

- Starting from the secret key, we can calculate the public key through exponentiation, which is simple to do.
- But going in the other direction requires taking a discrete logarithm, which is believed to be hard.

There are a few digital signature schemes that can be designed using keys of this form.

- One of them is a NIST standard called the **digital signature algorithm**, or DSA. If you use an elliptic curve group to build it, then we refer to the construction as the **elliptic curve digital signature algorithm**, or ECDSA. (Note: you will explore ECDSA signatures in this week's homework.)
- Another digital signature scheme, which is similar in style to DSA but slightly simpler to understand, is called **Schnorr signatures**.

I reproduce the construction of Schnorr signatures below; it is adapted from the presentation on Wikipedia. Don't worry too much on the math details; the main ideas are the same as the zero knowledge proof we constructed earlier.

3.9.1 Schnorr signatures

After generating keys as described above, here's how to sign a message M:

- Choose a random exponent k from the allowed set.
- Let $r = g^k$.
- Let $e = H(r, M)$, where H denotes a cryptographic hash function.
- Let $s = k - xe$.

The signature is the pair $\sigma = (s, e)$.

To verify the claimed signature $\sigma = (s, e)$ of a message M:

1. $r^* = g^s y^e$

2. Let $e^* = H(r^*, M)$

If $e^* = e$, then return true (i.e., that the signature is valid). Otherwise return false.

An observation: this signature scheme is randomized. If you request to sign the same message twice, you will likely get two different signatures back in response.

Nevertheless, all possible signatures will correctly be accepted by Verify.

Why does this construction work?

Correctness: if all parties are honest, it is a straightforward math calculation to check that the Sign algorithm produces a signature that passes the Verify algorithm. Try writing it out to convince yourself of this fact!

Unforgeability: the Schnorr signature algorithm has cleverly interwoven two hard problems here – the difficulty of solving discrete logarithms and the random oracle property of the cryptographic hash function H.

I will only give some intuition about the argument here. Put yourself in the shoes of Mallory. How would you break this construction?

- If you pick s and e first, then that determines the value of $r*$ in step 1. But the cryptographic hash function "acts randomly" and it is extremely unlikely that it happens to return e again.
- If you pick r^* first, that determines the value of e in step 2. Returning to step 1, you need to solve the equation $g^s = \frac{r^*}{y^e}$ for the only remaining unknown value: s. But that requires solving a discrete logarithm problem!

3.9.2 Conclusion

At this point, we have two cryptographic tools at our disposal:

- **Hash functions** SHA-2 and SHA-3 that satisfy collision resistance and act like a random oracle, as best as we can tell after decades of cryptanalysis and NIST certification.
- **Digital signatures** like RSA and ECDSA that satisfy existential unforgeability under a chosen message attack if it is difficult to factor large numbers or find discrete logarithms… which we also believe to be hard math problems due to decades (or more) of cryptanalysis.

These two cryptographic tools are all we need to build a cryptocurrency – a system for digital money.

We will begin our exploration of cryptocurrencies next time.