# P79: Cryptography and Protocol Engineering

University of Cambridge
MPhil in Advanced Computer Science / Computer Science Tripos, Part III
Lent term 2025/26

https://www.cl.cam.ac.uk/teaching/2526/P79/

Dr. Martin Kleppmann and Dr. Daniel Hugenroth
{mk428,dh623}@cst.cam.ac.uk

## Contents

**1 Introduction** **2**
1.1 About this module . . . 2
1.2 Basic cryptography recap . . . 7

**2 Elliptic Curve Diffie-Hellman** **13**
2.1 Groups and Fields . . . 13
2.2 Elliptic Curve Groups . . . 16
2.3 Scalar Multiplication and X25519 . . . 19

**3 Software Engineering** **23**
3.1 Cryptography standards . . . 24
3.2 Error handling . . . 31
3.3 Leaky implementations . . . 40
3.4 Types . . . 43

**4 Elliptic Curve Signatures** **53**

**5 Authenticated key exchange** **57**
5.1 Requirements for authenticated key exchange . . . 58
5.2 Implementing a secure AKE protocol . . . 62
5.3 Password-authenticated key exchange . . . 67

**6 Software Engineering II** **70**
6.1 Randomness . . . 70
6.2 Testing . . . 77
6.3 Serialization and marshaling . . . 85
6.4 Randomness II . . . 94
6.5 API Design . . . 102

# 1 Introduction

Welcome to the P79 module on Cryptography and Protocol Engineering. The goal of this module is to help you understand the nuts and bolts of the cryptographic algorithms and protocols that underpin much of the modern world, such as `https://` (more precisely, the TLS protocol) that protects your passwords and credit card numbers from snooping as you send them over the Internet. Without cryptography, any radio transmission (WiFi, cellular data, ...) could easily be intercepted by someone who wants to steal your data!

However, cryptography also has a reputation of being some kind of magic, or at least impenetrably complex. Many explanations of it therefore remain quite vague and high-level. In this module you will see that it's not magic: it's just some mathematics that we can implement with code. And to demonstrate that, you will write the code yourself.

### Motivation

- The Internet (and hence, the modern world) would not function without cryptography
- Cryptography has a reputation of being somehow magic
- Not magic! But complex and easy to get wrong
- This module aims to demystify modern cryptography

Slide 1

## 1.1 About this module

### About this module

- Ran first time in 2024/25
- Practical orientation: focus on you writing code
  - Theory (e.g. security proofs) is very important too
  - Implementation + formalisation would be too much for one module
  - Plenty of engineering challenges in implementation
- Assessed by lab reports + code submissions
  - Don't just write the code, also reflect on it critically
  - Code need not be production-quality, but you should explain what would be required to make it so

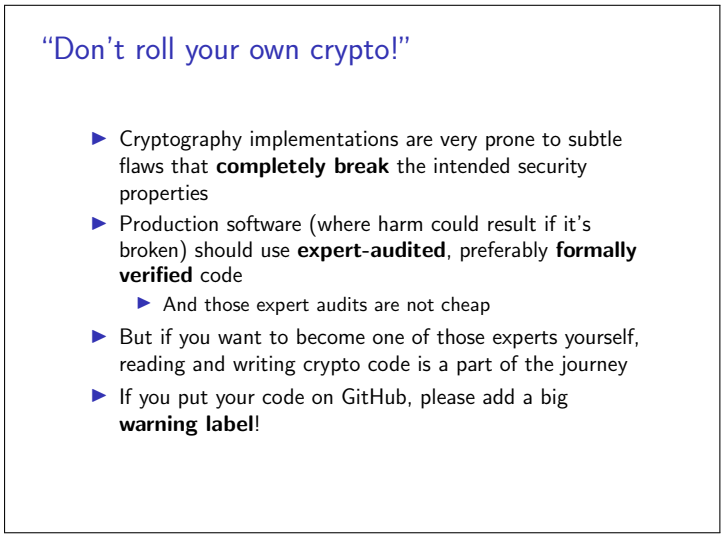
Slide 2

This module is a practical introduction to implementing cryptographic primitives and protocols, and it deliberately avoids a more formal, mathematical treatment of the topic. That's not because formalisation of cryptography is unimportant – quite the contrary: cryptography is so subtle and error-prone that you should be very suspicious of any cryptographic algorithm that does not come with a proof of security (and even a proof doesn't guarantee that a system is actually secure in practice, e.g. because the proof may make assumptions that are not guaranteed to be true in practice). However, formalising cryptography and proving it correct is a very different skill from implementing it, and within the scope of one master's module it would be impossible to do justice to both. We've therefore chosen to focus on implementation

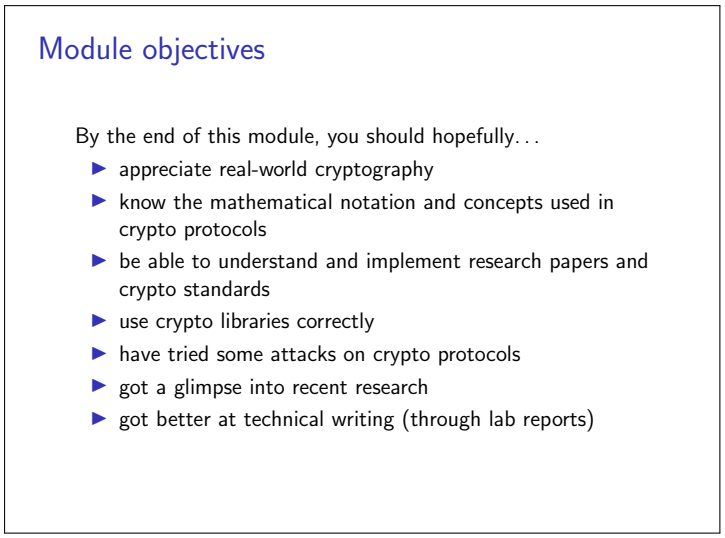
and engineering in this module. Maybe we'll write a separate module on formalizing cryptography in the future.

A slogan you will often hear in industry is "don't roll your own crypto!" – and, as a rule of thumb, this is good advice. There have been far too many examples of cryptographic software written by well-meaning developers that turned out to be utterly insecure, and you should assume that the same is true for any code you write as part of this module. In fact, we will deliberately take some implementation short-cuts in this module in order to make the code easier to understand and to write, even though we know that those short-cuts probably break the security of the implementation.

However, implementing cryptographic algorithms in order to learn about them and studying other people's implementations is very much worthwhile, and "don't roll your own crypto" shouldn't discourage you from doing that. Better advice would be "feel free to roll your own crypto, just make sure that you (or someone else) never use your experimental code for anything where security matters, unless you get it professionally audited". But admittedly that's not as snappy.




Slide 3




Slide 4

This module is assessed based on the code and lab reports that you submit for each of the four topics we cover. The deadlines for these submissions are 3 Feb 2026, 17 Feb 2026, 3 Mar 2026, and 17 Mar 2026 respectively, all at 4pm. Submissions are through Moodle as usual. We aim to give you feedback on each submission before the next submission is due, so that you can take the feedback on board. All four marks will count towards your final grade (weighted equally).

## Assessment

- Four assignments, deadlines two weeks apart:
  1. Elliptic curve Diffie-Hellman (X25519): 3 Feb 2026
  2. Elliptic curve signatures (Ed25519): 17 Feb 2026
  3. Authenticated key exchange (SIGMA): 3 Mar 2026
  4. Private information retrieval: 17 Mar 2026
- For each assignment you need to submit a **lab report** and **code**
- Equal weighting: each assignment counts 25% towards final mark
- Discussions with others are allowed, but code and lab report must be your individual work
- We're happy to answer your questions – please ask!
- We aim to get you feedback on one report before the next is due, so you can take it on board

Slide 5

We will get you started with Python code, and we recommend that you stick with Python as the language for your code submissions. If you feel strongly about your choice of language you may use another one, as long as we can test your code via Docker, but keep in mind that if you have problems we might not be able to help you if you're working in another language.

Although you will be implementing network protocols, you don't need to write any networking code. In fact, your code will be easier to test if you don't use a real network, but simply pass data from the sending function to the receiving function in your tests. However, you should ensure that all data that is sent over the simulated network is encoded as byte strings (not some other type of object), since the encoding and decoding of messages is an important part of the protocol logic.

## Code: what you will implement

We will focus on implementing asymmetric cryptography:

- For hash functions (e.g. SHA-3) and symmetric ciphers (e.g. AES), just use a library
- For big integer arithmetic, use Python's built-in integers
  - NOTE: this is not constant-time
- Everything else (e.g. elliptic curves) you will implement from scratch
- Write suitable tests to catch bugs
- Tolerate maliciously generated input from the network
- No need for real networking, but do encode/decode to bytes
- Don't bother building user interfaces

Slide 6

Production-quality cryptographic code needs to be written to run safely in a very hostile environment in which adversaries are actively trying to break it. For example, any message that is received over the network should be assumed to have been maliciously manipulated, and your code needs to be able to handle such inputs safely.

However, to keep the scope of this module manageable, we will also take some short-cuts and ignore some issues that production-quality code would have to deal with. In particular, we will ignore side-channels, in which an adversary obtains secret information not through the values that are explicitly returned from your functions, but from side-effects of its execution, such as its timing or its power consumption. We will also ignore fault injection attacks, in which the hardware is manipulated to intentionally compute some steps incorrectly.

## Code: how to submit

- We provide you with a template for Python
- You can use another language (e.g. Rust), but we won't be able to help you with it
- If using Python, use type annotations and a static type-checker (we suggest `ty`)
- Submit as a `.zip` archive that includes at least:
  - Your code
  - A `Dockerfile` that typechecks/compiles and runs your code and tests
  - A `run.sh` that builds and starts your Dockerfile
  - Your lab report
- At the end of today's lecture, we will set up a sample project together.

Slide 7

Your submission will be a zipped archive that contains everything to run the project. Using Docker containers ensures that the code not only runs on your machine but also on ours. Importantly, it enforces that all environmental state such as required dependencies is explicitly managed.

## Code: what we look for

- **Correctness** $\gg$ **robustness** $\gg$ performance
- **Simplicity and clarity** are good, complexity is bad
- **Tests** covering both common and edge cases
- **Design of your API:** type checking, error handling, misuse resistance, naming, consistency
- **Documentation:** high-level picture, do not comment every line, make meaningful comments, API language should be self-documenting
- **Well-motivated extensions and comparisons:** benchmarking, interesting testing approaches, compatibility with other libraries, extra hardening, side-channel resistance, ...
  - More features $\neq$ better! More features = more bugs

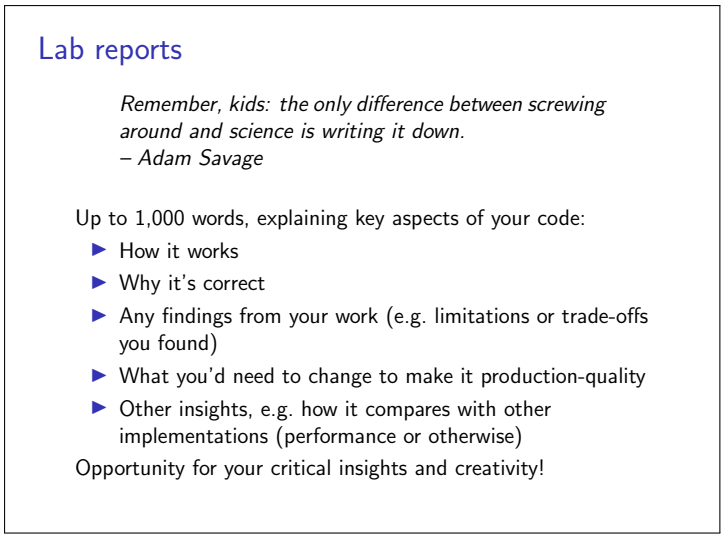
Slide 8

We expect that you write high quality code focusing on correctness and robustness foremost. Differently to some other courses, we do not provide you with a set API—designing a good one is one of your important tasks. Performance is not a central concern, but it will be interesting to compare your implementation with others and to show you understand what parts of the code are slow and would benefit from optimizing.

Please keep the code as simple as possible: every unnecessary line of code is an opportunity for bugs to creep in. With such security-critical code it is important to make it as easy as possible for someone reading the code to confirm that it implements the algorithm correctly. We thus award marks for writing clear, correct code, not for writing lots of code.

Submissions with high grades will have *well-motivated* extensions and comparisons. Extensions might develop the basic task in a meaningful direction (e.g. making it more robust or flexible). Comparisons might compare two implementation variants you have considered or critically compare your API against those of existing libraries. We encourage all original ideas that are related to the topics of this course.

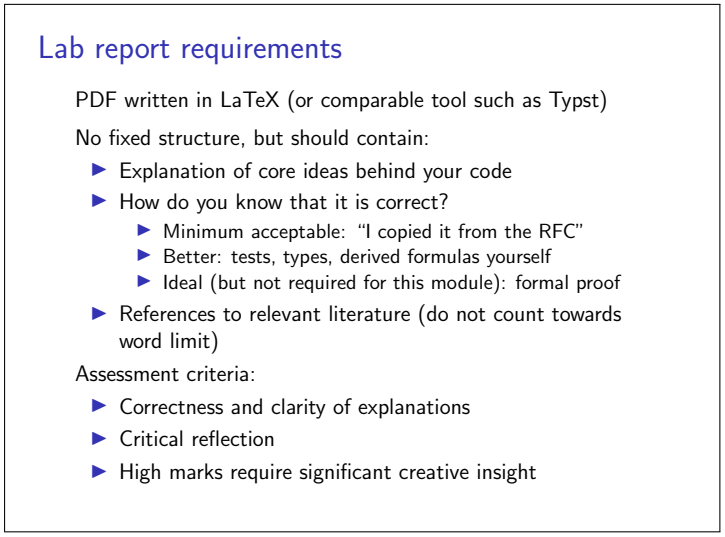
Please don't rush into extensions – we'd much rather have an implementation with no extensions that is correct and well-designed than one that does a bunch of extra things badly. In particular, adding more options and variants of a protocol can easily make things worse, not better. Complexity is the enemy of correctness, and we hope that in this module you'll develop a good taste for making things as simple as possible in order to maximise your chances of getting things right.




Slide 9

Although you will submit your code, an equally important part of each assignment is the lab report that you submit. You need to write the code in order to be able to write the report, but writing the report is how you really think through the topic. It's how you communicate what you've done, document any weaknesses of your code, and explain how they might be fixed.

Since master's modules are expected to include a research element, grades in the upper bands are awarded not just for correctness, but for critical thought and significant creative insights. For that reason we don't have a fixed structure for lab reports: we want to leave space for you to bring in your own ideas. We also highly value the clarity of your explanations, so it's worth investing some time to make sure the report is well written.




Slide 10

Your reports do not need to repeat material that is already in the lecture notes, such as the definition of an algorithm. We'd prefer that you focus on the specific design decisions and insights from your own implementation. You could imagine your reader as being someone who has attended the lectures, but who has not done the assignment themselves, and you are trying to convince them why your solution is a good one.

It's fine for lab reports to be in a style that is more like a technical blog post than a research paper. They may contain your personal thought process, opinions, and unvalidated hypotheses, as long as you clearly label them as such. But it's still good to back up your claims with evidence (e.g. literature references, data) when possible. The language can be somewhat informal as long as it is precise and clear. For example, see Filippo Valsorda's blog (https://words.filippo.io/) for some great examples of technical writing about cryptographic engineering.

Code snippets can be included in your lab report, and they don't count towards the word limit, but please don't overdo it by quoting tons of code. We will read your lab report and code side-by-side, so it's fine for them to cross-reference each other.

### Lab report style tips

- No need to repeat anything from the lecture notes
  - Write for a reader who has read the lecture notes
  - Different from a dissertation or research paper!
- No need for lengthy introduction, motivation, background
- Use the first person ("I designed...")
- Give evidence for your claims when possible
- Critical reflection, subjective opinions, experience reports, trade-off discussions are welcome
- Thought process is welcome, e.g. alternatives considered
- It's fine if you're not sure about something; best to call out doubts explicitly
- Don't give pseudocode; if you want to include code snippets, take them from your actual implementation
  - No need for lengthy code quotations
  - Code and lab report can reference each other

Slide 11

### Recommended reading

We don't know of a textbook that covers the material in this module.

No required reading, but if you want a bit more background (earlier editions are fine too; check the library):

- **More practical:** Jean-Philippe Aumasson. Serious Cryptography, 2nd Edition. No Starch Press, 2024.
- **More formal:** Jonathan Katz and Yehuda Lindell. Introduction to Modern Cryptography, 3rd edition. CRC Press, 2020.
- **On elliptic curves:** Darrel Hankerson, Alfred Menezes, and Scott Vanstone. Guide to Elliptic Curve Cryptography. Springer, 2004.
- References in the lecture notes

Slide 12

## 1.2 Basic cryptography recap

This module assumes that you have already taken an introductory course in cryptography, and that you are already familiar with the basic primitives such as hash functions, symmetric ciphers, public-key encryption, Diffie-Hellman, and signatures. This section will give a brief recap of things you hopefully already know. If not, please catch up quickly, e.g. using the books listed on Slide 12!

We will also see our first few code examples. Please install Python and the PyNaCl library, as shown on Slide 13, so that you can play around with them.

## Basic cryptography recap

Hope you already know (roughly) what SHA-256, AES-GCM, and Diffie-Hellman are...

- Let's quickly recap the core primitives and their security properties.
- Let's also get you writing code that uses those primitives (provided by crypto libraries).
- We will use PyNaCl (Python wrapper for libsodium).
- PyCryptodome is also popular

Setup for the code examples:

```
brew install uv # or equivalent on your OS
uv run --with pynacl python
```

Slide 13

Hash functions are perhaps the easiest-to-use cryptographic construct, since they don't involve any secrets. Several common cryptographic hash functions are available in the Python standard library, and don't even require installing any additional libraries.

## Hash functions

$H(x)$ takes an arbitrary-length bit string $x$ and returns a fixed-length bit string

- e.g. SHA-256, SHA-3, BLAKE2/3
- **Preimage resistance**: given $H(x)$ you can only find $x$ by trying all possible values of $x$
- **Collision resistance**: computationally infeasible to find $x \neq y$ such that $H(x) = H(y)$
- **Birthday paradox**: need $O(\sqrt{2^n}) = O(2^{n/2})$ computation to find a collision in an $n$-bit hash function

```
from hashlib import sha256 # Python standard library
in_bytes = 'Hello'.encode('utf-8')
print(sha256(in_bytes).hexdigest())
# 185f8db32271fe25f561a6fc938b2e...
```

Slide 14

Symmetric encryption and its security definition are summarised on Slides 15 and 16.

## Symmetric encryption

- $key \leftarrow \mathrm{Gen}()$ generates a key
- $c \leftarrow \mathrm{Enc}(key, msg)$ returns ciphertext $c$
- $msg \leftarrow \mathrm{Dec}(key, c)$ decrypts $c$, returns $msg$ or error
- Generally want **authenticated encryption**: ensures that if $c$ is manipulated, Dec returns error
- Block/stream cipher + Msg Authentication Code (MAC)
- **AEAD**: Authenticated Encryption with Associated Data (AD is unencrypted but authenticated)
- e.g. AES-GCM, ChaCha20-Poly1305, XSalsa20-Poly1305

```
from nacl.secret import SecretBox
from nacl.utils import random
key = random(SecretBox.KEY_SIZE)
ciphertext = SecretBox(key).encrypt(b'Hello')
print(SecretBox(key).decrypt(ciphertext))
```

Slide 15

## Security definition for encryption

We normally require authenticated encryption to provide **indistinguishability under adaptive chosen ciphertext attack** (IND-CCA2)

A game between **challenger** and **adversary**:

- Challenger generates secret key
- Adversary may ask challenger to encrypt/decrypt any number of messages ("oracle")
- Adversary chooses two plaintexts $m_0$, $m_1$ of equal length
- Challenger encrypts one of them, chosen randomly, and returns ciphertext $c$ to adversary
- Adversary may continue to request any number of encryptions/decryptions (but not decryption of $c$)
- Adversary guesses which one of $m_0$, $m_1$ was encrypted
- Adversary can't do better than random guess (50/50)

Slide 16

Authenticated encryption and AEAD includes a message authentication code (MAC) as part of its construction, but sometimes a MAC is also useful as a standalone primitive. A MAC is similar to a signature, but it's symmetric: the key used to generate the MAC is the same used to check it.

## Message Authentication Code (MAC)

$\mathrm{MAC}(key, msg)$ takes a symmetric key $key$ and byte string $msg$, returns a fixed-length **authentication tag**

- Security definition: existential unforgeability against chosen-message attack (EUF-CMA). Cannot forge a tag on some message without knowing key, even knowing tags for other messages
- Proof that a message was constructed by someone who knows $key$, and that the message was not altered
- To check, recompute $\mathrm{MAC}(key, msg)$ and check whether you get the same result
- Use a constant-time comparison, otherwise timing allows adversary to guess the right tag!
- Implementations based on hash (HMAC), block cipher (CBC-MAC), or polynomial (Carter-Wegman, GCM)

Slide 17

To check a MAC, you simply recompute it from the message and the key, and compare it to the value provided. Even though this course generally ignores timing side-channels, it is worth pointing out that when checking a MAC it is especially important to use a constant-time comparison. If you compare one byte at a time and return when you find the first byte that differs, the timing leaks information about how many bytes the correct MAC and the provided value have in common, and with repeated attempts it is easy to guess the correct MAC without knowing the key. Writing this comparison function yourself is error-prone [Kario, 2018], so it's safest to use the constant-time comparison function provided by a cryptographic library, such as `hmac.compare_digest()` in Python.

You might be tempted by construct a MAC by simply hashing the key concatenated with the message to be authenticated. However, with SHA-256 and other Merkle–Damgård hash functions, that is insecure due to the internal construction of the hash function (it is vulnerable to length extension attacks). HMAC is a popular construction that works around this weakness by applying the hash function twice, along with some padding (*innerPad* and *outerPad* are just fixed constants).

## Hash-based MAC (HMAC) – RFC 2104

- $H(key \parallel msg)$ is not a secure MAC when $H$ is SHA-256: length extension attacks!
- $\mathrm{HMAC}(key, msg) = H((key \oplus outerPad) \parallel H((key \oplus innerPad) \parallel msg))$ where $\parallel$ is concatenation, $\oplus$ is bit-wise XOR
- Some hash functions (e.g. BLAKE2/3) have a **keyed mode**, which can be used as a MAC directly

```
import hmac
import secrets
key, msg = secrets.token_bytes(16), b'hello'
tag1 = hmac.new(key, msg, 'sha256').digest()
tag2 = hmac.new(key, msg, 'sha256').digest()
print(hmac.compare_digest(tag1, tag2))
```

Slide 18

## Asymmetric (public key) cryptography

Hash functions, symmetric ciphers

- Lots of bit shifts, XORs, lookup tables
- Not much underlying mathematical structure

Asymmetric cryptography:

- Number theory, algebraic objects (groups, finite fields...)
- Based on computational hardness assumptions
  - Multiplying numbers vs. factoring
  - Computing exponentials vs. discrete logarithms
  - Vector/matrix arithmetic vs. solving linear equations with random noise
- Many protocols rely on using asymmetric crypto in creative ways
- Focus of this module

Slide 19

Hash functions and symmetric encryption are well-understood, standardised building blocks, and plenty of resources about them are available if you want to know more. Their internals usually consist of lots of bit manipulation operations arranged in somewhat arbitrary patterns. In this module we won't bother looking inside them, and just take them as given.

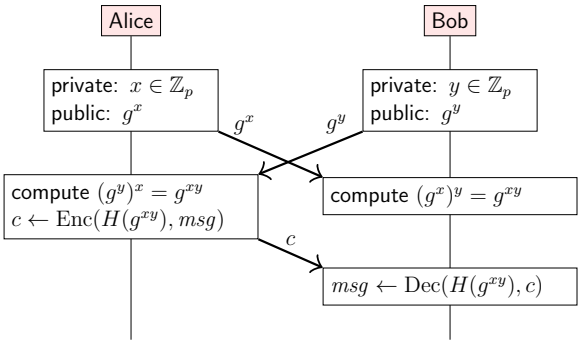
Asymmetric (public-key) cryptography is very different: it is often based on number theory and algebraic objects such as groups and finite fields, and its implementations are often based on analytically derived formulae. It relies on mathematical relationships that are easy to compute in one direction but conjectured to be infeasibly hard in the other direction: for example, multiplying numbers is easy but factoring them is hard; computing exponentials is easy but discrete logarithms are hard; multiplying matrices and vectors is easy but solving a system of linear equations with random noise is hard.

Many cryptographic protocols are based on using asymmetric cryptography in creative ways. Implementing and using asymmetric cryptography will therefore be the primary focus of this module.

Let's start with Diffie-Hellman (DH), the oldest public-key algorithm. The traditional formulation of DH over a finite field is falling out of use nowadays, as it is a bit slow, but its elliptic curve variant (which we will see in the next lecture) is fast and very widely used. DH is a protocol that allows two parties (called Alice and Bob in Slide 20) to agree on a shared key, which can then be used in a symmetric encryption scheme to encrypt messages.

## Diffie-Hellman

Let $g$ be a generator of a group of order $p$ in which discrete logarithms are hard (we'll explain this later).




Slide 20

## Diffie-Hellman

- $sk$ = private (secret) key, $pk$ = public key
- $\mathrm{DH}(sk_A, pk_B) = \mathrm{DH}(sk_B, pk_A)$
- Constructed as $\mathrm{DH}(sk, pk) = pk^{sk}$, $pk = g^{sk} \pmod p$
- Use hash of $\mathrm{DH}()$ output as key for symmetric encryption
- Discrete log: given $g$ and $g^{sk}$, hard to compute $sk$
- **Not authenticated**: network adversary could swap your public key for their own

```
from nacl.public import PrivateKey, Box
alice_sk = PrivateKey.generate()
bob_sk   = PrivateKey.generate()
alice_pk = alice_sk.public_key
bob_pk   = bob_sk.public_key
print(Box(bob_sk, alice_pk).shared_key().hex())
print(Box(alice_sk, bob_pk).shared_key().hex())
```

Slide 21

Plain Diffie-Hellman is generally insecure, since an adversary who can modify the messages exchanged by Alice and Bob can impersonate one user to another. We will see later in this module how to authenticate Diffie-Hellman so that it is secure against such attacks. Diffie-Hellman can also be used to construct a public key encryption scheme.

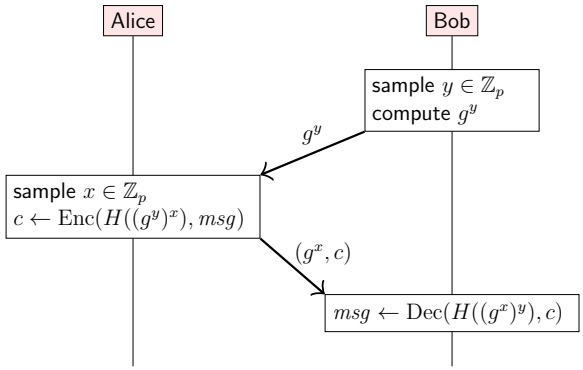
## Asymmetric (public key) encryption

- $(pk, sk) \leftarrow \mathrm{Gen}()$ generates keypair ($pk$ public, $sk$ secret)
- $c \leftarrow \mathrm{Enc}(pk, msg)$ returns ciphertext $c$
- $msg \leftarrow \mathrm{Dec}(sk, c)$ decrypts $c$, returns $msg$ or error
- Unauthenticated: anyone who knows $pk$ can encrypt
- e.g. RSAES-OAEP, Hybrid Public Key Encryption
- Often use Diffie-Hellman to compute shared key, then use authenticated encryption for the actual message
- IND-CCA2 like symmetric case, but adversary is given $pk$

```
from nacl.public import PrivateKey, SealedBox
private = PrivateKey.generate()
public = private.public_key
ciphertext = SealedBox(public).encrypt(b'Hello')
print(SealedBox(private).decrypt(ciphertext))
```

Slide 22

## Public key encryption from Diffie-Hellman




Slide 23

The final commonly-used asymmetric primitive is a digital signature, which allows one party to prove to another that a message is authentic. While a message authentication code (MAC) requires the sender and recipient to have the same symmetric key, a signature is constructed using a private key, and then anybody who has the public key can verify whether the signature is correct.

## Digital signatures

- $(pk, sk) \leftarrow \mathrm{Gen}()$ generates keypair ($pk$ public, $sk$ secret)
- $sig \leftarrow \mathrm{Sign}(sk, msg)$ returns signature
- $ok \leftarrow \mathrm{Verify}(pk, msg, sig)$ returns true or false
- e.g. DSA, ECDSA, EdDSA
- $\approx$ a MAC, but asymmetric
- Security definition: existential unforgeability against chosen-message attack (EUF-CMA). Cannot forge a signature on a message that the key owner didn't sign

```
from nacl.signing import SigningKey, VerifyKey
private = SigningKey.generate()
public = VerifyKey(private.verify_key.encode())
message = 'Hello'.encode('utf-8')
signed_msg = private.sign(message)
print(public.verify(signed_msg)) # b'Hello'
```

Slide 24

## Security parameter

Most cryptography is breakable, given sufficient resources!

Want brute-force to be sufficiently hard that breaking it on a human timescale would be cost-prohibitive.

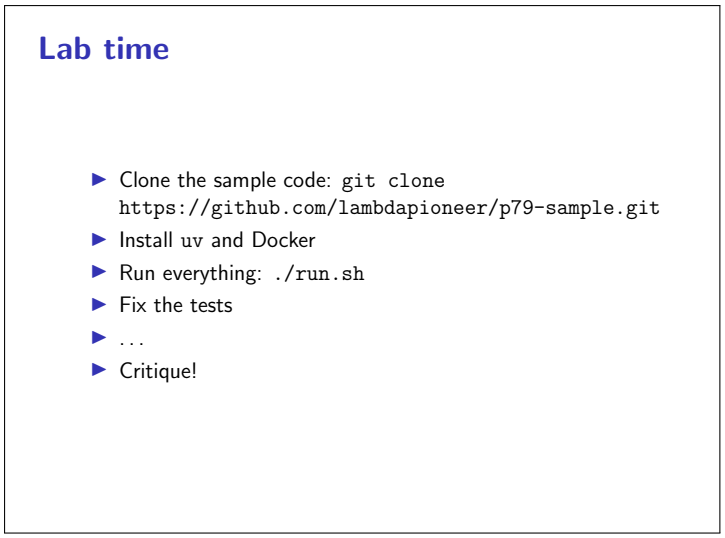
Generally we aim for **128-bit security**:

- On the order of $2^{128}$ computational steps required
- Finding the key for a 128-bit symmetric cipher
- Finding a collision in a 256-bit hash function
- Factoring a 3,072-bit RSA modulus
- Computing discrete log on an 256-bit elliptic curve

Sufficiently large quantum computers could efficiently factorise (break RSA) and compute discrete logs (break elliptic curves).

Can make symmetric ciphers quantum-safe by doubling key length (256 bits); quantum-safe hash is 384 bits

Slide 25

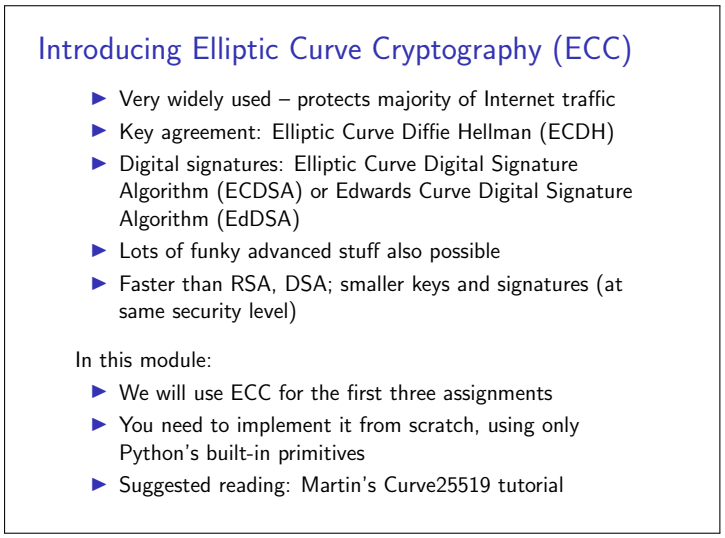



Slide 26

# 2 Elliptic Curve Diffie-Hellman

In this lecture we will get you up to speed with the essentials of elliptic curve cryptography (ECC), which we will use for the first three assignments of this module (the last assignment will use a different cryptosystem). ECC is the most widely used public-key cryptosystem today; a large fraction of TLS connections on the web use it. Its biggest advantage is that it can be secure with quite small keys, often 256 bits, whereas older algorithms such as RSA and DSA need keys to be thousands of bits long in order to be secure (as shown on Slide 25). Smaller keys also mean faster computation.

As a companion to these notes, you can also read Martin's elliptic curve tutorial [Kleppmann, 2020], which shows how to derive a C implementation of one particular elliptic curve algorithm (X25519) from the mathematical curve description.




Slide 27

## 2.1 Groups and Fields

In order to understand and implement ECC you will need a few mathematical tools. We will keep the mathematics to the minimum that you require in order to be able to implement a few key algorithms.

The first thing we need is the concept of a *group*, which is an abstraction of the addition or multiplication operators that you know. The idea is that rather than just adding numbers, we could add arbitrary objects, as long as they satisfy certain properties. If they have the properties shown on Slide 28, we can call them a group. Strictly speaking, by including commutativity we're defining an *abelian group* – however, all the groups you will encounter in this module are commutative, so we will just say "group" even if it should strictly be "abelian group".

## (Abelian) Groups

A set $E$ and an operation $\bullet$ such that:

| | **additive** | **multiplicative** |
|---|---|---|
| **closed**: $\forall a, b \in E.\ a \bullet b \in E$ | $a + b \in E$ | $ab \in E$ |
| **commutative**: $\forall a, b \in E.\ a \bullet b = b \bullet a$ | $a + b = b + a$ | $ab = ba$ |
| **associative**: $\forall a, b, c \in E.$ $(a \bullet b) \bullet c = a \bullet (b \bullet c)$ | $(a + b) + c = a + (b + c)$ | $(ab)c = a(bc)$ |
| **identity exists**: $\exists id \in E.\ \forall a \in E.\ a \bullet id = a$ | $a + 0 = a$ | $a \cdot 1 = a$ |
| **inverse exists**: $\forall a \in E.\ \exists b \in E.\ a \bullet b = id$ | $a + (-a) = 0$ | $a \cdot a^{-1} = 1$ |

Slide 28

Groups are often written using two different notations: additive notation, in which the group operation is $+$, and multiplicative notation, in which the group operation is $\cdot$ or simply writing the multiplied values next to each other. In the additive notation the identity (or neutral element) is 0 and the inverse of $a$ is written $-a$; in the multiplicative notation, the identity element is 1 and the inverse of $a$ is written $a^{-1}$. But these are just two different ways of writing the same thing. You can also use different symbols, such as $\bullet$ for the group operation.

Two examples of groups are shown on Slide 29, but many others also exist. The simplest to understand is the additive group of integers modulo $n$, where the set is the set of integers from 0 to $n - 1$, and the group operation is addition modulo $n$. This group has an identity element of 0, and the inverse element of $a$ is $n - a$.

A trickier example is the *multiplicative* group of integers modulo $n$, which exists for any $n$, but is easiest to describe when $n$ is prime. In that case, the identity element is 1, the operator is multiplication modulo $n$, and the set is the set of integers from 1 to $n - 1$ (0 is not a member of the set since it doesn't have a multiplicative inverse, i.e. $\nexists a.\ a \cdot 0 = 1$). It can be proved that every element of this set has an inverse, but it's not immediately obvious. More generally, even if $n$ is not prime, $a$ has a multiplicative inverse modulo $n$ if $\gcd(a, n) = 1$. There are several ways of computing a multiplicative inverse modulo $n$. The easiest way is using Fermat's little theorem, as shown on Slide 29.

## Groups of integers modulo $n$

$\mathbb{Z}_n$: Additive group of integers modulo $n$

- $\mathbb{Z}_n = \{0, 1, \ldots, n - 1\}$
- Operator is addition mod $n$. Python: `(a + b) % n`
- Inverse is $-a = n - a$

$\mathbb{Z}_n^*$: Multiplicative group of integers modulo $n$

- When $n$ is prime, $\mathbb{Z}_n^* = \{1, 2, \ldots, n - 1\}$
- Operator is multiplication mod $n$. Python: `(a * b) % n`
- Inverse of $a$ exists when $\gcd(a, n) = 1$
- For prime $n$, compute inverse by Fermat's little theorem:
  $a^{n-1} = a \cdot a^{n-2} \equiv 1 \pmod{n}$
  so $a^{n-2} \pmod{n}$ is the multiplicative inverse of $a$

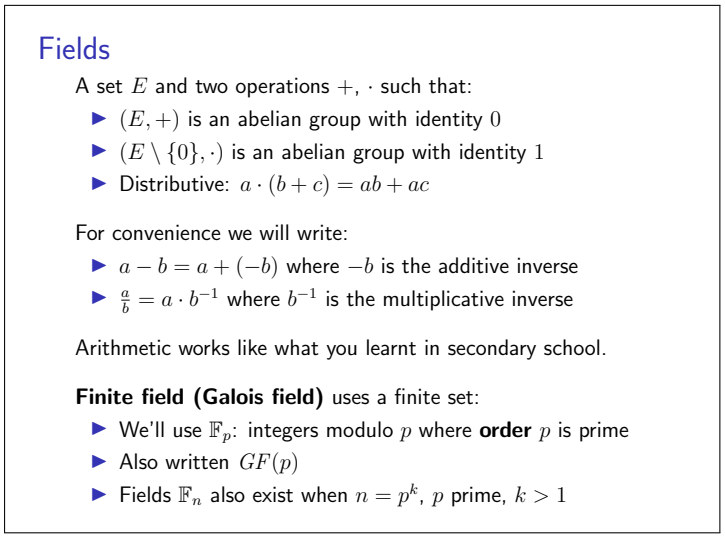
```
p = 2**255 - 19; a = 42 # p is prime
a_inv = pow(a, p - 2, p)
print((a * a_inv) % p) # 1
```

Slide 29

The next mathematical abstraction we will need is the *field*. Like a group, it is based on a set, but it has two operators, addition and multiplication. Each of the operators forms an abelian group with that set, except that 0 (the identity element of the addition operation) does not have a multiplicative inverse, and therefore isn't part of the multiplication group. We can then define subtraction and division in terms of additive and multiplicative inverses, and exponentiation in terms of repeated multiplication.

The rational numbers $\mathbb{Q}$, and the real numbers $\mathbb{R}$, each form a field with the usual addition and

multiplication operations. The arithmetic that you learnt in school using rational and real numbers actually works with any field: you can write polynomials, solve equations for some variable, and manipulate expressions using addition, subtraction, multiplication, and division.




Slide 30

In cryptography, real and rational numbers would be awkward to work with, since they are infinite sets, and so the representation of a field element may require an unbounded number of bits. More useful for our purposes are *finite fields*, also known as *Galois fields*. A finite field is based on a finite set, and hence has fixed-size elements, but it also allows you to do algebra in the familiar way. The number of elements in such a field is called the *order*. In this module we will only use fields whose order is a prime number $p$. Such a field is written $\mathbb{F}_p$; the elements of the set are the numbers $\{0, \ldots, p-1\}$; addition and multiplication are performed modulo $p$, like in the groups shown on Slide 29.

In this module, whenever you see a number, it's probably an element of a field $\mathbb{F}_p$ for some prime $p$. In the first two assignments you will implement the X25519 and Ed25519 algorithms, which are both based on the finite field $\mathbb{F}_p$ for the prime $p = 2^{255} - 19$ (hence the 25519 in the name of those algorithms). Elements of that field are 255 bits long, or almost 32 bytes. Most CPUs and programming languages don't natively support arithmetic on numbers that large, so you have to use a bignum (arbitrary-precision arithmetic) library. However, Python will quite happily let you work with such numbers, as its `int` datatype automatically switches to a bignum implementation internally when the values get big.

This means that implementing finite field arithmetic in Python is quite easy – but beware that it is not constant-time, and hence vulnerable to side-channel attacks. In this module you may use Python's integer arithmetic (if using another language you may use a bignum library), but this would not be appropriate in production code where side-channels matter. Production code therefore often contains custom field arithmetic code that is very carefully designed to be constant-time. (Addition, subtraction, and multiplication of big numbers is quite easy to implement; the difficult part is usually the reduction modulo $p$.)

Also beware that although Python offers two different division operators (`a / b` for floating-point division, and `a // b` for integer division that rounds downwards), neither is the correct definition of division for the finite field of integers modulo $p$. To implement finite field division, you need to compute a multiplicative inverse as shown on Slide 29.

### Implementing finite fields

For elliptic curves we will use the field $\mathbb{F}_p$ for a large prime $p$

- In particular, $p = 2^{255} - 19 =$ `0x7fffffffffffffffffffffffffffffff ffffffffffffffffffffffffffffffed` (255 bits long)
- Python integers have no fixed size: 255-bit (or bigger) ints are no problem
  - Addition `(a + b) % p` is fine
  - Subtraction `(a - b) % p` is fine
  - Multiplication `(a * b) % p` is fine
- Most CPUs natively have max. 64-bit arithmetic $\Longrightarrow$ need to break down big ints into several smaller ones
- ⚠ Python int arithmetic and most bignum libraries are **not constant-time** (not suitable for production code)
- ⚠ Python division operators `a / b` and `a // b` **do not work** for finite fields – need to use multiplicative inverse

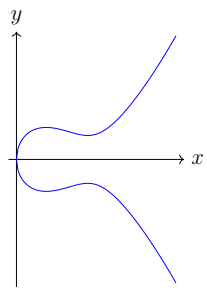
Slide 31

## 2.2 Elliptic Curve Groups

With that background material out of the way, we can now get started with elliptic curves. There are several different families of curves with slightly different curve equations; the one we will focus on for now is Curve25519, which belongs to the family of *Montgomery curves*. Its equation is shown on Slide 32.

Plotted over the field of real numbers $\mathbb{R}$, the curve equation produces a characteristic shape shown on Slide 32. You will notice that it does not have the shape of an ellipse, despite what you might expect from the name; the reason they are called "elliptic curves" goes deeper into mathematics than we need to care about in this module. In fact, we won't use the curve over $\mathbb{R}$, but rather over $\mathbb{F}_p$ where $p = 2^{255} - 19$: that is, the $x$ and $y$ coordinates are both elements of the field $\mathbb{F}_p$. The arithmetic works the same: it just means that when computing the expressions $y^2$ and $x^3 + ax^2 + x$, every addition and every multiplication is done using integers modulo $p$. We don't write "mod $p$" all the time because it would get very tedious. We could plot the curve over $\mathbb{F}_p$ too, but it doesn't look very interesting – it's just a bunch of seemingly-randomly scattered dots. The plot over $\mathbb{R}$ is better for getting an intuition of what is going on.

### Montgomery curves (a family of elliptic curves)




For now we will use **Curve25519**, the elliptic curve

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

over the field $\mathbb{F}_p$ where $p = 2^{255} - 19$ and $a = 486662$ (params chosen to make the curve cryptographically useful).

A point $(x, y)$ is *on the curve* if it satisfies the curve equation.

Plot shows what it would look like over $\mathbb{R}$ with $a = -1.9$.

Slide 32

Notice that the curve shape is symmetric with respect to the $x$ axis; this comes from the fact that the variable $y$ only occurs in the term $y^2$, and therefore if $(x, y)$ is a point on the curve, $(x, -y)$ must also be a point on the curve. (If you're wondering how we can negate $y$ when $\mathbb{F}_p$ contains no negative numbers: remember that $-y \equiv p - y \mod p$.)

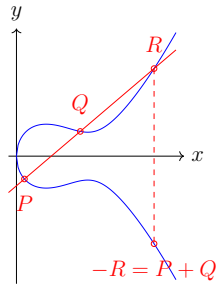
Now we can use this curve to define a group. These definitions will seem strange and arbitrary at first, but just accept them for now. Unlike the groups we saw on Slide 29, the elements $E$ of this group aren't numbers, but *points on the curve* (that is, pairs of $(x, y)$ coordinates where $x, y \in \mathbb{F}_p$), plus one

special element $\infty$, also written $\mathcal{O}$, that we call the "point at infinity" (you can think of this being a point that is located infinitely far up the $y$ axis, and doesn't have a $x$ coordinate; all vertical lines intersect that point). We will write the group operator as $+$, and use $\infty$ as the identity element: that is, we define $P + \infty = P$ for all points $P \in E$.

We define the inverse $-P$ of a point $P \in E$ as the mirror image of $P$ with respect to the $x$ axis, in other words the point with the same $x$ coordinate as $P$ and the $y$ coordinate negated. As explained previously, $-P$ must also be a point on the curve. We define the inverse of the point of infinity to be itself: $-\infty = \infty$.

The group operation $P + Q$ to add two points $P, Q \in E$ is defined as follows. First, Slide 33 shows the case where the $x$ coordinates of $P$ and $Q$ are different (i.e. $P \neq Q$ and $P \neq -Q$).

### Constructing a group from a curve




We will now define a **group** whose elements $E$ are **points on the curve** (plus one special element $\infty$ called "point at infinity").

$E = \{(x, y) \mid y^2 = x^3 + ax^2 + x\} \cup \{\infty\}$

Define identity element as $\infty$

Define inverse as: $-(x, y) = (x, -y)$

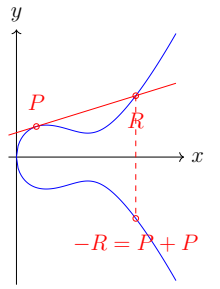
The $+$ operator combines two points $P, Q \in E$ to produce a new point:

- Draw straight line through $P$ and $Q$
- It intersects the curve at $R$
- Mirror $R$ by $x$ axis to get $P + Q$

Slide 33

To add a point $P \in E$ to itself, we draw a tangent to the curve at the point $P$ and then proceed in the same way (intersecting the curve at a third point and mirroring that point by the $x$ axis).

### Adding a point to itself (doubling)




The definition on the last slide works when $P \neq Q$ and $P \neq -Q$.

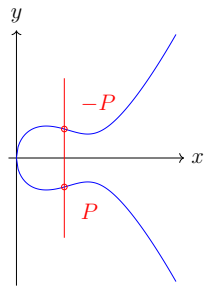
To add $P \in E$ to itself ($P + P = 2P$):

- Draw a tangent to the curve at $P$
- It intersects the curve at $R$
- Mirror $R$ by $x$ axis to get $P + P$

Slide 34

When the line through the two points is vertical, we define their sum to be the point at infinity $\infty$.

## Handling vertical lines




The final case we need to handle is $P + Q$ where $Q = -P$ (i.e. $P$ and $Q$ have the same $x$ coordinate, but different $y$ coordinates).

In this case the line through $P$ and $Q$ is vertical, and there is no (finite) third intersection point.

Define $P + Q = \infty$ if $P = -Q$.

Fits with definition of $-P$ as inverse of $P$, and $\infty$ as identity element.

(By definition, $\forall P \in E.\ P + \infty = P$)

Slide 35

From that geometric intuition we can derive formulas for adding two points and doubling a point; these are called the *group law*. The formulas look intimidatingly complicated, but the derivation is actually quite straightforward – the ECC tutorial [Kleppmann, 2020] shows how to do it. You can also look them up in a database: https://www.hyperelliptic.org/EFD/

## Constructing a group from a curve

Amazingly, that definition results in an abelian group $(E, +)$

(Closed, identity exists, and inverse exists by definition; easy to see that it's commutative. Proving associativity is harder.)

**Group law** for Montgomery curves ($y^2 = x^3 + ax^2 + x$):

**Point addition:** $P + Q = (x_1, y_1) + (x_2, y_2) = (x_3, y_3)$ where

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - a - x_1 - x_2$$

$$y_3 = \frac{(2x_1 + x_2 + a)(y_2 - y_1)}{x_2 - x_1} - \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^3 - y_1$$

**Point doubling:** $2P = (x_1, y_1) + (x_1, y_1) = (x_3, y_3)$ where

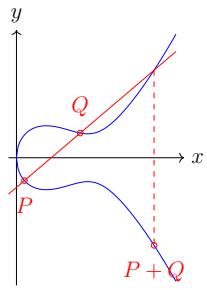
$$x_3 = \left(\frac{3x_1^2 + 2ax_1 + 1}{2y_1}\right)^2 - a - 2x_1$$

$$y_3 = \frac{(3x_1 + a)(3x_1^2 + 2ax_1 + 1)}{2y_1} - \left(\frac{3x_1^2 + 2ax_1 + 1}{2y_1}\right)^3 - y_1$$

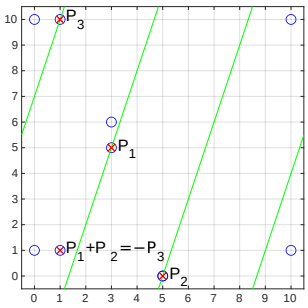
Slide 36

The geometric interpretation of the group law only really makes sense when the curve is defined over the real numbers $\mathbb{R}$: calculating a tangent, as on Slide 34, requires computing a derivative, which is not defined over $\mathbb{F}_p$. Slide 37 shows what the curve and the group law look like over a finite field. However, it turns out that even if you derive the formulas on Slide 36 over $\mathbb{R}$, the final equations work equally well in a finite field. And that's what we do for cryptographic purposes: we use formulas like those above, and just do all the arithmetic with integers modulo $p$. (Remember that the fractions in the equations on Slide 36 are shorthand for multiplicative inverses, so the result of a fraction is still an integer!)

## Elliptic curve over a finite field




Curve over $\mathbb{R}$




Curve over $\mathbb{F}_{11}$

Image by Markus Kuhn

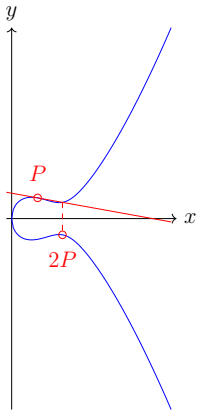
Slide 37

## 2.3 Scalar Multiplication and X25519

Now that we have defined + as the group operator on elliptic curve points, we can define the product between a scalar (i.e., an integer) $k \in \mathbb{N}$ and an elliptic curve group element $P$ (a curve point or $\infty$) as adding $P$ to itself $k$ times. Each of those additions uses the group law from Slide 36. Note the "types" of this multiplication: you can only multiply a scalar with a group element, and the result is another group element. You cannot multiply a point with a point, you can only add them.

You now have to be quite careful with the notation, since + sometimes means adding finite field elements (i.e. integers modulo $p$), and sometimes means adding elliptic curve group elements (i.e. curve points) using the group law. You have to look at the types of the variables involved to see which one it is. Likewise, multiplication is sometimes between field elements, and sometimes between a scalar and group element. Yes, this is very confusing. We considered using different notations to tell the two apart, but then decided to stick with the notation used in most of the literature on elliptic curves, since you will need to be able to read it. And, unfortunately, most of the literature overloads the operators in this way. Sometimes you see a notation like $[k]P$ to mean $P$ added to itself $k$ times.

## Repeated addition of a point to itself




Define **scalar multiplication** of a point $P \in E$ as:

$$kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}$$

If you look at the sequence of points $P$, $2P$, $3P$, ..., it "jumps around" all over the curve.

For a suitably chosen $P$, this sequence repeats only for very large $k$.

Given $P$ and $kP$, it's **hard** to determine $k$ (try all possible values!)

Slide 38

Say you have a group element $kP$ that you obtained by adding the group element $P$ to itself $k$ times. If you then add $kP$ to itself another $j$ times, the result is the same as if you had first multiplied $j$ and $k$, and then added $P$ to itself $jk$ times. This follows from the fact that the + operator on the elliptic curve group is associative. This in turn makes it possible to compute $kP$ efficiently, even when $k$ is a very large number.

## Multiplying a point by a number

Because the group operator $+$ is associative we have:

$$j(kP) = \underbrace{(\underbrace{P + \cdots + P}_{k \text{ times}}) + \cdots + (\underbrace{P + \cdots + P}_{k \text{ times}})}_{j \text{ times}}$$
$$= \underbrace{P + \cdots + P}_{j \cdot k \text{ times}} = (jk)P$$

**Double-and-add** algorithm to compute the scalar product:

$$kP = \begin{cases} P & \text{if } k = 1 \\ 2(\frac{k}{2}P) & \text{if } k \text{ is even} \\ 2(\frac{k-1}{2}P) + P & \text{if } k \text{ is odd and } k > 1 \end{cases}$$

Computes $kP$ with $O(\log k)$ point additions/doublings

Slide 39

The *order* of the elliptic curve group is the number of solutions to the curve equation, plus one for the the point at infinity. (Remember that when the curve is defined over $\mathbb{F}_p$, there are $p^2$ possible $(x, y)$ coordinate combinations, and only some of them are solutions to the curve equation.) The order of the group depends on the size of the underlying field and the parameters of the curve equation. There are efficient algorithms to determine the order of a particular curve, without having to enumerate all the possible points.

Often curve parameters are chosen such that the order of the group is a large prime number. In the case of Montgomery curves this is not possible, so the parameters are chosen to make the group order as cryptographically useful as possible: namely, the product of a small constant (8) and a large prime ($q = 2^{252} + 27742317777372353535851937790883648493$). Note that the order of the EC group ($8q$) is not the same as the order of the underlying field ($p$). Elliptic curves where the group order equals the field order are called *anomalous* and are insecure, so we don't use them. Another desirable criterion for the choice of parameters is that they are *rigid* (https://safecurves.cr.yp.to/rigid.html), which means that all parameter choices are explained, and avoiding the use of unexplained "magic constants" that could potentially hide a weakness.

We say that the set of group elements that can be produced by adding a group element $P$ to itself repeatedly is the set *generated* by $P$. That set is always a subgroup of $E$, and the size of that set is called the *order* of $P$. It can be proved that the order of a group element always divides the order of the group. In the case of Curve25519, this means there are group elements whose order is $q$. One of these (arbitrarily chosen, one with $x$ coordinate $x = 9$) will serve as the *base point* for the following algorithms.

## Generator of a group

$|E|$ (the number of elements in the group) is its **order**.

Curve parameters determine $|E|$; prime if possible.

In Curve25519 ($p = 2^{255} - 19$, $a = 486662$), we have $|E| = 8q$ where $q$ is a large (252-bit) prime.

Given $P \in E$, consider the series $P$, $2P$, $3P$, ...

If it repeats after $m$ steps, we say $|P| = m$ is the **order** of $P$.

$\langle P \rangle = \{iP \mid i \in \mathbb{N}\}$ is the set **generated** by $P$. $|\langle P \rangle| = |P|$

If $\langle P \rangle = E$, then $P$ is a **generator** of $E$ with $|E| = |\langle P \rangle|$.

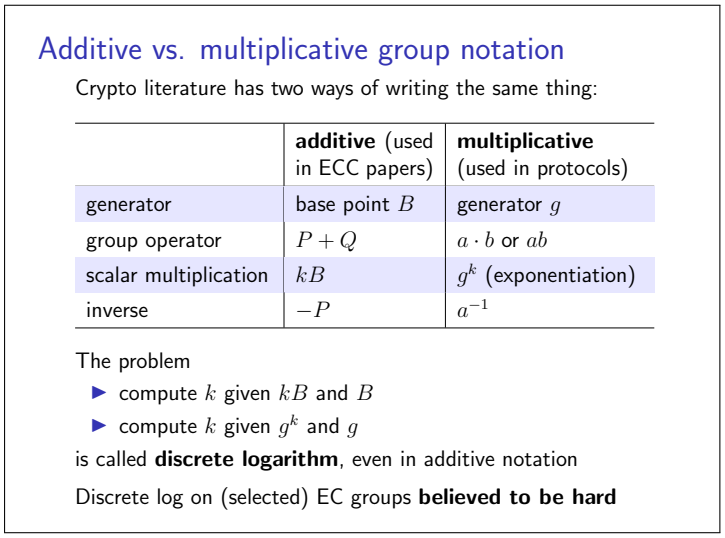
If $\langle P \rangle \subset E$, then $\langle P \rangle$ is a **subgroup** of $E$.

We choose a **base point** $P$ for which $|P|$ is a large prime ($q$).

Slide 40

Now we have to point out another way in which cryptography papers use confusing notation. Recall from Slide 28 that groups are often written with either additive notation (group operation is $+$) or multiplicative notation (group operation is $\cdot$). In the literature on elliptic curves it is traditional to write

the group operation as $+$, and to write $kP$ when applying $P$ to itself $k$ times using $+$. In the literature on cryptographic protocols it is traditional to assume a group but to write it using multiplicative notation, so the group operation is $\cdot$, and to write $g^k$ when applying $g$ to itself $k$ times using $\cdot$. Despite the different notation, these two are actually the same thing!



## Additive vs. multiplicative group notation

Crypto literature has two ways of writing the same thing:

| | **additive** (used in ECC papers) | **multiplicative** (used in protocols) |
|---|---|---|
| generator | base point $B$ | generator $g$ |
| group operator | $P + Q$ | $a \cdot b$ or $ab$ |
| scalar multiplication | $kB$ | $g^k$ (exponentiation) |
| inverse | $-P$ | $a^{-1}$ |

The problem

- compute $k$ given $kB$ and $B$
- compute $k$ given $g^k$ and $g$

is called **discrete logarithm**, even in additive notation

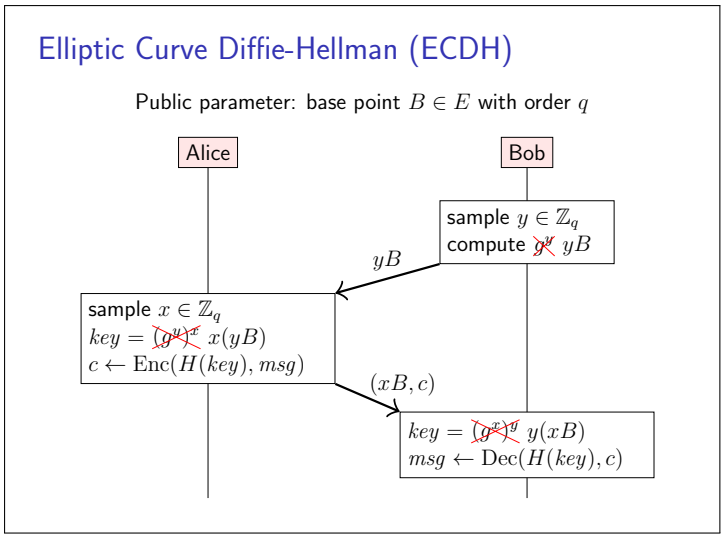
Discrete log on (selected) EC groups **believed to be hard**

Slide 41

In cryptographic protocols it is common to just assume a group in which discrete logarithms are hard, without caring how it is implemented (it could be done with elliptic curves, or with the finite field of integers modulo a prime, or some other mechanism). Here, the concept of a group serves as an abstraction that allows us to reason about a protocol without worrying about the implementation details. However, in order to actually implement the protocol, you have to instantiate the abstract group with something concrete. And elliptic curve groups are often used for this purpose, since they are fairly efficient, and discrete logs on those groups are believed to be hard as long as the curve parameters are suitably chosen.

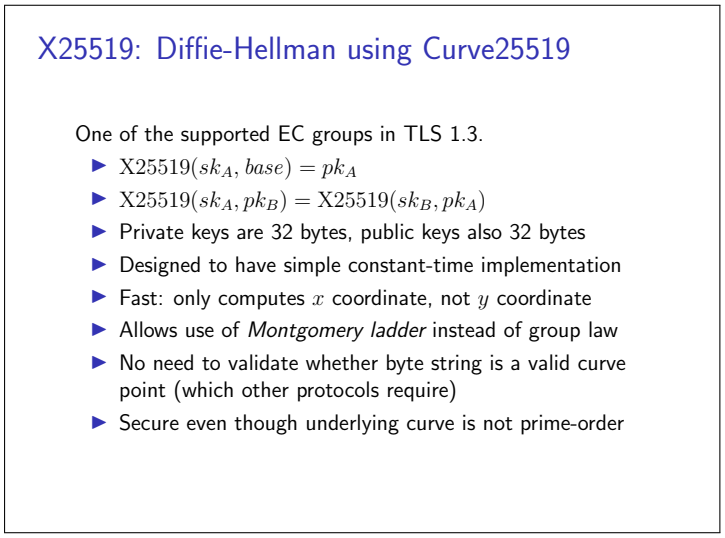
(There are a bunch of details that matter when choosing curve parameters, but we don't need to get into them in this module since we will just use published parameters of well-known curves. Another side-note: many cryptographic protocols assume that the group order is prime, but that is not the case with Curve25519, since it has $|E| = 8q$. Using a non-prime-order group in a context that requires a prime-order group is a source of subtle bugs, such as small subgroup confinement attacks. However, it's possible to construct a prime-order group on top of Curve25519 using Ristretto [de Valence et al., 2020].)

In fact, you already saw the multiplicative group notation in action on Slide 20 when we introduced Diffie-Hellman. Back then we said that $g$ is a generator of a group of order $p$ in which discrete logarithms are hard. Now we know how to construct such a group: we can use the Curve25519 elliptic curve group for example, and as generator we will use a curve point $B \in E$ with order $q$ from the curve $E$. Then we just replace the exponentiation of $g$ with a scalar multiplication of the base point, and voilà, we have Elliptic Curve Diffie-Hellman (ECDH)! By the argument on Slide 39, we have $x(yB) = (xy)B = (yx)B = y(xB)$, so Alice and Bob end up with the same symmetric key.




Slide 42

Let's take a look at X25519, a specification of how exactly to do Diffie-Hellman over the Curve25519 group. Please read up the details in the original paper by Bernstein [2006], as well as RFC 7748 [Langley et al., 2016] and Martin's ECC tutorial [Kleppmann, 2020]. X25519 makes a number of careful design choices to achieve both very good performance and strong security. For example, when a group element is encoded into bytes and sent over the network (like $yB$ is sent from Bob to Alice, and $xB$ is sent from Alice to Bob in Slide 42), the recipient would normally need to check that the byte string actually represents a valid point on the curve, and reject it if not (accepting invalid points can be a source of vulnerabilities). However, in the case of X25519, such validation is not required: the algorithm is designed to be safe given any arbitrary 32-byte string as input. That removes a source of bugs, but also makes the algorithm faster, since point validation has a computational cost.




Slide 43

You can find ugly pseudocode for X25519 in RFC 7748 (we hope your code will be nicer), and the algorithm is outlined on Slide 44. In case you're wondering how to compute a square root in a finite field: we will get to that on Slide 118. For now you can look up the $y$ coordinate of the base point in Section 4.1 of RFC 7748, where it is called `V(P)`.

### X25519 algorithm

- **Private key**: start with 32 random bytes
  - interpret bytes as little-endian integer, then do clamping:
  - set 3 least-significant bits to 0 (make it a multiple of 8)
  - set most significant bit to 0 (make it $< 2^{255}$)
  - set second-most significant bit to 1 (make it $\geq 2^{254}$)
- **Public key**: 32 bytes received from the network
  - interpret bytes as little-endian integer
  - set most significant bit to 0, then reduce mod $p$
  - result is the coordinate $x \in \mathbb{F}_p$ of a curve point
- **Base point**: $x = 9$, $y = \sqrt{x^3 + 486662x^2 + x}$
- **X25519 function**:
  - Input private and public key (or private key and base)
  - Compute scalar product of private key (scalar) and the public key (group element)
  - Output $x$ coordinate of the resulting group element
- Hash the result before using it as symmetric key

Slide 44

And that brings us to the first task of your first assignment.

### Assignment 1

**Implement X25519** from scratch, relying only on bignums for field arithmetic.

Do it two ways and check they agree:

- Using the Montgomery curve group law and a double-and-add algorithm for scalar multiplication
- Using the Montgomery ladder. See Bernstein's paper; RFC 7748 (Python code); Martin's ECC tutorial (C code)

You can find test vectors in RFC 7748.
See lecture notes for hints on interpreting RFC 7748.

Due date for code and lab report: 3 Feb 2026.

Hope you have fun!

Slide 45

For the variant that uses the group law, you can use the formulas on Slide 36 (affine coordinates). You will need the $y$ coordinate of the curve points you deal with; for the sake of this task you can assume that the $y$ coordinate is sent along with the $x$ coordinate when sending a public key over the network (even though that doubles their size). The Montgomery ladder implementation shouldn't ever need $y$ coordinates.

A few hints to help you with RFC 7748:

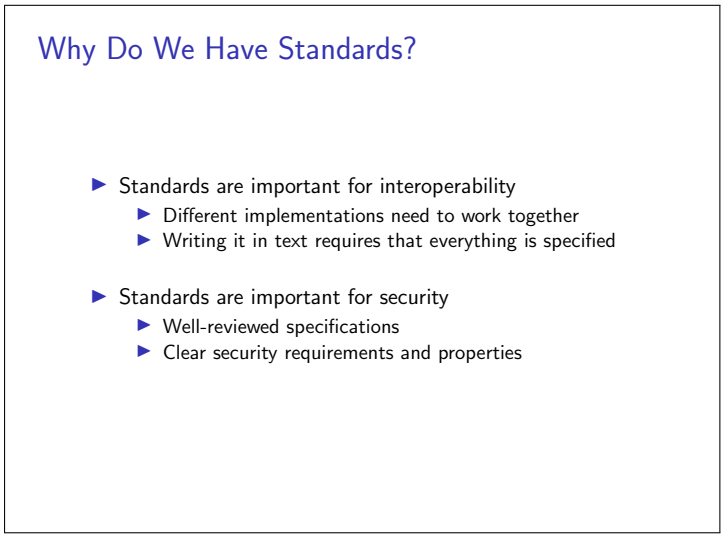
- In the test vectors, the scalars given in hexadecimal are given before clamping, but the scalars given in decimal are after clamping.
- The input $u$ coordinate in hexadecimal is before the most significant bit is set to zero, but the decimal value is after.
- In the second set of vectors, the input $u$ coordinate is not a square in the field.

# 3 Software Engineering

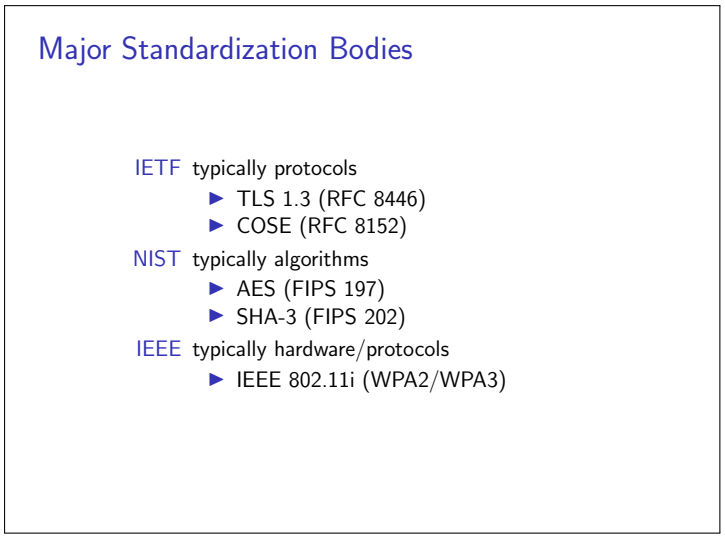
Implementations of cryptographic algorithms and protocols do not just rely on their mathematical design, but also require careful engineering to ensure security and practical performance. The translation into the real world often requires us to make choices that are not captured by the mathematical model. For instance, we need to choose the right parameter sizes, or how to handle errors. In this section we visit important aspects of software engineering that will become relevant not just for your lab reports, but also when you implement cryptographic protocols and other critical software in the future.

## 3.1 Cryptography standards

Standards are the important binding between the mathematical model and the real world. They provide a precise specification of the algorithm, including all parameters and execution steps, in the real world. That is, they are concerned with the actual implementation down to the individual instructions and byte-level details of the data structures.




Slide 46




Slide 47

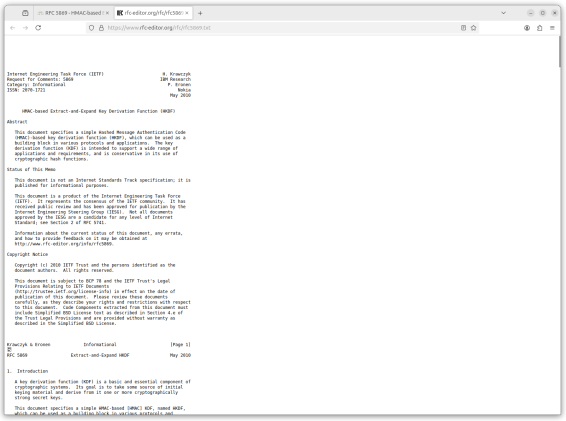
Most standards relevant for this course are published by the Internet Engineering Task Force (IETF) and the National Institute of Standards and Technology (NIST). In practice, you will encounter standards from other bodies as well, e.g. the Institute of Electrical and Electronics Engineers (IEEE) and the International Organization for Standardization (ISO).

## Reading RFCs

- Example: RFC 5869 - HMAC-based Extract-and-Expand Key Derivation Function (HKDF)
- Let's walk through how to read and understand an RFC
- They are available at https://datatracker.ietf.org/doc/html/rfc5869

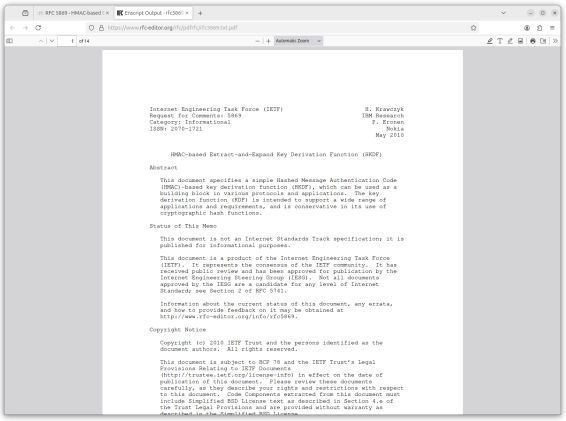
Slide 48

## RFC Formats - Text



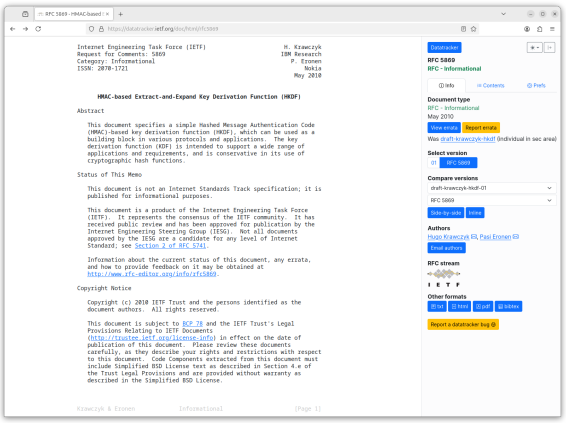
Slide 49

## RFC Formats - PDF



Slide 50

## RFC Formats - HTML



Slide 51

The HTML format proves particularly valuable when writing your lab report, offering convenient BibTeX integration and efficient navigation features.

## RFC Header (RFC 5869)

```
Internet Engineering Task Force (IETF)                    H. Krawczyk
Request for Comments: 5869                                IBM Research
Category: Informational                                     P. Eronen
ISSN: 2070-1721                                                Nokia
                                                            May 2010

        HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

Abstract

   This document specifies a simple Hashed Message Authentication Code
   (HMAC)-based key derivation function (HKDF), which can be used as a
   building block in various protocols and applications.  The key
   derivation function (KDF) is intended to support a wide range of
   applications and requirements, and is conservative in its use of
   cryptographic hash functions.
```

Slide 52

The abstract and introduction provide context and motivation that can be very helpful for understanding the document. In addition, the header contains other key information about the document, such as the category, the status, and the date of publication.

## RFC Categories

- Standards Track
  - Proposed Standard: Initial standardization
  - Internet Standard: Proven, stable standard
- Informational: Background information, guidelines
- Experimental: Experimental protocols

Slide 53

## Introduction (RFC 5869)

**1. Introduction**

A key derivation function (KDF) is a basic and essential component of cryptographic systems. Its goal is to take some source of initial keying material and derive from it one or more cryptographically strong secret keys.

This document specifies a simple HMAC-based [HMAC] KDF, named HKDF, which can be used as a building block in various protocols and applications, and is already used in several IETF protocols, including [IKEv2], [PANA], and [EAP-AKA]. The purpose is to document this KDF in a general way to facilitate adoption in future protocols and applications, and to discourage the proliferation of multiple KDF mechanisms. It is not intended as a call to change existing protocols and does not change or update existing specifications using this KDF.

Slide 54

## Notation (RFC 5869)

**2.1. Notation**

HMAC-Hash denotes the HMAC function [HMAC] instantiated with hash function 'Hash'. HMAC always has two arguments: the first is a key and the second an input (or message). (Note that in the extract step, 'IKM' is used as the HMAC input, not as the HMAC key.)

When the message is composed of several elements we use concatenation (denoted |) in the second argument; for example, HMAC(K, elem1 | elem2 | elem3).

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [KEYWORDS].

Slide 55

## Notation

Key words (defined in RFC 2119):

- **MUST=SHALL** (= is required to)
- **SHOULD** (= strongly recommended)
- **MAY** (= optional)
- **SHOULD NOT** (= not recommended)
- **MUST NOT=SHALL NOT** (= prohibited)

> *"MUST This word, or the terms REQUIRED or SHALL, mean that the definition is an absolute requirement of the specification."*

Slide 56

## Notation

Key words (defined in RFC 2119):

- **MUST=SHALL** (= is required to)
- **SHOULD** (= strongly recommended)
- **MAY** (= optional)
- **SHOULD NOT** (= not recommended)
- **MUST NOT=SHALL NOT** (= prohibited)

> *"SHOULD This word, or the adjective RECOMMENDED, mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course."*

Slide 57

Understanding these keywords is fundamental for making sound implementation decisions. When writing your lab report, you'll need to carefully justify any deviations from SHOULD requirements or explain your choices between MAY options.

## Algorithm (RFC 5869)

```
Inputs:
   PRK      a pseudorandom key of at least HashLen octets
            (usually, the output from the extract step)
   info     optional context and application specific information
            (can be a zero-length string)
   L        length of output keying material in octets
            (<= 255*HashLen)

Output:
   OKM      output keying material (of L octets)

The output OKM is calculated as follows:

N = ceil(L/HashLen)
T = T(1) | T(2) | T(3) | ... | T(N)
OKM = first L octets of T

where:
T(0) = empty string (zero length)
T(1) = HMAC-Hash(PRK, T(0) | info | 0x01)
T(2) = HMAC-Hash(PRK, T(1) | info | 0x02)
T(3) = HMAC-Hash(PRK, T(2) | info | 0x03)
...

(where the constant concatenated to the end of each T(n) is a
single octet.)
```

Slide 58

## Algorithm Section (RFC 5869)

- Contains detailed technical specification
- Describes the protocol/algorithm step by step
- Often includes:
  - Input/output parameters
  - Processing steps
  - Implementation requirements
  - Pay attention to the allowed parameter ranges
- Typically does *not* include error handling
- May contain pseudocode or formal specifications

Slide 59

## Test Vectors

**Appendix A. Test Vectors**

This appendix provides test vectors for SHA-256 and SHA-1 hash functions [SHS].

**A.1. Test Case 1**

Basic test case with SHA-256

```
Hash = SHA-256
IKM  = 0x0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b (22 octets)
salt = 0x000102030405060708090a0b0c (13 octets)
info = 0xf0f1f2f3f4f5f6f7f8f9 (10 octets)
L    = 42

PRK  = 0x077709362c2e32df0ddc3f0dc47bba63
       90b6c73bb50f9c3122ec844ad7c2b3e5 (32 octets)
OKM  = 0x3cb25f25faacd57a90434f64d0362f2a
       2d2d0a90cf1a5a4c5db02d56ecc4c5bf
       34007208d5b887185865 (42 octets)
```

Slide 60

Test vectors are crucial for verifying your implementation. We discuss them later in Section 6.2 in more detail.

## References

**7. References**

**7.1. Normative References**

[HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997.

[KEYWORDS] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[SHS] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-3, October 2008.

**7.2. Informative References**

[1363a] Institute of Electrical and Electronics Engineers, "IEEE Standard Specifications for Public-Key Cryptography - Amendment 1: Additional Techniques", IEEE Std 1363a-2004, 2004.

[800-108] National Institute of Standards and Technology, "Recommendation for Key Derivation Using Pseudorandom Functions", NIST Special Publication 800-108, November 2008.

Slide 61

When implementing a standard, pay special attention to normative references as they contain essential requirements, while informative references provide helpful context and background information.

## Errata

**Status: Reported (1)**

RFC 5869, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", May 2010

Source of RFC: IETF - NON WORKING GROUP
Area Assignment: sec

Errata ID: 5161
Status: Reported
Type: Editorial
Publication Format(s) : TEXT
Reported By: Dale R. Worley
Date Reported: 2017-10-20

Section 2.3 says:

```
(where the constant concatenated to the end of each T(n) is a
single octet.)
```

It should say:

```
(where the constant concatenated to the end of each T(n) is a
single octet with value mod(n, 256).)
```

Notes:

It's clear what the values of the octets are supposed to be, but the text doesn't actually say what they are.

Slide 62

Always check the errata before implementation as they might contain critical corrections that were

not part of the original document.

## How NIST Competitions Work

- Open call for submissions from the cryptographic community
- Multiple rounds of evaluation:
  - Security analysis by cryptographers worldwide
  - Performance benchmarking across platforms
  - Implementation characteristics and complexity
  - Public feedback and discussion
- Candidates may be eliminated due to:
  - Security vulnerabilities discovered
  - Poor performance characteristics
  - Implementation difficulties
- Final selection based on balance of security, performance, and practicality

Slide 63

The NIST competition process exemplifies the careful balance between security, performance, and practicality in cryptographic standards. Let's look at a recent example.

## NIST Post-Quantum Cryptography Standardization

- Started in 2016 to standardize quantum-resistant cryptographic algorithms
- Goal: Protect against both classical and quantum computer attacks
- Focus on:
  - Key-establishment mechanisms
  - Digital signatures

Slide 64

## Selection Process Timeline

- **Round 1 (2017)**: 69 candidates accepted
  - 14 published attacks
  - 9 submissions withdrawn
- **Round 2 (2019)**: 26 candidates selected
- **Round 3 (2020)**: 7 finalists + 8 alternates
- **Round 4 (2022-2023)**:
  - Selected CRYSTALS-Kyber for key encapsulation
  - Selected CRYSTALS-Dilithium, FALCON, and SPHINCS+ for digital signatures

Slide 65

### Case Study: Dual_EC_DRBG

- NIST SP 800-90A standardized Dual_EC_DRBG in 2006
- Concerns raised about potential backdoor:
  - Outputs "too many" bits
  - Unclear choice of parameters $P$ and $Q$
  - Observer might learn internal state of RNG
- Snowden leaks in 2013 suggested NSA involvement
- NIST withdrew the standard in 2014

Slide 66

The Dual Elliptic Curve Deterministic Random Bit Generator (Dual_EC_DRBG) case highlights the importance of transparency and public scrutiny in standardization. Even after concerns were raised about its design, it took years and external events to trigger its removal. This experience led to increased emphasis on explaining design choices and parameter selection in newer standards, e.g. for X25519. The principle of *nothing-up-my-sleeve* is now widely accepted in cryptographic standards. For example, the choice of the prime field $2^{255} - 19$ and curve parameter $A = 486662$ for X25519 are carefully justified and documented [Bernstein, 2006]. The quotes below are taken from Bernstein's Curve25519 paper.

### Case Study: Choice of $A$ in X25519

> *"To protect against various attacks discussed in Section 3, I rejected choices of $A$ whose curve and twist orders were not $\{4 \cdot \text{prime}, 8 \cdot \text{prime}\}$; here 4, 8 are minimal since $p \in 1 + 4\mathbb{Z}$. The smallest positive choices for $A$ are 358990, 464586, and 486662. I rejected $A = 358990$ because one of its primes is slightly smaller than $2^{252}$, raising the question of how standards and implementations should handle the theoretical possibility of a user's secret key matching the prime; discussing this question is more difficult than switching to another $A$. I rejected 464586 for the same reason. So I ended up with $A = 486662$."*

Bernstein 2006, p. 14f

Slide 67

## 3.2 Error handling

## Error handling

- Error handling is crucial for production software
- Still, development is often focussed on the happy path
- In cryptographic software, we need to handle errors carefully
  - Indicate benign failures (e.g. out-of-memory)
  - Indicate malicious interference (e.g. invalid signature)

Slide 68

## Approaches to error handling

I introduce three main paradigms for error handling that you will encounter in different languages:

- Return values (C, C++, ...)
- Exceptions (Java, Python, ...)
- Result types (Rust, Go, ...)

These interact a lot with the language's type system as well. And we will discuss these aspects in more detail later.

Slide 69

## C-style error handling

Declared as such:

```
int decrypt_aes_gcm(
    uint8_t* key,
    uint8_t* ciphertext, size_t ciphertext_len,
    uint8_t* plaintext, size_t plaintext_len
);
```

Used as such:

```
plaintext = malloc(...)
if (decrypt_aes_gcm(&key, &ciphertext, &plaintext)) {
    // do something here
}
```

Slide 70

Classic C-style error handling relies on return values (usually `int`) to indicate success or failure. Typically we use `0` for success and non-zero for failure—this is convenient, because it implicitly casts to `true` or `false`. However, it is generally not very expressive and only allows for a single error code to be conveyed.

## C-style error handling

However, it's error prone:

```
plaintext = malloc(...)
decrypt_aes_gcm(&key, &ciphertext, &plaintext)
// do something here
```

Slide 71

More critically, it is very easy to forget to check the return value. This might be relatively easy to detect in code like the one above, where failure is quite obvious if the plaintext we receive is not valid. However, in more complex code, it might be much harder to detect.

## C-style error handling

```
int random_key(uint8_t* key)

uint8_t* key;
random_key(&key)
// ....
uint8_t* ciphertext = malloc(...);
if(aes_gcm_encrypt(&key, &plaintext, &ciphertext)) {
    // send ciphertext over the internet
}
```

Slide 72

One particularly dangerous area are values and conditions that are difficult for humans to detect. For instance, that variables are properly initialized with random values. The same is true for code paths that only rarely fail, e.g. due to out-of-memory conditions.

## C-style error handling

```
int random_key(uint8_t* key)

uint8_t* key;
if (!random_key(&key)) {
    // handle error
    return
}
// ....
uint8_t* ciphertext = malloc(...);
if(aes_gcm_encrypt(&key, &plaintext, &ciphertext)) {
    // send ciphertext over the internet
}
```

Slide 73

## C-style error handling: lessons learned

- Making error checking optional is dangerous
- Relying on humans to follow patterns is infeasible
- We need help from our tools!
  - Static analysis (e.g. `-Wunused-result`)
  - Linting (e.g. `clang-tidy`)
  - Or, maybe using better paradigms...

Before we look at modern paradigms, let's see how we can deal with this in C in our call sites.

Slide 74

While we would avoid writing and using code that follows this pattern, sometimes we have no choice. This is often the case for embedded systems, where proprietary build chains keep us from easily switching to better paradigms. Similarly, many underlying cryptographic libraries are still written in C. Where there is no existing binding, we have to deal with their paradigms in our own binding code.

## C-style error handling: writing better call sites

```
int decryptProtocolMessage(...) {
    if (checkSignature(&otherPublicKey, &ciphertext)) {
        uint8_t key = malloc(16);
        if (dh(&privateKey, &otherPublicKey, &key)) {
            if (decrypt(&key, &ciphertext, &plaintext)) {
                free(key)
                return OK
            } else {
                free(key)
                return ERR_DECRYPTION_FAILED
            }
        } else {
            free(key)
            return ERR_DH_FAILED
        }
    } else {
        return ERR_SIGNATURE_CHECK_FAILED
    }
}
```

Slide 75

Using nested `if` statements tends to be error-prone and leads to hard to reason code. For instance, it creates heavily indented code, which is hard to read. Also, the error conditions, i.e. the `else` branches, are often far away from the call site.

## C-style error handling: writing better call sites

```
int decryptProtocolMessage(...) {
    if (!checkSignature(&otherPublicKey, &ciphertext)) {
        return ERR_SIGNATURE_CHECK_FAILED
    }

    byte[] key = malloc(16);
    if (!dh(&privateKey, &otherPublicKey, &key)) {
        free(key)
        return ERR_DH_FAILED
    }

    if (!decrypt(&key, &ciphertext, &plaintext)) {
        free(key)
        return ERR_DECRYPTION_FAILED
    }

    free(key)
    result = &plaintext
    return OK
}
```

Slide 76

Using early returns is a good way to avoid the problems of nested `if` statements. It also makes the code easier to read and understand. However, we still have to be careful to not forget about freeing resources or other cleanup tasks.

## C-style error handling: writing better call sites

```
int decryptProtocolMessage(...) {
    int err = -1
    byte* key = NULL
    byte* plaintext = NULL

    if ((err = checkSignature(&otherPublicKey, &ciphertext)) != 0)
        goto fail;

    key = malloc(16);
    if ((err = dh(&privateKey, &otherPublicKey, &key)) != 0)
        goto fail;

    if ((err = decrypt(&key, &ciphertext, &plaintext)) != 0)
        goto fail;

    fail:
        if (key) free(key)
    done:
        return err
}
```

Slide 77

By using a `goto` statement, we can avoid the problems of nested `if` statements. If consistently used, it can make the code easier to read and understand. However, it is also much easier to introduce bugs and errors by anyone not familiar with the pattern, as the long variable lifetimes limit how much the compiler can help us.

## Case study: goto fail (CVE-2014-1266)

```
static OSStatus
SSLVerifySignedServerKeyExchange(/* ...*/)
{
        OSStatus        err;
        /* ... */
        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                goto fail;
        /* ... */
fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

Slide 78

One famous example of a bug due to the `goto fail` pattern occurred in Apple's iOS and macOS in 2014, breaking signature validation and making it possible to spoof TLS servers. Note the duplicate `goto` statements: only the first one is part of the `if` statement, the second one is not. Hence, the `goto` statement is executed unconditionally. Since the `err` variable is set within the `if` statement, we will (almost always) return `0`, hence success, from the function without executing the other tests at all.

## Exception-based error handling

Declared as such:

```
byte[] decrypt(byte[] key, byte[] ciphertext)
throws DecryptionFailedException {
    // ...
}
```

Used as such:

```
int doSomething() {
    try {
        plaintext = AesGcm.decrypt(key, ciphertext)
        // do something
    } catch (DecryptionFailedException e) {
        // handle error
    }
}
```

Slide 79

The challenges of using return values has led to adoption of exception-based error handling in many languages as a first-class citizen, most notably Java. It is also used in languages like Python, and as such worth a consideration for your assignments.

## Exception-based error handling

```
int doSomething() {
    try {
        plaintext = AesGcm.decrypt(key, ciphertext)
        doSomethingWithPlaintext(plaintext)
        andSomeOtherThings(plaintext)
    } catch (DecryptionFailedException e) {
        // handle error
    } catch (OtherException e) {
        // handle error
    } catch (AndAnotherException e) {
        // handle error
    }
}
```

Slide 80

One risk of exception-based error handling is that it can lead to a blow-up of error handling cases where the callsite grows—making it hard to maintain consistency.

## Exception-based error handling

```
int doSomething() {
    try {
        plaintext = AesGcm.decrypt(key, ciphertext)
        doSomethingWithPlaintext(plaintext)
        andSomeOtherThings(plaintext)
    } catch (Exception e) {
        // super defensive!!11
    }
}
```

Slide 81

In turn, this might cause the opposite reaction: developers might be tempted to catch all exceptions in an attempt to avoid having to handle them individually. That often leads to *exception swallowing*, where exceptions are simply ignored.

## Exception-based error handling

```
int doSomething() throws Exception {
    plaintext = AesGcm.decrypt(key, ciphertext)
    doSomethingWithPlaintext(plaintext)
}

int main() throws Exception {
    try {
        doSomething();
    } catch (Exception e) {
        // handle error
        // - but what exactly should we do?
        // - benign or malicious error?
    }
}
```

Slide 82

Alternatively, the callee might decide to simplify their job by throwing a generic exception. However, this often breaks the understanding of which exceptions are expected and which are not. Similarly, it makes it difficult to correctly handle the exceptions in the caller, as a lot of the relevant context is lost.

## Result types for error handling

Declared as such:

```
fn decrypt_aes_gcm(key: &[u8], ciphertext: &[u8])
    -> Result<Vec<u8>, DecryptionError>
```

Used as such:

```
fn do_something(&self) -> Result<(), Error> {
    let maybe_text = decrypt_aes_gcm(&key, &ciphertext);
    match maybe_text {
        Ok(text) => { /* do something */; Ok(()) },
        Err(err) => Err(Error::from(err, "aes gcm failed"))
    }
}
```

Slide 83

The third paradigm is result types, which are popular in languages like Rust, Go, and Swift. Here, functions return a `Result<T, E>` type, where `T` is the type of the value we want to return and `E` is the type of the potential error. By explicitly requiring the caller to unwrap the result, we avoid the problems of gathering many exceptions at single choke points or mindlessly passing them upwards. Second, they typically encourage more detailed typing for the errors.

## Result types for error handling (ergonomics)

Using `let-else` for error handling:

```
fn do_something(&self) -> Result<(), Error> {
    let Ok(text) = decrypt_aes_gcm(&key, &ciphertext) else {
        return Err(MyDecryptionError::DecryptionFailed);
    };
    /* do something with text */;
    Ok(())
}
```

Using `anyhow` for error handling:

```
fn do_something(&self) -> anyhow::Result<()> {
    let text = decrypt_aes_gcm(&key, &ciphertext)?;
    /* do something */;
    Ok(()) // No need for return keyword
}
```

Slide 84

While handling each result type individually can be cumbersome (looking at you, Go!), some languages have features that improve ergonomics. In Rust, we can use `let-else` to handle result types. A similar feature is available in other languages, e.g. `guard let` in Swift. Other libraries, like `anyhow`, provide a more ergonomic way to handle result types—providing an approximation of exception-based error handling.

Note that a big difference between exceptions and result types is also their underlying implementations. While exceptions often rely on the ability to unwind the call stack, result types can be implemented using a simple return value. That makes result types typically more efficient and robust—at the cost of being more verbose. We are focused on overall API design, so we will not dive deeper into the implementation details of the three different paradigms.

## Error messages

Consider the following code:

```
cipher = AesGcm.create(key)
plaintext = cipher.decrypt(ciphertext)
# do something with the plaintext
```

Let's assume the library is generally following a exception-based error handling approach. What behaviors would be good? Helpful? Unhelpful? Dangerous?

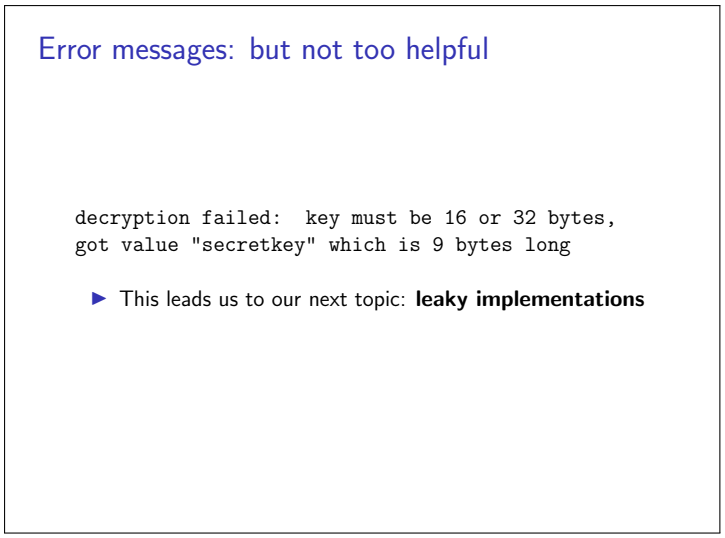
Slide 85

## Error messages: be helpful

- `decryption failed`
  - No information about what went wrong
  - No guidance on how to fix it
- `decryption failed: bad key length`
  - Indicates the specific issue
  - Still lacks guidance on how to fix it
- `decryption failed: key must be 16 or 32 bytes, but was 128 bytes`
  - Clearly states what went wrong
  - Provides exact requirements
  - Shows the actual problematic value

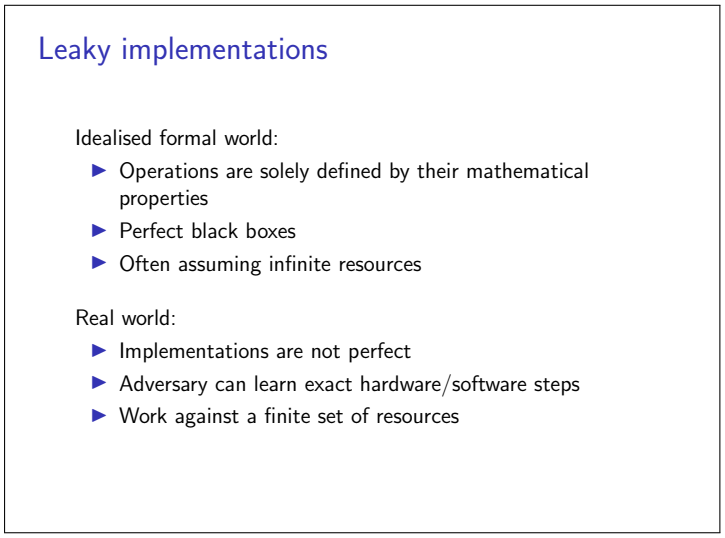
Slide 86

Creating great error messages is hard and it requires careful balancing between conciseness, helpfulness, and safety. Consider the setting in which the error is seen and by whom. Is it the developer while writing or debugging the call site? Then we want to be rather technical, helpful, and provide as much information as possible. Is it the end-user using the software that integrates our code? Then we want it to be "friendly", i.e. not technical, and provide references they can forward to the developer. For achieving this balance, a library might decide to change its behaviour based on its environment.



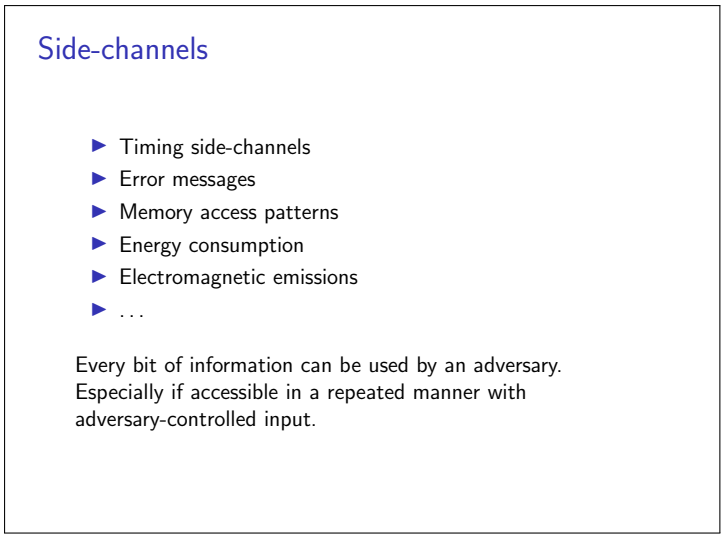

Slide 87

## 3.3 Leaky implementations



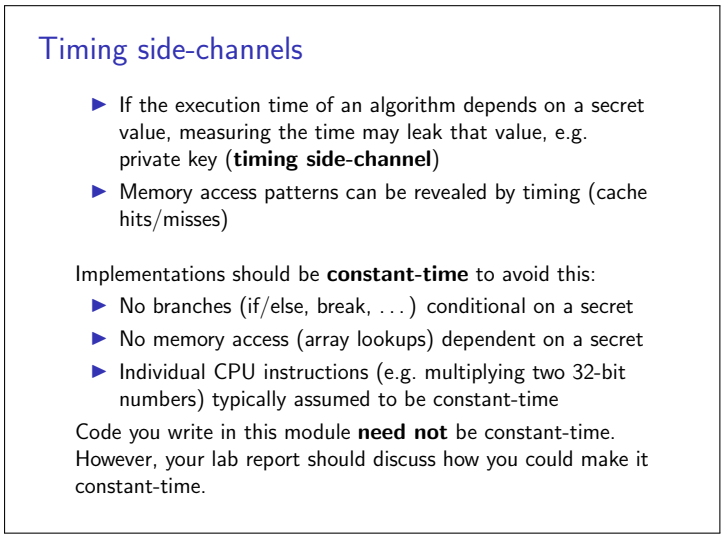

Slide 88

When moving from the idealised world to the real world, we need to be aware that our code will never be a perfect implementation of the mathematical properties. Mathematical operations are instant and do not leak any information, whereas in practice each operation is a sequence of discrete hardware/software steps that can be observed. The multiplication of two curve points, very easily translates into thousands of underlying operations such as: loading instructions from memory, comparing registers, performing divisions, .... As such, code will necessarily provide an attack surface for an adversary that is otherwise not visible in the mathematical specification. Hence, this advantage can be used by an adversary to learn about the internal state, e.g. secrets, of our implementation at runtime.




Slide 89

We call these side-channels, because they allow an adversary to learn information about the internal state of our implementation without directly accessing it. Today we will focus on timing side-channels and error messages. However, many other side-channels exist, e.g. memory access patterns, branch prediction, energy consumption, electromagnetic emissions—even the flickering LED on a network device might leak information. In your exercises you are not required to make your implementation side-channel resistant. However, you should be aware of the challenges and potential pitfalls and discuss them in your lab report.



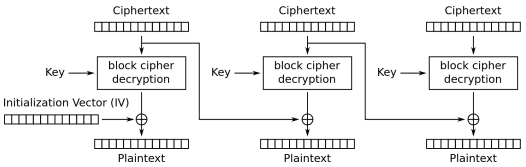

Slide 90

Production-quality code is often expected to be *constant-time*, which means that it always executes in the same amount of time, regardless of the value of any secret inputs. This is important since even small timing variations can result in a timing side-channel that is sufficient to allow an adversary to recover a private key [Genkin et al., 2017].

Making code constant-time in practice can be rather fiddly [Pornin, 2017], and therefore code you write in this module *does not need to be constant-time*. However, as part of your critical reflection in your lab report you should discuss the ways in which your code's timing is secret-dependent, and suggest ways how you could make it constant-time in the future.

## Padding oracle attack

- Assume AES-CBC with PKCS#7 padding
  - Padding pattern: `??  ??  ??  ??  04 04 04 04`
  - Decryption: $P_i = D_K(C_i) \oplus C_{i-1}$ and $C_0 = IV$




- Server returns either *nothing* or `PADDING_ERR`

Figure: Wikipedia, public domain.

Slide 91

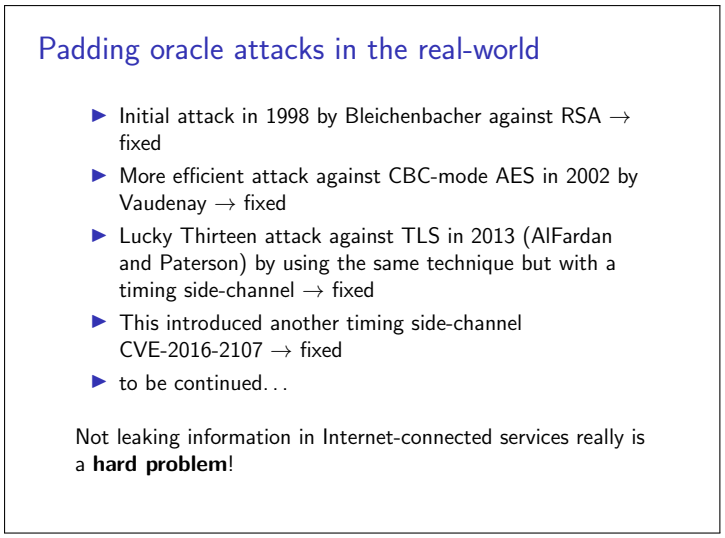
Padding oracle attacks are a classic example of a error message based side-channel that leaks 1-bit of information per adversary-controlled attempt. Using just the availability of an error messaging indicating whether the padding is correct or not, we can recover the plaintext of the entire message.

Let's first recap how PKCS#7 padding works. PKCS#7 padding is a scheme for a block cipher that pads messages to a multiple of the block size by appending bytes with a value equal to the number of padding bytes needed. For example, if a message needs 3 bytes of padding to reach the next block boundary, three bytes each with value `0x03` are appended. If the message is already aligned with the block size and we use 16-byte blocks, we append a full block of `0x10` repeated 16 times.

Let $C_1, C_2, C_3$ be three blocks of our correctly-padded CBC-encrypted ciphertext as per the diagram above in slide 91. Let $C_0$ be the Initialization Vector (IV). We first want to recover the plaintext $P_3$ for which we know that the last bytes are a valid padding, i.e. `0x01` or `0x02 0x02` or ....

We start our recovery attempt by assuming an initial padding where the last byte is `0x01`. Based on the XOR operation, we can set the last byte of $P_3$ to `0x02` by setting $C_2[15] = C_2[15] \oplus \texttt{0x01} \oplus \texttt{0x02}$. Now we try all possible values for the second-to-last byte of $P_3$ and check if we find a value that results in a valid padding. If we do, then our assumption of the `0x01` padding was correct. If not, our assumption of the initial padding was wrong and we try again with the next possible padding, i.e. `0x02 0x02`. For this we change the last bytes to `0x03 0x03` and repeat the same process. It will take us $16 \cdot 256$ attempts to find the correct padding length. Through this we now definitely know the last bytes of the plaintext $P_3$. That means we can take our known padding, increase all values by one and try to make the first byte before the padding such that it becomes part of the longer padding.

For example, we found that the last bytes of $P_3$ are `0x02 0x02`. We then change the last bytes to `0x03 0x03` and try to find the correct padding. If it is directly a valid padding, we know that the first byte before the padding must have been `0x03`. Otherwise, we try all variants for $C_2'[16-3]$ until we have a match. We can then compute the original plaintext $P_3[16-3] = C_2'[16-3] \oplus C_2[16-3] \oplus \texttt{0x03}$. This analogously applies for all possible padding lengths and the other bytes of $P_3$. By repeatedly applying this process byte for byte we can recover the entire plaintext.

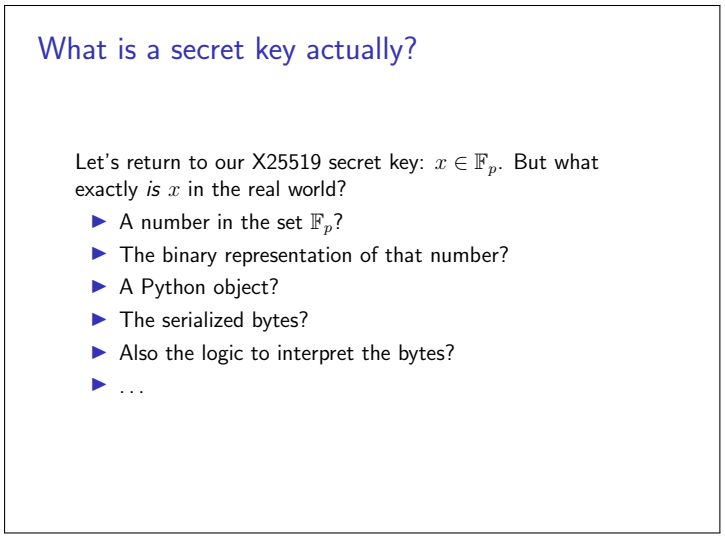



Slide 92

More details are in the original papers by Bleichenbacher [1998], Vaudenay [2002], and Al Fardan and Paterson [2013] as well as in this write-up of the details of CVE-2016-2107: https://blog.cloudflare.com/yet-another-padding-oracle-in-openssl-cbc-ciphersuites/.

## 3.4 Types

Types improve the robustness of cryptographic libraries and make it easier for developers to use them correctly. In this section we will discuss how static typing can move many correctness guarantees to the compilation time. We will discuss in later lectures more involved patterns including *Type State* that ensure we not only pass in correct types, but can capture important logical guarantees throughout the lifetime of our programs.




Slide 93

## Example: X25519 key

```
x: int
```

- Simple
- Confusion possible with different keys
- Unclear how to turn into/from bytes
  - This process is often called serializing or marshalling

Slide 94

## Example: X25519 key

```
x: bytes
```

- Relatively simple
- Still able to confuse with different keys
- Might be too short/long

Slightly better in languages, e.g. Rust, with fixed-sized arrays:

```
x: [u8; 32]
```

Slide 95

## Example: X25519 key

```
@dataclass
class X25519SecretKey:
    x: int
```

- Strong type avoids mixing up different keys
- Can have extra functionality, e.g. comparison
- Can have extra checks, e.g. avoid uninitialized value

Slide 96

Using strong types helps prevent common mistakes and makes the code more maintainable. However, we need to carefully consider how these types interact in more complex scenarios where multiple components work together.

## Going past one type

This sounds great at first. However, it becomes tricky when integrating multiple components, e.g. asymmetric key agreement and symmetric encryption.

```
def enc(
    x: X25519SecretKey, gy: X25519PublicKey,
    message: bytes,
) -> bytes:
    shared = x.dh(gy)  # probably just bytes
    aes_key = CipherLib.AESKey(shared)
    # encrypt message and return
```

Slide 97

Types might also come into play when we want to pass a result of one cryptographic operation, e.g. an asymmetric key agreement like X25519, into a next one, e.g. as a key to a symmetric encryption. By avoiding weaker types, e.g. `bytes`, we can prevent many potential error sources. Since cryptographic operations usually can not be easily inspected, many wrong usages "work" but with devastating consequences. For example, the developer might pass in the hexadecimal formatted output of a digest (easy to do with some Python libraries) directly into the key function of a cipher. They might expect to receive 128-bit security with a 16 byte long input, however, they would effectively only get 64 bit security as each byte only takes the value `0-9A-F`.

## Going past one type

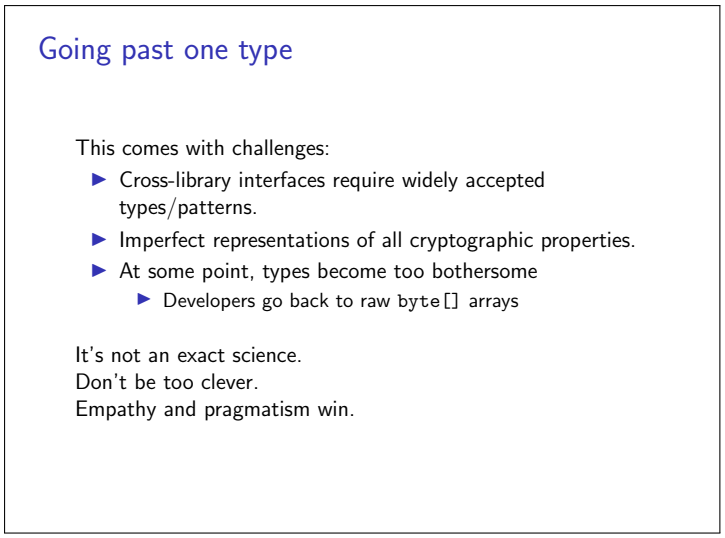
We might want to use more types to describe these boundaries. For example:

- `DerivedSecret`: result from a DH operation
- `SecureRandom`: anything that gives us high-entropy random numbers

```
KeyMaterial = DerivedSecret | SecureRandom
class AesKey:
    def __init__(self, key: KeyMaterial):
        self.key = key
```

Slide 98

Strong types can help us to mitigate some of these risks. For instance, we could ensure that an AES key is never directly derived from a byte array, but only from high-entropy output of either a cryptographically-secure random number generator or from the result of a key exchange. We might go even further and e.g. require that the type needs to be promoted further by using a suitable key derivation function (KDF). We discuss KDFs and why using them is a good idea in more detail in a later lecture.



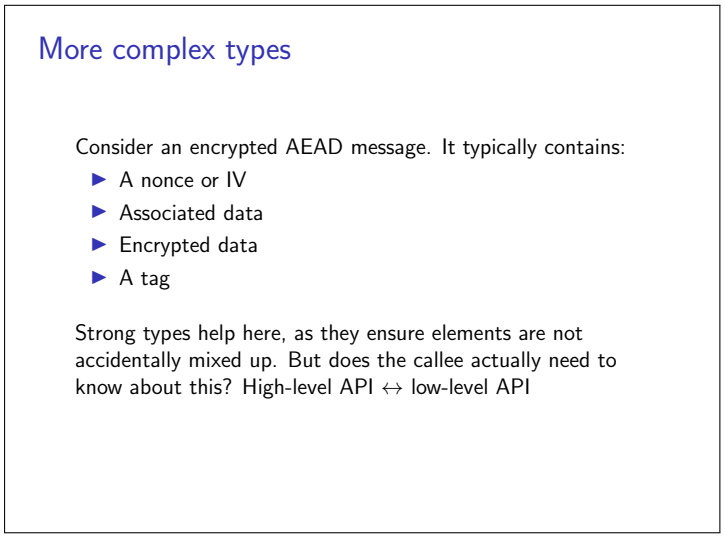

Slide 99

At large, the idea of adding zero-cost abstractions and moving more guarantees to the compilation stage (or static checking) is great and promising. However, we must not overlook its challenges.

For example, real-world protocols and implementations often rely on multiple cryptographic libraries to work together. Strong types will only work if both libraries are build upon the same type hierarchy provided either by the language's standard library or a widely-accepted library. One example is the Java Cryptography Architecture (JCA, `java.security`) which defines a type hierarchy for cryptographic operations. However, such frameworks tend to be difficult to extend and upgrade later, e.g. when adding new primitives such as hybrid encryption with a different set of parameters. As a result, these retrofitting new ideas often result in frankenstein-ish libraries that are hard to maintain and use.

Even where types are aligned, they typically are imperfect representations of the underlying cryptographic properties. For example, the bias in a result, e.g. from X25519, might have implications when used as a key for some ciphers, but might be perfectly fine for others.

Adding more and more information to types might also lead to a situation where the design becomes too cumbersome to use. When leaving the carefully designed path, developers might find that their only workable solution is to use the available escape hatches and fall back to raw `byte[]` arrays. Overall, too complex type systems can distract from the actual cryptographic reasoning and make the engineer focus primarily on "making the type checker happy". Other drawbacks include hard-to-parse error messages, hard-to-maintain extensions that need updating with the imported type hierarchy, and prolonged compilation times.




Slide 100

The following slides provide a crash course on how to use `ty` to statically type check your Python code. You'll find it valuable to experiment with type checking in your exercises and explore the expressive capabilities of Python's type system. If you're already familiar with statically typed languages like Rust, you'll notice interesting differences in what various type systems can express.

## Static Type Checking with ty

- ty is a static type checker for Python
- Helps catch type errors before runtime
- Popular alternatives:
  - mypy
  - Pyright (from Microsoft)
  - Pyre (from Meta/Facebook)
  - Pytype (from Google)
- Easy to install and configure:

```
# Install ty:
uv add --dev ty

# Run ty:
uv run ty check your_file.py
uv run ty check .
```

Slide 101

For more information about ty, see https://docs.astral.sh/ty/. We chose ty for this course because it is easy to use, provides helpful error messages, and is very fast. However, you might find other type checkers more convenient for your use case.

## Basic Type Annotations

- Start with built-in types
- Use generic types from the typing module
- Type checking works with nested types

```
from typing import List

def sum_numbers(numbers: List[int]) -> int:
    total: int = 0
    for num in numbers:
        total += num
    return total

# OK
result = sum_numbers([1, 2, 3])

# type error: List[str] not assignable to List[int]
bad_result = sum_numbers(["1", "2", "3"])
```

Slide 102

## Runtime Type Checking

- Type hints are not automatically checked at runtime
- Use assertions for runtime checks

```
from typing import List
import typing

def sum_numbers(numbers: List[int]) -> int:
    assert isinstance(numbers, list),\
        "numbers must be a list"
    assert all(isinstance(x, int) for x in numbers),\
        "all elements must be integers"

    total: int = 0
    for num in numbers:
        total += num
    return total
```

Slide 103

The Python typing system is designed to be gradual—you can add types incrementally to your code-

base. For more ergonomic runtime type checking, consider using libraries like `typeguard` or `beartype`.

## Advanced Type Features

- TypeAlias: Create aliases for complex types
- Union: Allow multiple types
- Modern Python also supports | syntax for unions

```python
from typing import TypeAlias, Union, List

Matrix: TypeAlias = List[List[float]]
Number: TypeAlias = Union[int, float]

def add_constant(matrix: Matrix, c: Number) -> Matrix:
    return [[x + c for x in row] for row in matrix]

# Both valid
result1 = add_constant([[1.0, 2.0]], 1)    # [[2.0, 3.0]]
result2 = add_constant([[1.0, 2.0]], 0.5)  # [[1.5, 2.5]]
```

Slide 104

## Self-referential Types

- `Self` type for methods returning instances
- Useful for creation methods and operations

```python
from typing import Self

class Matrix:
    def add(self, other: Self) -> Self:
        return Matrix([
            [self.data[i][j] + other.data[i][j]
            for j in range(self.cols)]
            for i in range(self.rows)
        ])

    @classmethod
    def zeros(cls, rows: int, cols: int) -> Self:
        return cls([[0.0] * cols for _ in range(rows)])

m1 = Matrix.zeros(2, 2)
m2 = m1.add(m1)
```

Slide 105

The `Self` type is available since Python 3.11 and is used to refer to the class that is being defined. For example, in the code above, we cannot simply use `Matrix` in the return type of the `add` method, because `Matrix` is not defined yet. For more advanced typing features, including Protocol, TypeVar, and Generic types, refer to Python's typing documentation at https://docs.python.org/3/library/typing.html.

## Pattern: type state (1, motivating example)

```rust
struct AkeClient {
    x: Secret,
    k: Option<DerivedKey>,
    has_verified_server: bool,
}

impl AkeClient {
    fn handle_server_response(
        &mut self,
        response: ServerHelloResponse,
    ) -> Result<()> {
        if self.has_verified_server {
            bail!("already verified server");
        }
        // verify server response ...
        self.has_verified_server = true;
        self.k = Some(derive_key(&self.x, response));
        Ok(())
    }
}
```

Slide 106

The ends, e.g. client or server, of protocols often go through state transitions. For instance, in an mTLS-like handshake the client is first in an initialized state where they generate or load their secrets. Afterwards they send a message to the server and transition into a waiting state. Once they received a reply, they can derive the main session keys, send the next message to the server, and transition to a connected state.

The motivating example in Slide 106 shows an implementation approach that's very common. We can see that it is difficult to understand which methods modify what state. Also, there is no consistent way to ensure methods are called in the correct order—we only find out at runtime if we call methods in the wrong order or too often. Also, the secret `x` lives longer than it needs to and there will be considerable extra effort in dealing with an `Option<...>` type for the derived key.

## Pattern: type state (2)

```rust
struct AkeClientInitialized {x: Secret}
struct AkeClientWaiting {x: Secret}
struct AkeClientVerified {k: DerivedKey}

impl AkeClientWaiting {
    fn handle_server_response(
        self, response: ServerHelloResponse
    ) -> Result<AkeClientVerified> {
        // verify server response ...
        // derive key ...
        let k = derive_key(&self.x, response);
        Ok(AkeClientVerified {k})
    }
}
```

Slide 107

Instead, we can use the type system to our advantage using the Type State Pattern [Biffle, 2019]. By making the transition functions "consume our current state" and returning a new state with a different type, our state transition graph is modelled using the type system. This works particularly well with Rust's type system—however, similar patterns can provide comparable benefits in other languages. The Type State Pattern can also enforce additional constraints, e.g. that we process only one server response per connection attempt and chosen `x: Secret`.

## Pattern: type state (3)

```
struct AkeClient<S> { state: S }

trait AkeClientState {}
impl AkeClientState for AkeClientInitialized {}
impl AkeClientState for AkeClientWaiting {}
impl AkeClientState for AkeClientVerified {}

impl AkeClient<AkeClientWaiting> {
    fn handle_server_response(
        self, response: ServerHelloResponse
    ) -> Result<AkeClient<AkeClientVerified>> {
        // verify server response ...
        // derive key ...
        let k = derive_key(&self.state.x, response);
        Ok(AkeClient {state: AkeClientVerified {k}})
    }
}
```

Slide 108

Another implementation variant of the Type State Pattern uses generic types. This has the advantage of further reducing boilerplate code and allows us to have shared functions on the main `AkeClient` class. Also, using the `trait` can give more control over the extent to which users of our API can add new states. Finally, we can now also easily introduce common state, e.g. logging and protocol transcripts. Note that we want to move this onto the heap using `Box<...>` so that it is not copied around the stack unnecessarily.

## Pattern: type state (4)

```
struct SharedState {debug_log: Vec<ProtocolLogEntry>}
struct AkeClient<S> {
    state: S,
    shared: Box<SharedState>,
}

impl<S> AkeClient<S> where S: AkeClientState {
    fn log(&mut self, entry: ProtocolLogEntry) {
        self.shared.debug_log.push(entry);
    }
}
```

Slide 109

## Pattern: type state (5)

Benefits:

- Move important logic to compile time (or type checker)
- Improves IDE ergonomics
- Easier reasoning about state

For cryptography in particular:

- Clear management of secret life times
- Protections against accidental secret reuse

More examples for constraints that can be modelled:

- We can only access API methods after authentication
- Once a connection has been terminated, we cannot call any send/receive methods
- After A, both B and C have been called (in any order), before D can be called

Slide 110

## Pattern: type promotions

```
struct UnverifiedKey { bytes: [u8; 16]}
struct VerifiedKey { bytes: [u8; 16]}

fn load_root_of_trust(
    path_buf: &PathBuf,
    trusted_digests: &[u8]
) -> RootOfTrust {
    // check: date constraints && matches trusted digests
    return ...
}

fn verify_key(
     key: UnverifiedKey,
     certificate: &Certificate,
     root_of_trust: &RootOfTrust
) -> Result<VerifiedKey> {
    // check: date constraints && chained signature
    return ...
}
```

Slide 111

We can also use our type system to give us stronger guarantees about the validity of our secrets. In particular, we can "promote" keys to a verified state only when we have properly validated them. This ensures that we never accidentally forget check a signature. Also, by controlling the visibility of the constructing methods, we can help audits of the code as there are intentional bottlenecks in the logic flows.

We can extend the previous example by giving our keys a scope. For instance, keys might be assigned a `Role` (user or server) and/or a `Purpose` (signing or encrypting). Binding these to the types themselves makes it hard to use a key for the wrong operation. In Rust we have to use `PhantomData<T>` to convince the compiler that we are actually "using" the type parameter.

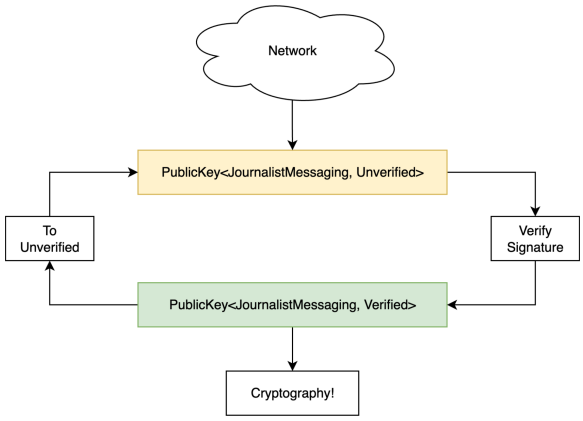
## Pattern: scoped secrets

```
struct ScopedUnverifiedKey<Role>
    { bytes: [u8; 16], _role: PhantomData<Role> }
struct ScopedVerifiedKey<Role>
    { bytes: [u8; 16], _role: PhantomData<Role> }

fn verify_key_for_role<S>(
    root_of_trust: &RootOfTrust,
    certificate: &ScopedCertificate<S>,
    key: &ScopedUnverifiedKey<S>
) -> Result<ScopedVerifiedKey<S>> where S:Role {
    // ...
}

fn sign_message_to_server(
    key: &ScopedVerifiedKey<Client>, msg: &[u8]
) -> Vec<u8> {
    // ...
}
```

Slide 112

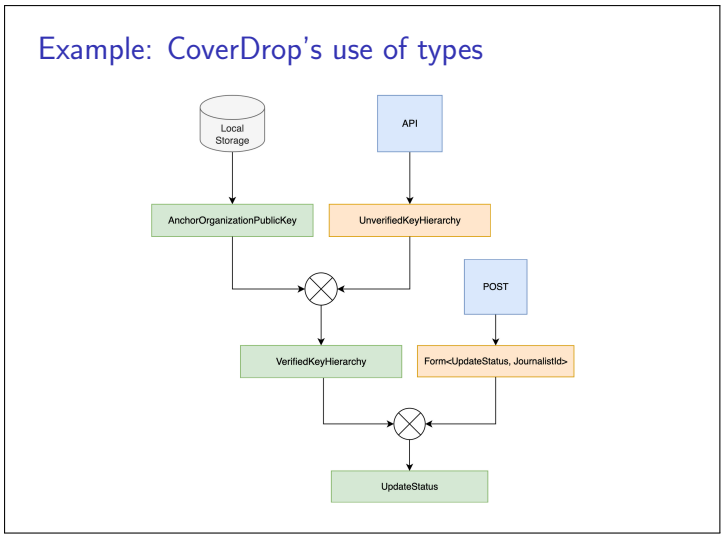
By combining type promotions and scoped secrets, we can create a powerful system that helps us manage our secrets correctly. The following example illustrates this concept using the (simplified) type hierarchy of the CoverDrop project [Hugenroth et al., 2025] – a secure whistleblowing system that was researched at our department and is now being deployed in practice.

## Example: CoverDrop's use of types




Slide 113

CoverDrop employs a relatively complex key hierarchy containing trust anchors, long-term intermediate signing keys, and short-lived message encryption keys. While the trust anchors are baked into the apps, the other keys are typically read from the network or a local cache. Using type promotion, the system ensures that keys can only be used after proper validation including signature checks and expiry date verifications. In addition to their verified/unverified state, key types in CoverDrop also have a role, e.g. `JournalistMessagingKey` or `CoverNodeIdentityKey`. This ensures that keys are only used for their intended purpose and prevents accidental misuse.

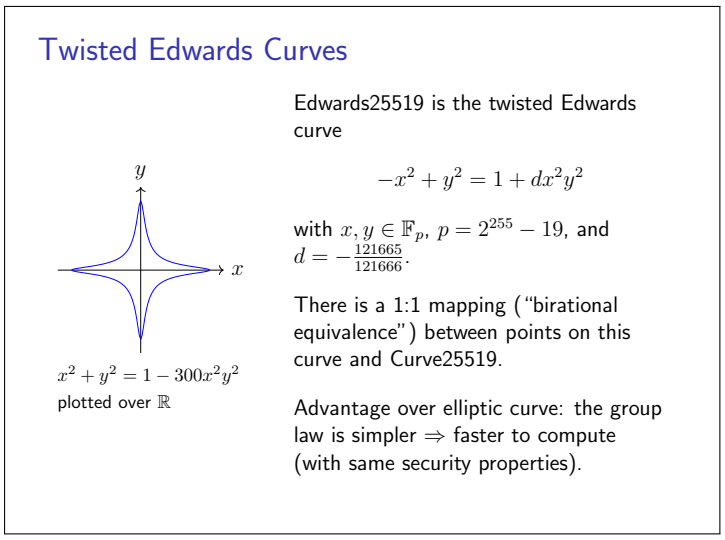



Slide 114

The type hierarchy extends to API calls as well. For example, if a journalist wants to update their availability status, they create a `Form<UpdateStatus,JournalistId>` object which is signed with their long-term identity key and then serialized as JSON for transmission via HTTPS. The server then deserializes the JSON into a `Form<UpdateStatus,JournalistId>` object and verifies the signature using the key hierarchy which is only available after proper validation due to the required type promotions. This results in the actual `UpdateStatus` object which is the only type accepted by the components interacting with the database. Hence, by seeing this type in a method signature, we can always be sure that the request was signed by a valid journalist, with a valid key, which is correctly cross-signed by higher levels of the key hierarchy, rooted in a trust anchor baked into the app.

# 4 Elliptic Curve Signatures

For the second assignment, you will implement a digital signature scheme called Ed25519. As you can guess from the number in the name, it is somewhat related to Curve25519. In fact, it uses the same field of integers modulo $p$, with $p = 2^{255} - 19$, but it uses a different curve equation called a *twisted Edwards curve* (which was discovered as recently as 2008). The concepts are very similar though, and the Edwards curve is equivalent to an elliptic curve. The twisted Edwards curve is used mainly because, compared to operations on Montgomery curves (Slide 32), it's faster and easier to make constant-time.




Slide 115

The group law is shown on Slide 116: you see that it's much simpler than what we saw on Slide 36. Another good thing about it is that we can prove that the denominators of the fractions are never zero – unlike the addition law on Slide 36, where the denominators go to zero when the points being added have the same $x$ coordinate, necessitating separate formulas for point doubling to handle this case. Having

separate addition and doubling formulas is problematic if you want to make the computation constant-time: if the time it takes to compute the addition formula is different from the time it takes to compute the doubling formula, then the timing leaks information on the $x$ coordinates of the points. Having a single, complete addition formula makes this particular side-channel go away.

Given the group law, we can then implement scalar multiplication using the same double-and-add approach as we saw on Slide 39. As a further optimisation, instead of computing a fraction every time you apply the group law (which requires computing the multiplicative inverse of the denominator, which is by far the slowest part of the group law), you can use *projective coordinates* (also known as *homogeneous coordinates*), which essentially store the numerator and denominator in two separate variables across all steps of the double-and-add algorithm. Once the double-and-add algorithm is finished, you can convert the projective coordinates back into regular (*affine*) coordinates by computing the multiplicative inverse of the final denominator value. You can find formulas for the twisted Edwards curve group law in projective coordinates in RFC 8032 [Josefsson and Liusvaara, 2017], and in the Explicit Formulas Database at http://www.hyperelliptic.org/EFD/g1p/auto-twisted.html.

### Group law on twisted Edwards curve

**Point addition** for curve $-x^2 + y^2 = 1 + dx^2y^2$:

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{x_1x_2 + y_1y_2}{1 - dx_1x_2y_1y_2}\right)$$

**Complete**: no need for seperate doubling formulas since the denominators are always non-zero. (Helps with constant-time)

Define **scalar multiplication** like on elliptic curve, using double-and-add.

Faster scalar product by working in **extended homogeneous (projective) coordinates**: instead of $(x, y)$ use $(X, Y, Z, T)$ where $x = X/Z$, $y = Y/Z$, $xy = T/Z$. See RFC 8032.

Compute the inverse of $Z$ only at the end of scalar product, not for every point addition.

Slide 116

Another thing we need for Ed25519 signatures is the concept of *point compression*, as shown on Slide 117. This applies to both elliptic curves and Edwards curves. For elliptic curves we usually encode the $x$ coordinate along with the sign bit of the $y$ coordinate. Ed25519 does it the other way round, encoding the $y$ coordinate along with the sign bit of the $x$ coordinate, but the idea is the same.

### Point compression

For X25519, we could get away with only using $x$ coordinates.

For signatures, we need the $y$ coordinate as well. But: sending both $x$ and $y$ coordinates doubles data size ($32 \to 64$ bytes).

For a given $x$ coordinate there are at most two possible values $\pm y$ such that $(x, y)$ lies on the curve.

**Point compression**: encode the $x$ coordinate along with one bit to say which $y$ value is used ("sign bit").

When $x \in \mathbb{F}_p$ for $p = 2^{255} - 19$, $x$ fits in 255 bits $\Rightarrow$ use the 256th bit for the sign of $y$, still fits in 32 bytes.

(Actually Ed25519 encodes $y$ coordinate and the sign of $x$.)

Define $y$ to be **positive if even, negative if odd**.
Note $-y \equiv p - y \pmod p$, so $y$ is even iff $-y$ is odd.

Then the "sign bit" is simply the least significant bit of $y$.

Slide 117

To decompress a compressed point, we need to recover the $y$ coordinate. We can to that by solving the curve equation on Slide 115 for $y$, resulting in the equation shown on Slide 118. (Note that to implement Ed25519 you need to instead solve the curve equation for $x$, but the result looks similar.)

Computing the $y$ coordinate requires a square root. Yes, square roots are also defined on finite fields!

The result of the square root is still an element of the field $\mathbb{F}_p$ (not a real number). We define $\sqrt{a} = b$ as a value such that $b^2 = a \pmod{p}$, and it can be computed as shown on Slide 118. The square root $\sqrt{a}$ is only defined if $a$ is a square in $\mathbb{F}_p$; if not, the square root computation fails. In the context of point decompression, this would happen if an adversary sends you an $x$ coordinate that does not correspond to any point on the curve. A correct implementation of point decompression must handle that error case (in the case of a signature, by returning that the signature is invalid).

The Tonelli-Shanks algorithm can be used to compute square roots. In general this algorithm is a bit complicated, but RFC 8032 specifies a simpler version that is specific to $p = 2^{255} - 19$. As Ed25519 only ever needs to compute square roots modulo $p$, and simpler is better, we recommend you use the algorithm from RFC 8032 for your implementation.

## Point decompression

**Point decompression**: take the low 255 bits (little-endian) as $x$, error if it's $\geq p$. Then

$$y = \pm\sqrt{\frac{1 + x^2}{1 - dx^2}} \quad \text{using } + \text{ or } - \text{ according to sign bit.}$$

How to compute **square root** modulo $p = 2^{255} - 19$:

$$\sqrt{a} = \begin{cases} a^{(p+3)/8} & \text{if } (a^{(p+3)/8})^2 = a \\ a^{(p+3)/8}\sqrt{-1} & \text{if } (a^{(p+3)/8})^2 = -a \text{ where } \sqrt{-1} = 2^{(p-1)/4} \\ \text{error} & \text{otherwise} \end{cases}$$

See RFC 8032, Tonelli–Shanks algorithm.

This also ensures that $(x, y)$ is a valid point on the curve.

Slide 118

With that background, we can now move on to the definition of the Ed25519 signature scheme. You can read more about Ed25519 in the original paper [Bernstein et al., 2012] and RFC 8032 [Josefsson and Liusvaara, 2017]. EdDSA/Ed25519 is designed to avoid some of the pitfalls with earlier signature schemes such as ECDSA, which are prone to implementation bugs [Madden, 2022]. In particular, ECDSA requires a unique nonce for every signature; if you ever sign two different messages with the same key and the same nonce, anybody can easily calculate the private key from those two signatures. This happened in the PlayStation 3, for example, inadvertently revealing Sony's signing key for software updates [fail0verflow, 2010]. Even a small amount of bias in an ECDSA nonce can allow private key recovery, and this has occurred recently in widely-used software [Tatham, 2024]. To avoid such risks, EdDSA does not require any random numbers for signing, and instead generates a pseudorandom value deterministically from a hash of the private key and the message.

## The Ed25519 signature scheme

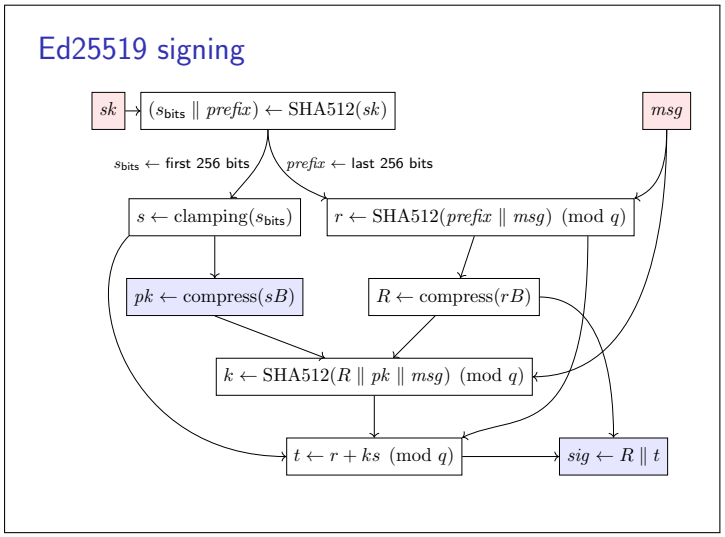
- EdDSA: general algorithm;
  Ed25519: EdDSA over Edwards25519 curve
- Very widely used, e.g. TLS 1.3, SSH, Signal
- $sk$ 32 bytes, $pk$ 32 bytes, signature 64 bytes
- Deterministic: signing requires no randomness, no nonce
  - Whereas ECDSA requires nonce per signature
  - Nonce reuse in ECDSA is catastrophic
  - Ed25519 computes $r$ from hash of private key and message ⇒ safer to use
- Variant of Schnorr signatures
- $B$ is base point with order $q$, where $|E| = 8q$ is group order (different from field order $p$)

Slide 119

The process for generating a signature is outlined on Slide 120. Inputs are shaded red, outputs

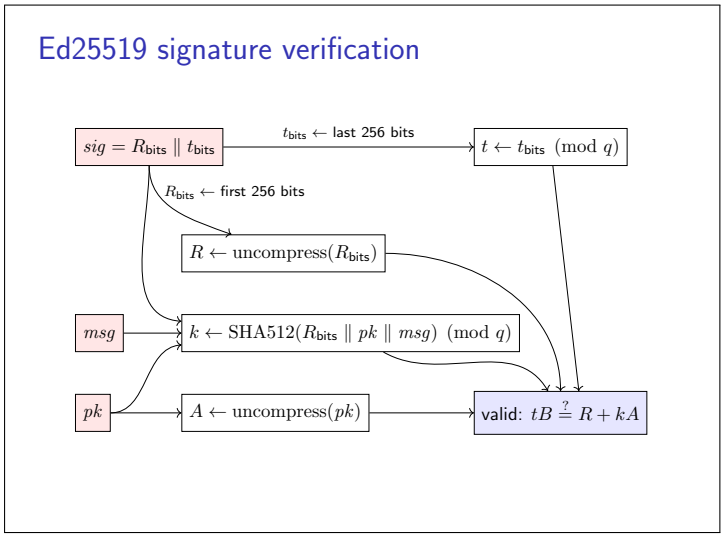
blue. In this module we don't have time to go into why these operations yield a secure digital signature scheme. If you want to understand this better, we suggest you read the section on Schnorr signatures in the textbook [Katz and Lindell, 2020], much of which applies to EdDSA as well.

On this slide, *clamping* refers to the practice of setting some bits to fixed values as explained on Slide 44 (RFC 8032 calls it "pruning the buffer"). The two occurrences of scalar multiplication are done on the Edwards25519 curve, followed by point compression. There is also some arithmetic in the field $\mathbb{F}_q$ of integers modulo $q$; note that this is different from the field $\mathbb{F}_p$ that is used for point coordinates! What we call $q$ is called `L` in RFC 8032, and $l$ in the original paper [Bernstein et al., 2012]. In some cases the output of the hash function needs to be interpreted as an integer $\in \mathbb{F}_q$; this is done by first interpreting the byte string as a number in little-endian encoding, and then reducing modulo $q$. Likewise, when $t$ is included in the signature, it is encoded as a little-endian 32-byte string. (The value we call $t$ is called $S$ in the paper and RFC; we use a lowercase letter since by convention lowercase letters usually refer to field elements, and uppercase letters to curve points, and the variable $s$ is already taken.) $\|$ denotes concatenation of byte strings.




Slide 120

The corresponding signature verification logic is shown on Slide 121. Verification should fail if the final equation is not satisfied, of any of the point decompressions fail, or if any value is outside of the allowed range (e.g., if the last 256 bits of the signature are the little-endian encoding of an integer $\geq q$).

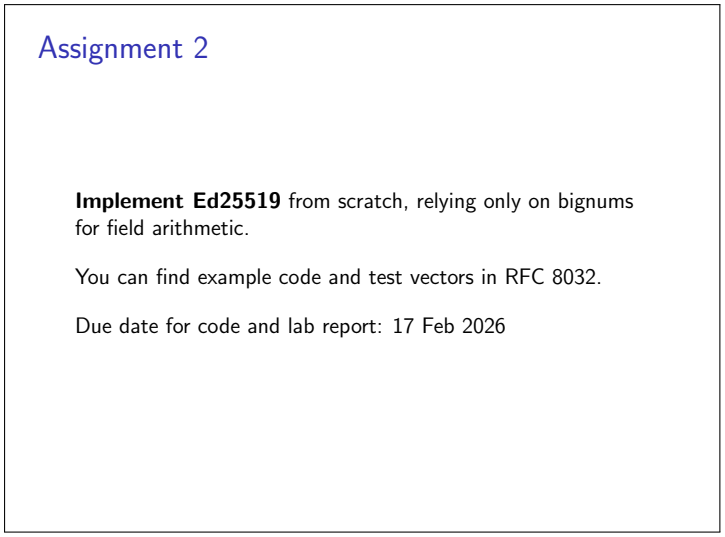



Slide 121

You now know enough about Ed25519 to go and implement it yourself. You can use RFC 8032 as a reference (including the Python code in Section 6); however, assignment submissions that are obviously just copied and pasted from the RFC are unlikely to score well. As a reminder of Slide 8, we want to see code that not only produces the right output, but which has a well-designed API, which is appropriately documented and tested, and extensions are also welcome. For example, you could look into the ambiguities

in the RFC 8032 specification [de Valence, 2020]. You can ignore the Ed25519ph and Ed25519ctx variants that are specified in RFC 8032.

It is important that your implementation performs all necessary validation on input values, since incorrect input validation is a common source of vulnerabilities [Denis, 2025, Schiffermuller, 2025]. Project Wycheproof [Bleichenbacher et al.] could be a useful source of test vectors.




Slide 122

# 5 Authenticated key exchange

Now that you have seen how to implement cryptographic primitives such as Diffie-Hellman and signatures, we can build protocols out of those primitives. The first thing we need to do when talking about a protocol is to define its *threat model*: that is, which participant in the protocol (including the network) is trusted or not trusted to do certain things? A common model is the *Dolev–Yao model*, in which the network is assumed to be completely untrusted: that is, the adversary is allowed to do anything with the messages that travel over the network. This is actually a reasonable model for communicating over the internet: for example, when you are connected to a random coffee shop wifi, the owner of the coffee shop (who controls the router that your communications pass through) is in a position to perform arbitrary passive and active attacks on your communications.

In principle, the Dolev–Yao model allows all messages to be dropped; of course, no protocol will be able to get anything done if no message is ever delivered. We therefore often assume *eventual delivery*: that is, the network protocol detects dropped messages and resends them (if necessary, multiple times) with the expectation that one of the retries eventually succeeds. The recipient may need to filter out duplicate messages resulting from such retries. For simplicity, we won't worry about retries and deduplication in this module. However, it's important that a dropped message only results in the protocol not terminating; the protocol must not violate its advertised security properties because a message was dropped.

## Cryptographic protocols

We've seen **cryptographic primitives**: symmetric encryption, hashes, Diffie-Hellman, signatures.

Now let's look at **protocols**: interactive communication between two or more parties, sending messages over a network.

**Threat model**: assume what each party may do / not do, and what the network may do / not do

Common assumption: **Dolev–Yao model**. The adversary...

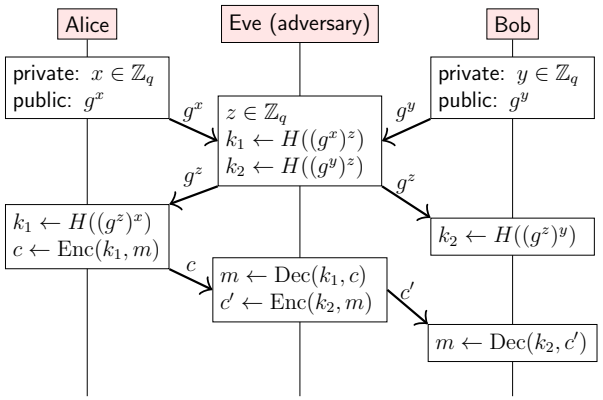
- sees everything sent over the network (eavesdropping)
- chooses if and when messages are delivered
- can inject fake messages, replay old messages (active)
- sees multiple protocol runs between different parties

Reasonable model if you're on a coffee shop wifi!

Slide 123

Under such a threat model the classic Diffie-Hellman protocol from Slide 20 is insecure. As shown on Slide 124, the adversary can intercept the $g^x$ and $g^y$ messages from Alice and Bob, and replace them with $g^z$ where the private key $z$ is known to the adversary. Since Alice and Bob cannot distinguish the adversary-generated $g^z$ from the genuine $g^x$ and $g^y$, they proceed with the protocol as usual. As a result, Alice and Bob think they share a key, but in fact Alice ends up sharing a key $k_1$ with the adversary, and Bob shares a different key $k_2$ with the adversary. The adversary can now decrypt the ciphertext $c$ from Alice and re-encrypt it under the key shared with Bob. As a result, the adversary learns the plaintext message $m$ and can even modify this message without Alice or Bob noticing.

## Unauthenticated Diffie-Hellman




Slide 124

This is an example of a *machine-in-the-middle* (MITM, also *person-in-the-middle*, or traditionally called *man-in-the-middle*) attack on the protocol. The problem is that plain Diffie-Hellman lets you establish a shared key with some other party, but it doesn't tell you who that party is – it could be anyone. To prevent it, we need to add *authentication*: the parties need to prove to each other who they are.

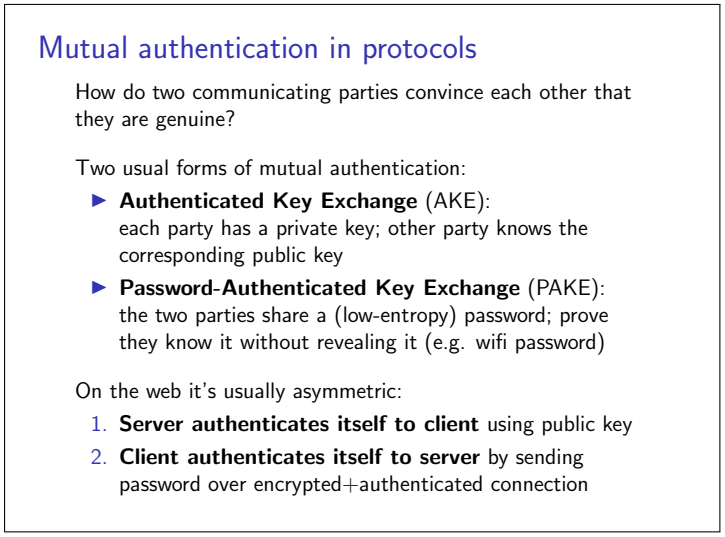
Even if the Enc/Dec functions on Slide 124 are from an authenticated symmetric encryption scheme, that's not sufficient, since symmetric authentication only proves that a message was encrypted by someone who knows the symmetric key. If an honest party shares a key with the adversary, the adversary can generate arbitrary messages that are correctly authenticated from the encryption scheme's point of view.

## 5.1 Requirements for authenticated key exchange

*Authenticated key exchange* (also known as *key agreement*) protocols establish a shared symmetric key while verifying who the other party is. They come in two main flavours: those that identify parties using public keys, and those that rely on knowledge of a shared password (we assume that only the genuine

parties know the password, and the adversary doesn't). In the password case, we usually assume that the password has less entropy than the $\geq 128$ bits that a symmetric key would need – if it had enough entropy, you could just use the password itself as symmetric key and be done with it. A Password-Authenticated Key Exchange (PAKE) is designed to establish a secure channel even if the password has much less entropy than that.

For example, a PAKE is used for WiFi passwords in the WPA3 standard. If a client device were to simply send the password to the access point (even encrypted), that would not be secure, because the adversary could simply operate a fake access point in order to learn the password, and there is no way for a client device to know whether an access point is fake or not. The PAKE allows the client device to prove to the access point that it knows the password, and for the access point to prove the same to the client device, without revealing the password to each other. That prevents the adversary from stealing the password.




Slide 125

The authentication model you're probably most familiar with is that of the web, using the TLS protocol. Here, the server authenticates itself to the client using a public key, which is part of a *server certificate*. TLS supports the client also authenticating itself to the server using a *client certificate* – this mode is known as *mutual TLS* or mTLS – but this is rarely used on the web (in some countries, citizens can authenticate to government websites using a client certificate stored on their ID card).

More commonly, a web client remains unauthenticated from a TLS point of view, and then at the application level the client authenticates itself to the server by sending the user's password over an encrypted and server-authenticated TLS connection. In this case, sending the password over the network (and not using a PAKE) is secure, because the adversary cannot impersonate the server – assuming that the client knows the correct public key for the server! (There is also the risk of phishing, where the user types their password into a lookalike website on a different domain name; that's a separate matter that we won't go into in this module.)

Passkeys/WebAuthn are a way for a web browser client to authenticate itself to a server using a public key. This authentication is not done at the TLS level, but rather on top of HTTP requests at the application level.

That brings us to the question of how each party actually knows the public key for the other party. For client authentication, a user may register their public key when they create an account. For server authentication, in some cases the client knows the server's public key in advance: for example, if you're building a mobile app that talks to a backend service that you operate yourself, then you control both ends of the communication. In this case, you can simply compile the server's public key into the mobile app. This practice is known as *public key pinning* or *certificate pinning*.

However, in general, you don't know in advance which parties a piece of software will want to communicate with. For example, a web browser can't know the public keys of all websites in the world. (In principle it could, but then it would be very slow to set up a new website, because you would have to wait for everybody to update to a new browser version that contains your website's public key before users could connect to your website.) Moreover, a server may need to rotate its key after it has become compromised, or as a precaution. To solve this, we need a PKI.

## Public Key Infrastructure (PKI)

But how does each party find out the other party's public key?

- **Web** (WebPKI):
  - Certificate is (domain name, pubkey, validity dates)
  - Certificate is signed by a **certificate authority** (CA)
  - Several CAs' public keys are built into the web browser
  - New website: CA checks via HTTP request or DNS whether you own the domain
- **Secure messaging** (Signal, WhatsApp, iMessage, ...)
  - **Key directory**: database of phone number → pubkey
  - Identity of key directory is built into messaging app
  - User registration: directory sends SMS to phone number

Can check whether CA/directory is honest using **certificate transparency** (and other transparency logs)

Slide 126

In practice, a TLS certificate often isn't signed directly by the CA's root key, but rather by some intermediate key. You then have a *certificate chain* in which the CA's root key signs the intermediate public key, and the intermediate public key signs the certificate for a particular domain name. It's even possible to have several intermediate certificate, but the basic idea and the trust model remain the same.

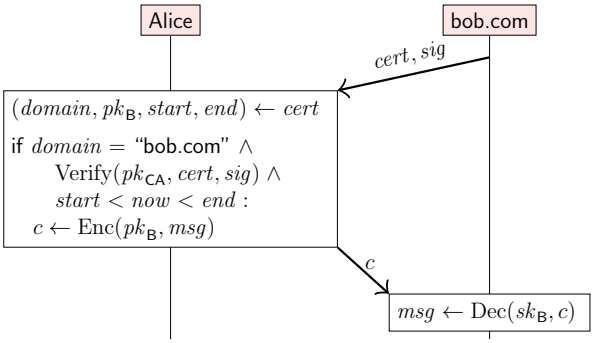
Both a certificate authority (CA) and a key directory are parties that we have to trust: if Bob manages to convince the CA that Alice's username (domain name, phone number) should be mapped to a public key that Bob controls, then Bob could intercept all of Alice's communications. It's hard to entirely avoid this trust relationship, and in practice it works quite well most of the time. To encourage CAs and key directories to be honest, one solution is to require all issued certificates and all key directory updates to be recorded in a public audit log. This log does not prevent a CA from doing something wrong, e.g. issuing a certificate to someone other than the true owner of a domain, but it ensures that such misbehaviour leaves clear evidence that can be used to sanction the CA by removing its certificate from the main browsers. This is what Certificate Transparency [Laurie, 2014] does; for key directories there are variants that hide phone numbers from the public log [Melara et al., 2015].

Slide 127 shows one way how a web browser could use a certificate to authenticate a server. When the client (web browser) first connects, the server sends the client the server's certificate, signed by a CA. The client checks that the certificate is for the correct domain name, that it's signed by a CA whose public key is built into the client, and that it's within the specified validity period. If so, the client encrypts its message to the public key included in the certificate.

## Simple server authentication

Let $cert = (\text{“bob.com”}, pk_{\mathsf{B}}, startDate, endDate)$

Let $sig = \mathrm{Sign}(sk_{\mathsf{CA}}, cert)$ and assume Alice knows $pk_{\mathsf{CA}}$




Slide 127

This mode of authenticated key exchange was supported in TLS 1.2 and earlier versions, but it was removed in TLS 1.3. The reason is that it doesn't offer *forward secrecy*: if the server's key were ever to be compromised, all past communications with that server could be decrypted.

## Including key compromise in the threat model

A simple scheme:

- Server sends its certificate to client, client checks it
- Client samples a random session key, encrypts it under server's public key (using a public key encryption scheme)
- Server decrypts session key
- Encrypt+authenticate communication using session key

If the server's private key is ever compromised, **all communication ever** with that server can be decrypted!

Adversary could record all ciphertexts now and hope to compromise key in the future ("store now, decrypt later")

We should try to handle key compromise as well as possible

Slide 128

If a private key is compromised, the adversary will inevitably be able to decrypt some messages. However, many modern protocols try to minimise the impact of the compromise to make it less catastrophic. One good property to aim for is *forward secrecy*.

## Forward secrecy

**Forward secrecy** (aka *perfect forward secrecy*):
If adversary learns private keys, they cannot decrypt any communication prior to compromise

Considered essential in many modern protocols
TLS since 1.3 always offers forward secrecy

- Use ephemeral keys (i.e. new keys for every connection)
- Keep generating new keys from old ones (ratchet)
- Diffie-Hellman with ephemeral keys is forward secure...
- ...if we can authenticate it correctly!

What about communication after compromise?

- Can still offer **post-compromise security** against passive eavesdropping
- Refresh keys from time to time, e.g. with new DH
- Can't prevent active impersonation by adversary

Slide 129

We can now list all the properties that we expect an authenticated key exchange protocol to have. We will focus on the case where two parties are both trying to mutually authenticate to each other, which is required e.g. for secure messaging (where the parties are users) or for mTLS (where one party is the client and the other party is the server). If you only need one party to authenticate to the other (like on the web, where the server is authenticated but the client not), it's easy to take a protocol with mutual authentication and just leave out the authentication on one side.

### Requirements for authenticated key exchange

For secure two-party communication, establish a **session key** for use with an authenticated symmetric encryption scheme with the following properties:

- **Confidentiality**: when two honest parties communicate, the adversary learns nothing about the session key
- **Authentication**: each party can verify the identity of the other party; adversary cannot impersonate
- **Consistency**: if $A$ thinks it's communicating with $B$, then $B$ thinks it's communicating with $A$
  - Violation is called **identity misbinding attack**
- **Forward secrecy**: if adversary compromises a party's private state, past session keys remain confidential

There are also **group key exchange** protocols for more than two parties (beyond scope of this module)

Slide 130

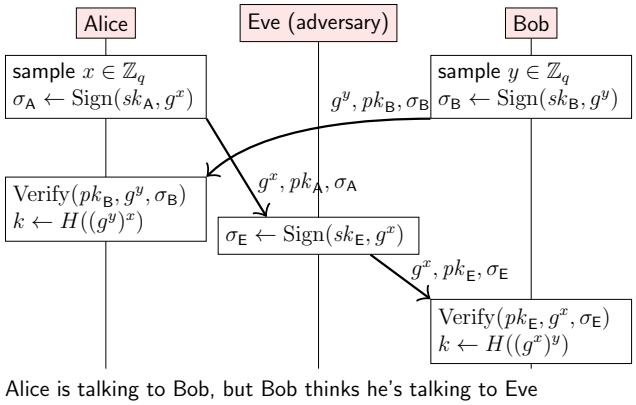
An identity misbinding attack is sometimes also called an *unknown key-share attack* [Blake-Wilson and Menezes, 1999].

## 5.2 Implementing a secure AKE protocol

It turns out that actually creating a secure authenticated key exchange protocol is not that straightforward: many plausible-looking protocols are in fact insecure. In this section we'll look at some examples from Krawczyk [2003]; that paper is recommended reading for your second assignment.

As a first attempt, let's assume that there is a PKI via which Alice and Bob can learn each other's correct public keys. Then each party can sign the Diffie-Hellman message it sends with its private key, and the other party can verify the signature using the public key.

### Badly Authenticated Diffie-Hellman (1)



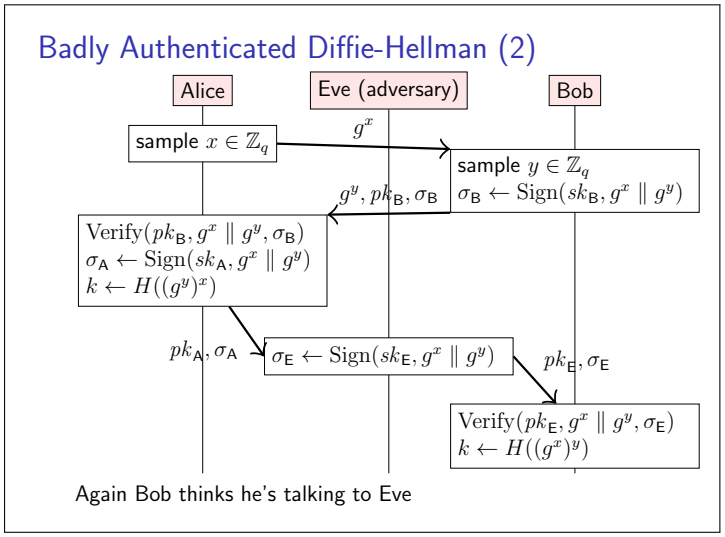

Slide 131

Unfortunately, this protocol is vulnerable to an identity misbinding attack as shown on Slide 131. Eve lets Bob's message to Alice pass through unmodified. From Alice's message Eve takes $g^x$ and then signs it with her own private key. Bob now believes that $g^x$ came from Eve, when in fact it came from Alice. Alice and Bob now share a key $k$, and Alice correctly believes she's talking to Bob, while Bob incorrectly believes he's sharing $k$ with Eve. Eve doesn't learn $k$, so this attack does not violate confidentiality, but it does violate the consistency property on Slide 130.

As an example of a situation where this attack could be a problem, imagine that Bob is a bank, and Alice and Eve are customers of the bank. After establishing an authenticated session with the bank, Alice sends the bank some information that has financial value (e.g. an instruction to deposit a cheque), asking the bank to credit it to her account. However, Eve has interfered with the communication as in Slide 131, and so the bank believes that the information is coming from Eve. As a result, the bank credits Eve's account instead of Alice's. Even though Eve never learnt the session key, she has broken the security of

the protocol.

Another problem with this protocol is that Eve could record the messages in one protocol run and replay them in another run of the protocol, because Alice's message doesn't depend on Bob's message and vice versa. For example, if Eve ever manages to learn one of Bob's private exponents $y$, she could replay the corresponding $(g^y, pk_\mathsf{B}, \sigma_\mathsf{B})$ message (which bears Bob's valid signature) to impersonate Bob in future runs of the protocol. To prevent such replay attacks, we need to ensure that a signature from one run of the protocol has no value in another protocol run. The protocol in Slide 132 does this by computing the signature over both Diffie-Hellman exponentials.

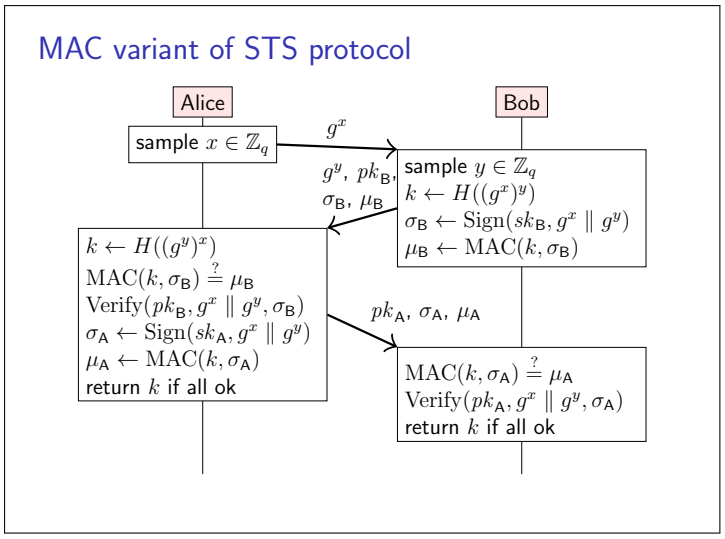



Slide 132

This solves the replay attack, but not the identity misbinding attack. Eve can still replace the last message from Alice to Bob with one containing her own signature, and thereby make Bob believe that he has established a session with Eve.

To prevent an adversary who doesn't know the shared key from tampering with the messages, we could try using a MAC in addition to the signature. This allows one party to prove to another party that it knows some symmetric key, without revealing it.

The *Station-to-Station* (STS) protocol [Diffie et al., 1992] improves on the protocol on Slide 132 by sending not only the signature over the two Diffie-Hellman exponentials, but also a MAC over that signature computed using the session key produced by the Diffie-Hellman exchange.[1]




Slide 133

[1] Actually, the STS protocol as originally published uses symmetric encryption instead of a MAC; Krawczyk [2003] highlights that this doesn't make sense, since we don't need the signature to be confidential – we need to ensure that it can't be replaced by a party who doesn't know the session key. That property is provided by a MAC or by authenticated encryption, which includes a MAC.

The STS protocol is better than the earlier ones, but it still has weaknesses, as shown on Slide 134 [Blake-Wilson and Menezes, 1999].

The first weakness is that the same $k$ is used for the MAC and returned as the session key; by itself this does not break the scheme, but it goes against the principle that a key should be used for only one purpose. Ideally we would like that the session key $k$ is indistinguishable from random to the adversary, but by sending a MAC under $k$ over the network the protocol allows the adversary to test whether a key guess $k$ is correct. Fortunately, this weakness is easy to fix, as we will see shortly.

The second weakness is that the protocol only includes the participants' public keys, but not their human-readable identities; it relies entirely on the PKI to supply the mapping between human-readable names and public keys. If a PKI allows a user to register someone else's public key (without checking that this user actually controls the corresponding private key), it's possible to have an identity misbinding attack where Bob has authenticated Alice's public key, but believes that this public key actually belongs to Eve. Normally, a PKI is most focussed on ensuring that it doesn't associate an honest user's name with an adversary's public key; this issue is the other way round, where an adversary's name is associated with an honest user's public key. The PKI can prevent this attack by checking that whoever registers a public key can generate signatures that validate with that key. However, it would be nice if the key exchange protocol was secure even without making this assumption about the PKI.

The third weakness is that, depending on which signature scheme is used, the adversary might be able to generate a new keypair so that Alice's genuine signature also validates under the adversary-generated keypair. Even if a signature scheme offers existential unforgeability, this might be possible! This is known as a *key substitution attack* [Jackson et al., 2019].

### Weaknesses in the STS protocol

1. The same $k$ is used as MAC and as session key
   - Violates single-purpose principle
2. Identifies the public key of the other side, but not the human-readable name
   - Could Eve take Alice's public key $pk_{\mathsf{A}}$ and register ("Eve", $pk_{\mathsf{A}}$) with the PKI?
   - If so, we have an identity misbinding attack again
   - To prevent, PKI must check whether Eve knows $sk_{\mathsf{A}}$
3. Adversary might be able to create a new keypair $(sk_{\mathsf{new}}, pk_{\mathsf{new}})$ such that signature $\sigma_{\mathsf{A}} = \mathrm{Sign}(sk_{\mathsf{A}}, m)$ validates with $pk_{\mathsf{new}}$, i.e. $\mathrm{Verify}(pk_{\mathsf{new}}, m, \sigma_{\mathsf{A}}) = \mathsf{true}$
   - Then Eve registers $pk_{\mathsf{new}}$ with PKI and replaces $pk_{\mathsf{A}}$ with $pk_{\mathsf{new}}$ in last message
   - Depends on signature scheme
   - Existential unforgeability says you can't make a new valid signature for a given key; it doesn't say you can't make a new key that validates an existing signature!

Slide 134

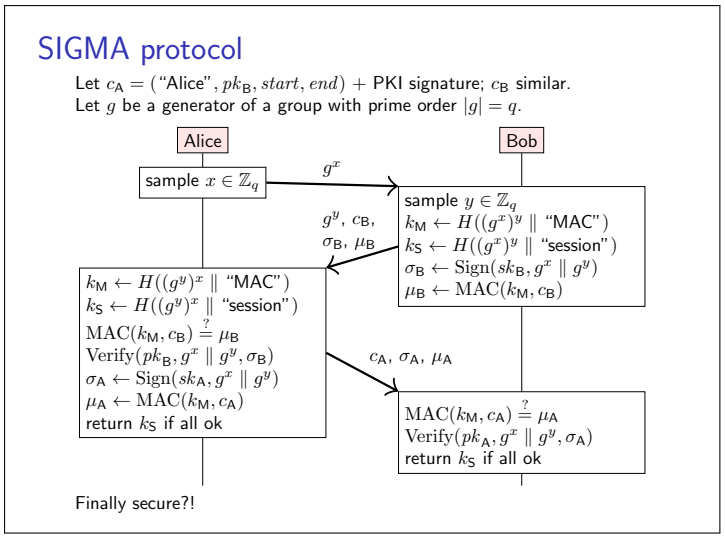
Finally, the SIGMA protocol [Krawczyk, 2003] fixes these remaining weaknesses. It's not obvious that there are no remaining bugs, but that's what security proofs [Canetti and Krawczyk, 2002] are for!

The differences to the STS protocol on Slide 133 are:

1. Two different keys, a MAC key $k_{\mathsf{M}}$ and a session key $k_{\mathsf{S}}$, are derived from $g^{xy}$ by concatenating it with different constant strings (called *domain separation tags*) before hashing. This produces two keys that are *computationally independent*: that is, no information about $k_{\mathsf{M}}$ can be learnt from $k_{\mathsf{S}}$ and vice versa, thanks to the preimage resistance of the hash function.

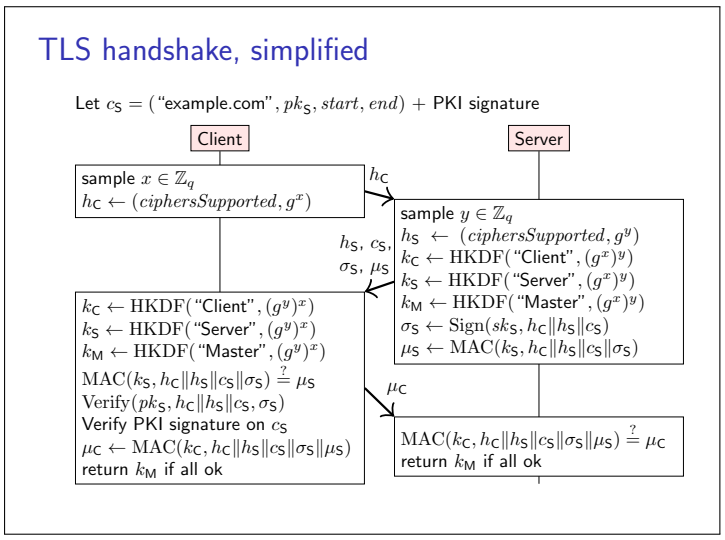
2. The parties send each other their certificates containing their human-readable names and their public keys, rather than just their public keys.

3. Instead of computing the MAC over the signature $\sigma_{\mathsf{A}}$ or $\sigma_{\mathsf{B}}$, the MAC is computed over the certificate of the party sending the MAC. This binds the Diffie-Hellman exponentials to the human-readable names and the public key of the parties, preventing key substitution attacks.




Slide 135

The SIGMA protocol is the basis of the handshake in TLS 1.3 [Rescorla, 2018], which is outlined on Slide 136. Some key differences are that it omits the authentication of the client to the server (although this is supported as mTLS), that it uses a HMAC-based key derivation function HKDF instead of a plain hash function, and that the signature and MAC cover more fields than the minimum required by SIGMA (in fact, they cover the entire *transcript* of data sent so far during the handshake). This does no harm and reduces the risk of bugs: it's better to authenticate too many fields than too few! The client and server also tell each other which algorithms they support, so that they can choose a ciphersuite that both ends understand. There are lots of additional details not shown here; for example, the signatures are actually computed over strings that contain additional context, to prevent a signature generated in one context from being replayed in another context.




Slide 136

And now you understand the core idea behind most secure communications over the Internet. There are lots more variations on the theme of authenticated key agreement, some of which aim to achieve additional security properties. For example, mTLS computes signatures over the handshake transcript, which produces a cryptographic proof that the two parties communicated; in contrast, the X3DH key exchange used by the Signal Protocol [Marlinspike and Perrin, 2016] and the Off-the-Record (OTR) protocol [Borisov et al., 2004] aim to provide *deniability*, which means that even though the communicating parties are authenticated to each other, they cannot prove cryptographically to any other party that the communication took place. (Whether the absence of such cryptographic proof would sway a legal case is so far unclear.)

## More on authenticated key exchange

Now just use the session key with an authenticated symmetric encryption scheme, and that's the core that makes most secure communications (TLS, secure messaging, VPNs, ...) work!

Lots of extensions to that core:

- Identity protection: encrypt public keys + identifiers so that eavesdropper can't see who is communicating
  - e.g. Great Firewall of China blocks websites based on hostname in TLS handshake
- Ratchet: for long-running sessions, periodically refresh keys (for forward secrecy + post-compromise security)
- From two-party to group communication
  - including adding and removing group members
- Managing trust in the PKI (transparency logs, ...)
- Some protocols (e.g. OTR, Signal's X3DH) offer deniability (no cryptographic proof of communication)

Slide 137

If you want to see the full gory details of how TLS works, you can look at RFC 8446 [Rescorla, 2018], and there is also a byte-for-byte breakdown of a TLS handshake online [Driscoll, 2018]. You should now understand all of the cryptographic building blocks that make it work! However, it's a very complex protocol – partly to maintain compatibility with older versions of TLS, partly because of extra features (such as the 0-RTT mode) that allow performance improvements, and partly because it supports a range of different cryptographic algorithms (providing *cryptographic agility*, i.e. allowing algorithm to be swapped out without changing the protocol, in case an algorithm turns out to be broken).

If we had more time in this module, we'd ask you to write your own basic TLS implementation that is able to establish a connection with a real server on the Internet. In principle, given your implementations of X25519 and Ed25519 (plus a hash function and a symmetric cipher from a library), you could now do this. However, given the complexity of the protocol, we think that is not feasible. So your first task for the second assignment is a simplified version that demonstrates the core idea by implementing the SIGMA protocol from Krawczyk [2003], as described on Slide 135. We suggest that you implement it using your own X25519 and Ed25519 implementations from the first assignment, but if you prefer you can also use an off-the-shelf library for these algorithms instead (e.g. if you want to use a different language than in your first assignment). For HMAC and symmetric encryption you can use library implementations in any case.

## Assignment 2, Task 1

**Implement the SIGMA protocol** using X25519, Ed25519, and HMAC.

- There's no RFC, so you need to define the format of the messages yourself (and justify it in your lab report)
- Use it to build a basic **two-party secure messaging protocol** (use a library for hashes and symmetric crypto)
- As PKI, implement a basic CA that issues certificates (signed using Ed25519), and include certificate validation in your protocol implementation
  - X.509 certificates are complicated; make your own simple format
  - Omit check whether user controls the phone number/email address/domain name
- Identity protection, ratcheting, etc. are not required
- Simulate the network in a single process

Slide 138

Unlike the previous assignment, for this task there isn't a specification that defines the exact byte-for-byte structure of your data, so part of the task is for you to define your own formats for encoding messages as bytes that could be sent over a network. For ease of testing, the actual network should be simulated in your code – nothing complicated: you can simply have a function that returns a byte string to be sent, and pass that string to another function that handles the message on the recipient side. Please also include a basic certificate authority in your implementation; the format in which you encode

certificates is also something for you to define. (Real TLS certificates use the X.509 certificate format defined in RFC 5280 [Boeyen et al., 2008], but this comes with a lot of baggage such as needing to parse the ASN.1 file format, which is rather complicated – something that's important to get right in a real TLS implementation, but mostly irrelevant to the actual cryptography.)

A real CA would need to check that the user registering a name (domain name, phone number, email address, etc.) actually controls that name, e.g. by sending it a test message containing a nonce. You can ignore this in your implementation, since it's a separate concern from the cryptography.

## 5.3 Password-authenticated key exchange

Let's look into an alternative form of authenticated key exchange, which we briefly saw on Slide 125.

### Password-Authenticated Key Exchange (PAKE)

Authenticated Key Exchange requires knowing the other party's public key. In most cases that means trusting a PKI (CA or key directory) for global name → pubkey mapping.

Can we manage **without a PKI**?

PAKE: no global names, no pubkeys, just a shared password

- e.g. wifi router: password is printed on the box
- e.g. device pairing: one device displays a code, type it into the other (or scan a QR code)
- e.g. link shared via secure messaging/video call/email

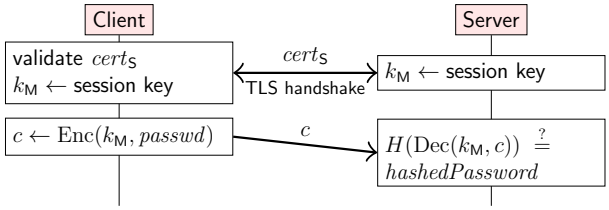
Password should be short enough that you can sensibly type it (i.e. much less than 128 bits entropy), and PAKE should upgrade this to a strong shared secret

Slide 139

A PAKE is different from password authentication as it's normally used on the web. On the web, the client authenticates the server via its TLS certificate (relying on the WebPKI), and then sends its password over that encrypted and server-authenticated TLS connection. If we want to avoid the dependency on a PKI, we have to use a different approach. Simply sending an encrypted password would not work if we don't know the identity of the other party with which we're sharing a key: the other party might be the adversary, which would then learn the password.

### Passwords on the web: Not a PAKE

Let $cert_S = (\text{"example.com"}, pk_S, start, end)$ + PKI signature



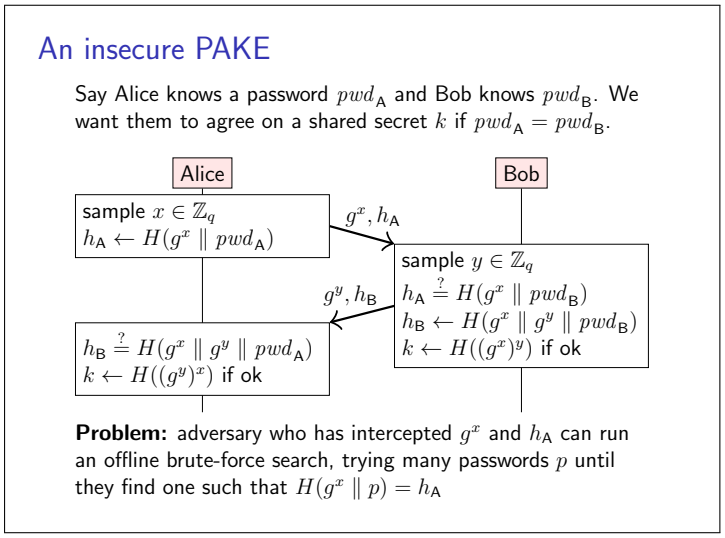

Doesn't work without a PKI: client wouldn't know who it's sending the password to

Slide 140

Using a PAKE instead of a PKI-based system can have a number of advantages. It can be simpler: for example, with a wifi router you can just configure a password and then give that password to anybody who should have access; you don't have to register the router with any CA. It can be more resilient: with a PKI you have to trust that the company operating it keeps it secure (and doesn't shut it down), but a PAKE is more decentralised because it doesn't rely on any external infrastructure. And often it better

reflects the trust relationships that exist between people in the real world [McKelvey et al., 2021].

Then how can one party prove to the other party that it knows the password, but without revealing it? One thing we might try is to send a hash of the password instead, and to bind it to a Diffie-Hellman exchange by including the public Diffie-Hellman values in the hash, as shown on Slide 141.

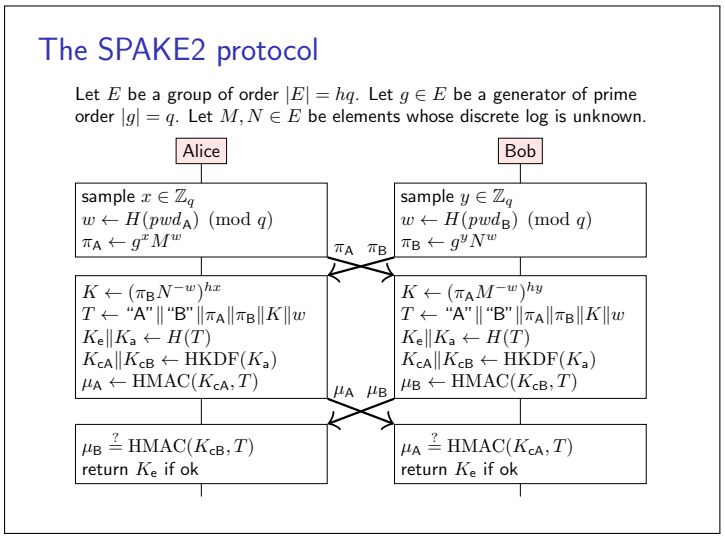



Slide 141

Unfortunately, that protocol is insecure: an adversary can eavesdrop on one message, and then run an offline brute-force search to recover the password. Since we are assuming that passwords are low-entropy, such a brute-force search is likely to be feasible. Even assuming that the hash function is preimage resistant (as on Slide 14) doesn't help if an adversary can just try all the likely passwords. With a typical hash function such as SHA-256, password recovery tools like Hashcat can test tens of billions of potential passwords per second on a single GPU. Even if you use an intentionally slow password hashing function such as Argon2 or scrypt, the adversary is only limited by the computing resources they are willing to pay for. Any password that is short enough to be realistically typed by users is probably weak enough to be broken by a motivated adversary.

We therefore need a better protocol, in which the messages that an adversary might intercept incorporate so much entropy that a brute-force search is infeasible (as discussed on Slide 25, this is usually at least 128 bits). There are a number of PAKE protocols that meet this requirement [Hao and van Oorschot, 2022], and we will look at one example protocol called SPAKE2. We chose it because it is fairly simple and can be implemented using the cryptographic building blocks that you have seen in this module.

SPAKE2 was originally published by Abdalla and Pointcheval [2005], and is further described in RFC 9382 [Ladd, 2023]. Note that they use different notation: Abdalla and Pointcheval [2005] use multiplicative group notation, whereas the RFC uses additive notation, as discussed on Slide 41. We use multiplicative notation on Slide 142 for consistency with the earlier protocols in this lecture.




Slide 142

Multiplying $g^x$ with $M^w$ is essentially a way of encrypting $g^x$ using the password (this is also called *blinding*). If the recipient knows the same password, they can decrypt (unblind) the message by multiplying with the inverse element of $M^w$ to recover $g^x$, and then we have a regular Diffie-Hellman once again. If the two parties have different passwords and hence different values of $w$, they will compute different values for $K$, and thus all the derived keys will also be different and the final MAC check will fail.

The additional factor $h$ in the exponent is the cofactor of the group, which is relevant in elliptic curve groups such as Curve25519 and Edwards25519, which are not prime-order but have $h = 8$. Multiplying by the cofactor has the same function as the clearing of the three least significant bits that you saw as part of the clamping process in X25519 and Ed25519: it protects against small subgroup confinement attacks.

The orginal SPAKE2 protocol description just consists of a single round that outputs the hash of the transcript $H(T)$, which includes $K$. However, at this point the parties don't yet know whether they actually have the same password. RFC 9382 adds to this a HKDF-based key derivation process and a confirmation step, in which a session key $K_e$ and two different confirmation keys $K_{cA}$ and $K_{cB}$ are derived from $T$. Alice uses $K_{cA}$ as a MAC key to prove to Bob that she knows the password, and using $K_{cB}$ Bob proves the same to Alice.

### SPAKE2: Why it works

- $\pi_A = g^x M^w \implies (\pi_A M^{-w})^{hy} = (g^x M^w M^{-w})^{hy} = g^{hxy}$
- $\pi_A$ and $\pi_B$ are uniform random group elements $\implies$ leak no information about password
  - $g^x M^{H(pwd)}$ essentially encrypts $g^x$ with the password
  - Intentionally unauthenticated! Authentication would enable offline brute-force
- The MAC $\mu_A$ allows the party that generated $\pi_B$ to verify whether the passwords were equal (similarly with $\mu_B$ and $\pi_A$), but not an eavesdropper
- In any run of the protocol, adversary can talk to Alice, pretend to be Bob, and guess some password $pwd_B$.
  - If it succeeds (MAC is correct), guess was correct
  - If it fails, adversary only learns that password was wrong
  - $\implies$ adversary gets one password guess per protocol run
  - $\implies$ limit protocol retries based on password entropy

Slide 143

With any PAKE, an adversary who is actively manipulating the communication can make one guess at the password per protocol run that they interfere with; that is inevitable. The honest parties can't tell the difference between a protocol run where the passwords didn't match and a run where there was active interference from an adversary – both result in the final MAC check failing. If the protocol fails, the parties can retry it; however, they shouldn't retry too often, since every retry gives the adversary one guess at the password. The lower the entropy in the password, the stricter this rate limit needs to be.

One thing you might be wondering about are the special group elements $M$ and $N$ in the protocol. Their importance is explained on Slide 144 [Warner, 2016].

### SPAKE2: Trusted setup

If adversary knows discrete log $n$ such that $N = g^n$:

- Adversary pretends to be Bob, sets $\pi_B = g^y$, receives $\pi_A = g^x M^w$ and $\mu_A$ from Alice
- Alice computes $K = (\pi_B N^{-w})^{hx} = (g^y g^{-nw})^{hx} = (g^x)^{(y-nw)h} = (\pi_A M^{-w})^{(y-nw)h}$
- Adversary knows $\pi_A$, $M$, $y$, $n$, $h$; just not $w$
- Adversary can now guess $w = H(pwd) \pmod q$, compute $K$, hence compute $K_{cA}$ and the MAC
- If the MAC matches $\mu_A$ from Alice, $pwd$ guess is correct
- Adversary can now run offline brute-force search

Attack is prevented if nobody knows $n$.

Requires **trusted setup**: whoever generates $M$ and $N$ must convince others that their discrete log is unknown.

Slide 144

RFC 9382 contains $M$ and $N$ values for various curves, as well as the algorithm that was used to generate them. The fact that the algorithm is open and reproducible, and that it doesn't seem to be doing anything strange, gives us confidence that the values were generated in a trustworthy way. This is another example of the *nothing-up-my-sleeve* principle, like on Slide 67. When values are generated in such a transparent way, we don't need to trust whoever generated them (unlike a PKI, which has to be trusted all the time that a system is in operation).

You're now ready to implement SPAKE2 yourself! RFC 9382 is less rigorous than the RFCs you saw for X25519 and Ed25519: it doesn't fully specify the byte-for-byte encoding, so it's not detailed enough to ensure that different implementations are interoperable [Warner, 2017]. But it does specify more detail than Abdalla and Pointcheval [2005]'s paper, such as the key derivation functions to use.[2]

### Assignment 2, Task 2

**Implement the SPAKE2 protocol** using Edwards25519.

- Same curve as for Ed25519 – you can use your implementation from assignment 1 (but don't have to)
- RFC 9382 contains $M$ and $N$ values you can use (given using compressed point encoding)
- The RFC contains some ambiguities; you'll need to decide on some details yourself
- Deadline: 3 Mar 2026

Slide 145

# 6 Software Engineering II

## 6.1 Randomness

Randomness is a core ingredient in cryptography. It is used for generating cryptographic keys, nonces, tokens, salts, and many other important values. An adversary who can predict these values can break the security of many cryptographic primitives. For example, knowing the prime factors of an RSA modulus allows for easy factoring and thus breaking the encryption. Hence, good cryptography engineering requires careful handling of randomness.

[2] RFC 9382 first hashes the transcript and then feeds half of the transcript hash into HKDF to derive the confirmation keys. We don't know why that construction was chosen – it would have been simpler to pass the transcript directly into HKDF and then obtain the session key and both confirmation keys from HKDF.

## Randomness

- Cryptography requires random inputs
  - Key generation
  - Nonce generation
  - Tokens
  - ...
- Building strong random number generators is not trivial

Slide 146

## **LAB:** build your own PRNG (10 min)

Hidden from the published slides.

Slide 147

## **LAB:** collect (5 min)

Hidden from the published slides.

Slide 148

## Real-world entropy

Instead of relying on deterministic algorithms, we can use real-world entropy sources.

- Interrupts (e.g. from user input)
- Hardware sources (e.g. electrical noise, nuclear decay, . . . )
- Observation of physical phenomena
- Lava lamps
- . . .

Slide 149

## Lava lamps


Image from Wikipedia (CC-BY 2.0 Dean Hochman)

Slide 150

## Real-world entropy

Instead of relying on deterministic algorithms, we can use real-world entropy sources.

- Interrupts (e.g. from user input)
- Hardware sources (e.g. electrical noise, nuclear decay, . . . )
- Observation of physical phenomena
- Lava lamps
- . . .

Problem:

- Not uniformly distributed
- Slow entropy sources
- Might be temporarily unavailable

Slide 151

## Cryptographically secure PRNGs (CSPRNGs)

PRNGs are not suitable for cryptographic applications. We need a CSPRNG with strong guarantees:

- **Next-bit test**: given all previous $i$ output bits, predicting the next bit is computationally infeasible
- **Forward security:** if an adversary learns the state of the PRNG, they cannot use it to predict previous outputs
- **Recovery:** after compromise of the state, new entropy can be added

Slide 152

Instead of relying on real-world entropy sources directly, we can use the "real randomness" to seed a cryptographically secure PRNG. This combines the benefits of real-world entropy sources with the performance and convenience of a deterministic source. A normal PRNG is not suitable for cryptographic applications as its output leaks information about its previous state. Therefore, we require that a CSPRNG fulfils the properties introduced on Slide 152. The next-bit test is the most important one as it ensures that the PRNG is unpredictable. It can be shown that a CSPRNG that passes the next-bit test produces an output that is computationally indistinguishable from random. Forward security and the recovery property provide interesting parallels to the properties that we have sought for messaging protocols (Slide 129.)

## Measuring entropy

Entropy is the amount of uncertainty in a source, i.e. "how surprising is the next event?".

The entropy $H$ of a discrete random variable $X$ with possible values $\{x_1, \ldots, x_n\}$ and probability mass function $P(X)$ is defined as:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i)$$

**Example:** A fair coin has an entropy of 1 bit per toss (we assume it will not land on its edge). A sequence of four fair coin tosses has an entropy of 4 bits.

Slide 153

## **LAB:** compare (10 min)

Hidden from the published slides.

Slide 154

## Randomness tests

The NIST Statistical Test Suite is a collection of tests that can be used to evaluate the quality of a random number generator.

- Frequency test
- Block frequency test
- Runs test
- Compression tests
- ...

Slide 155

The NIST Statistical Test Suite [NIST, 2010] is a collection of tests that can be used to evaluate the quality of a random number generator. However, they are only heuristic tests and therefore not foolproof. One should particular attention if the output of the RNG is used for different distributions, e.g. during machine learning [Dahiya et al., 2024].

## The Linux CSPRNG design

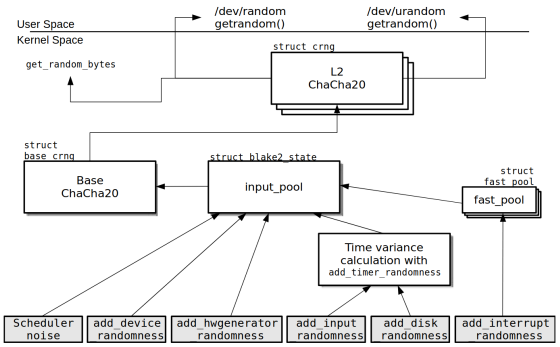



Image from [BSI, 2022].

Slide 156

This Linux CSPRNG design relies on a pool of entropy collected from interrupts and hardware sources. Randomness from these sources is extracted and added to the pool where it is mixed with the existing pool state using the BLAKE2 hash function. Whenever new information is added, its entropy is estimated and a global counter is incremented accordingly.

The input pool seeds a Base ChaCha20 stream which in turn generates random bytes for seeding Level 2 CSPRNGs. A typical system has a instance of the Level 2 CSPRNG for each core so that applications can speedily and without contention access random numbers. During the lifetime of the system, the input pool is continuously reseeded with new entropy coming from the system's entropy sources. And so are its dependent CSPRNGs.

User programs can use the `getrandom()` system call and the files `/dev/urandom` and `/dev/random` to access the Linux CSPRNG. The only difference is that `/dev/random` blocks if the entropy pool is empty, e.g. early during boot. One can check the available entropy from `/proc/sys/kernel/random/entropy_avail`. By design, the total entropy in the pool grows continuously and will from then on always report its maximum value, which is typically 256 bits. To overcome the potential lack of entropy early during boot, some systems store a seed on disk and use it for bootstrapping.

## CSPRNGs in practice

**Linux:**

- Read from `/dev/urandom` or `/dev/random`[1]
- `getrandom()` system call

**Python:**

- Preferred method: `secrets` module
- Still common: `os.urandom(...)`

[1] The only difference is that `/dev/random` blocks if the entropy pool is empty, e.g. early during boot. You can read the available entropy from `/proc/sys/kernel/random/entropy_avail`.

Slide 157

## Case study: Debian OpenSSL Predictable PRNG (CVE-2008-0166)

- OpenSSL collects entropy from many different sources (`/dev/urandom`, time, ...)
- Read method tried to be clever and includes also uninitialized parts of a buffer

```
int RAND_load_file(const char *file, long bytes) {
    /* ... */
    i=fread(buf, 1, n, in);
    if (i <= 0) break;
    /* even if n != i, use the full array */
    RAND_add(buf, n, double(i));
    /* ... */
}
```

Slide 158

## Case study: Debian OpenSSL Predictable PRNG (CVE-2008-0166)

- Down-stream developers (Debian) saw Valgrind warnings
- Remove two lines that adds these uninitialized buffers
- The only remaining source of entropy was the PID...
- Keyspace $|K| = 32\,768$

**Take-aways:**

- Low entropy can be more dangerous than no entropy at all
- Avoid user-level CSPRNGs. Use the kernel-level CSPRNG instead.

Slide 159

## Dealing with imperfect randomness

Often we deal with randomness that is not perfect, e.g. biased and not uniformly distributed. We can capture these conditions with the following definition:

A probability distribution $\mathcal{X}$ has **min-entropy** (at least) $m$ if for all $a$ in the support of $\mathcal{X}$ and for random variable $X$ drawn according to $\mathcal{X}$:

$$\mathrm{Prob}(X = a) \leq 2^{-m}$$

We cannot use these sources directly as input for our cryptographic primitives that require uniform randomness. This includes pseudo-random functions (PRFs) that are the basis of many constructions.

Slide 160

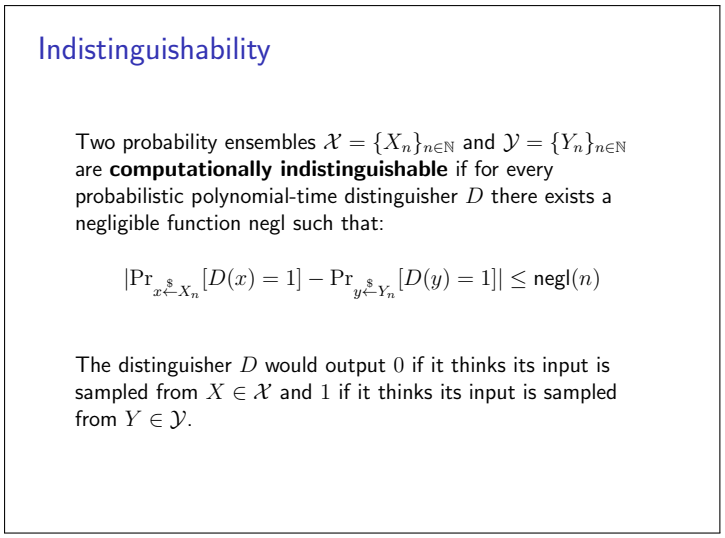
## Key derivation functions (KDFs)

We can use a **KDF** to generate cryptographically strong keys from a source with min-entropy $m$:

- The initial extraction step also relies on a secret salt
- HKDF [Krawczyk, 2010] is a popular KDF with an HMAC-based extract-and-expand construction

As such we can expand a short random seed into a larger number of pseudorandom bytes. In addition, extra *info* parameters enable domain separation for keys.

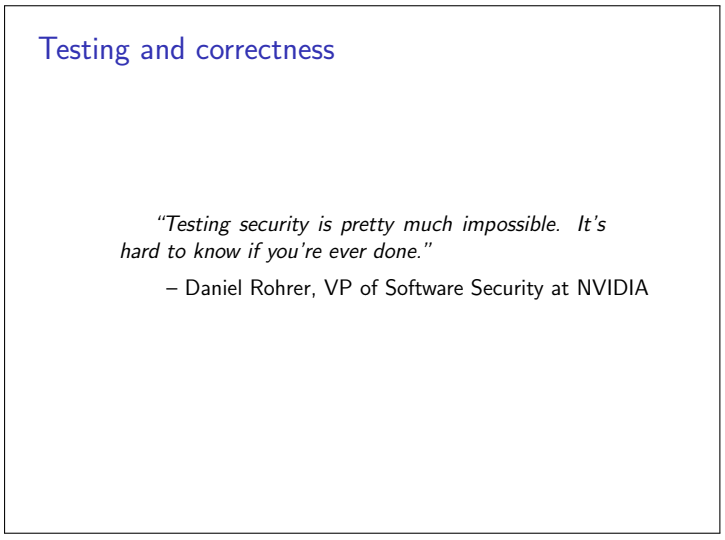
Slide 161




Slide 162

The above definition is adapted from Definition 7.30 in [Katz and Lindell, 2020]. Using probability ensembles, i.e. infinite sequences of probability distributions, are required so that we capture the asymptotic behavior. Where samples $x_n$ from the ensembles are shorter than $n$, we'd also want to technically pass in the unary $1^n$ input to the distinguisher to allow it to run in polynomial time. From this definition it follows that also having polynomial many samples of each distribution is sufficient for indistinguishability, which then directly applies to practical PRNGs.

## 6.2 Testing




Slide 163

This quote from a 2025 blog post reflects an important challenge with cryptography and protocol implementations. Our normal, heuristic approaches to convince ourselves that software is fit for purpose are not sufficient: it only takes a single input that triggers a broken behavior to potentially result in devastating consequences. Therefore, implementing cryptography software is special and benefits from a close connection between formal methods and robust engineering.

During our journey we saw that even the seemingly simple task of precisely defining what we mean with "correct and secure implementation" is quite difficult. For instance, as we saw on previous slides, we need to be precise about the exact error and edge case behaviors. Also, where side-channels are of concern, considering our implementation only on the source code abstraction is no longer sufficient but dependent on the underlying hardware platform. The same is true about the compiler model. For instance, the same source code can result in different binary code depending on a compiler version—a snapshot that compiled side-channel-free today, might do something different tomorrow with a new compiler version. This blog post shows how some compiler versions and configurations can introduce branches in the final assembly even where the source code has been carefully written using a constant-time construction.

## Approaching correctness

- It is really really hard to convince ourselves that our code is always, always correct and secure
- In fact, it is already hard to precisely define what this means
  - Edge cases and error handling
  - Side-channels
  - Abstraction: source code, binary file, execution, ...
  - Assumptions about the compiler, hardware model, ...

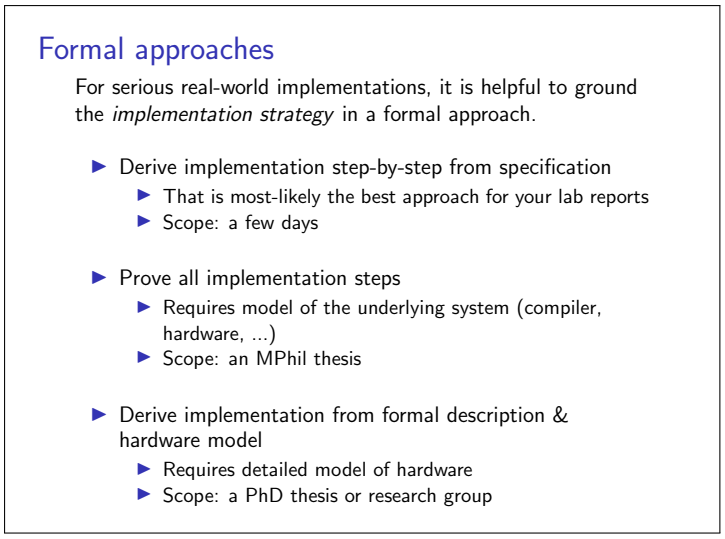
Slide 164

## Testing strategies

- Bottom-up: testing individual operation thoroughly across their specification range
- Top-down: economic testing of overall functionality and compatibility
- Edge cases and error handling
- Randomized tests
  - Against another implementation
  - Fuzzing (later slides)

Slide 165

These testing strategies complement each other, each addressing different aspects of correctness and security. While bottom-up testing helps ensure individual components work correctly, top-down testing verifies that these components work together as intended. Edge cases and error handling are particularly important for cryptographic implementations since they often provide attack vectors, as we saw when discussing padding oracle attacks in slide 91. Randomized testing through fuzzing is a powerful technique for finding edge cases and implementation bugs that we might not think of ourselves, which we'll explore further in slide 172.

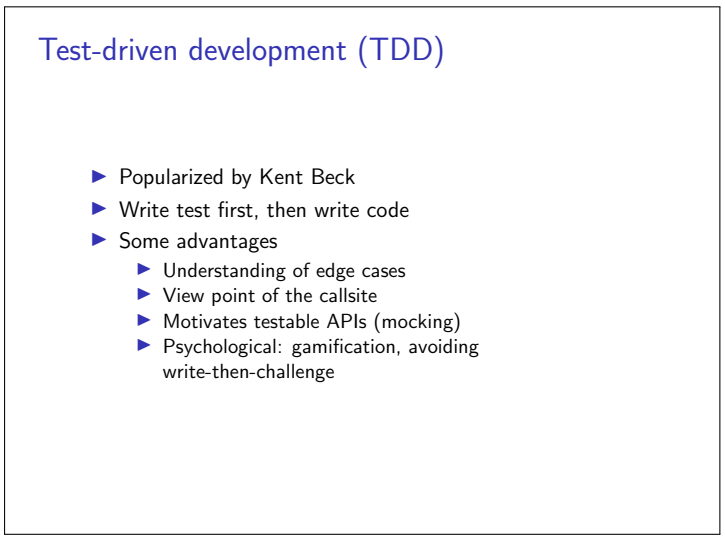



Slide 166

Formal approaches are a great way to *be sure* that an implementation is correct. In this slide we highlight three representative approaches ranging from simple to complex. The first one should feel familiar to you and is likely how you have been approaching many other challenges, e.g. implementing algorithms and data structures. However, because the correctness of cryptographic protocols is critical, this can only be the minimum requirement.

Many hardened real-world implementation use tools such as Isabelle [Paulson, 1994] to "prove" the correctness of the underlying model and resulting implementation. However, these proofs (and the properties they are proving), are only approximations of reality—limited by the fidelity of the model. One example for a hand-written implementation accompanied with a machine-checked formal proof is Amazon's s2n-bignum library.

Another approach is to first fully describe the specification/algorithm in a formal language. A precise model of the target architecture and build chain this is then used to derive a provable correct implementation. One example for this approach is "Fiat Cryptography" [Erbsen et al., 2020] which is used in production software, e.g. Google's Chrome browser.




Slide 167

Test-driven development (TDD) [Beck, 2022] has undergone a variety of interpretations over the last years and can mean different things to different people. Note that, as with any other approach, doing TDD for TDDs-sake is not helpful and as every tool it has its set of problems where it is very effective, and might not be a great choice in other scenarios.

## Test vectors

- The bread and butter of testing cryptographic implementations
- As a very basic minimum requirement, each library should pass the test vectors provided in the specification
- Good tests:
  - check intermediate values
  - generate additional test vectors for edge cases
  - strategically build coverage
- Project Wycheproof collects "tricky" test vectors

Slide 168

## Advanced test vector patterns

Test vectors are also a great way to test compatibility of different implementations.

Example: server written in Go and an Android app in Kotlin

- Server adds new test vectors as part of CI
- Android app tested in CI against vectors
- **ensures spatial compatibility**

Can be extended with persisted vectors

- continuously add test vectors from committed versions
- test new versions against existing vectors
- **ensures temporal/backwards compatibility**

Slide 169

## Case study: Wycheproof finds bugs in `elliptic` (2024)

The JavaScript `elliptic` library is downloaded over 10 million times weekly.

- Trail of Bits ran Wycheproof test vectors against the library
- **CVE-2024-48949** (EdDSA signature malleability):
  - Missing check that $s < n$ (FIPS 186-5, section 7.8.2)
  - Allows forging new valid signatures from existing ones
- **CVE-2024-48948** (ECDSA verification failure):
  - Leading zeros stripped from hash before truncation
  - Valid signatures rejected with probability $2^{-32}$

Slide 170

The EdDSA vulnerability is a classic example of signature malleability: given a valid signature $(R, s)$ for a message, an attacker can compute $s' = s + k \cdot n$ for any integer $k$ such that $s' \bmod n = s$. Since the library did not verify that $s < n$, the modified signature $(R, s')$ would also pass verification. This clearly violates the unforgeability property of digital signatures. In practice, the impact of this vulnerability

depends on the application context. It might allow an attacker to create multiple valid signatures for the same message, which could be exploited in certain scenarios, e.g. to bypass replay protections or to create confusion about the authenticity of messages. Alternatively, an attacker might be able to use the divergent behaviour from the standard to cause confusion in a distributed system where some implementations accept the malleated signature while others reject it.

The ECDSA bug was due to a more subtle issue: the hexadecimal representation of the hash was parsed into a big number object which stripped leading zeros. Subsequently, another method computes the wrong bit shift, causing valid signatures to fail verification. The probability $2^{-32}$ is small enough that it is unlikely to be encountered even during extensive, randomisd testing, but large enough that it is easily feasible to calculate a message with a hash value that triggers the bug.

This case study [Schiffermuller, 2025] illustrates highlights lessons for cryptographic software engineering. First, even widely-used and mature libraries can harbour subtle bugs that remain undetected for years. Second, comprehensive test vectors – particularly those designed to probe edge cases like leading zeros and boundary values – catch issues that conventional unit tests miss.

## Fuzzing

Automated testing of programs using randomized input.

Initial population:

- Test vectors
- State from previous runs
- Samples from production code

Classic fuzzing loop:

- Add new candidates (bit flips, dictionaries, . . . )
- Measure "coverage" (basic blocks, edges, . . . )
- Trim population

Slide 171

Fuzzing has become a widely used and well-supported technique for building confidence in software that processes (potentially malicious) input. Popular target software include protocol implementations as well as libraries that parse file and media formats. Traditionally, fuzzing has focused on identifying classic memory bugs and undefined behaviors that might occur in C programs. For this the programs are compiled and executed with special guard rails that turn these bugs into crashes that are then visible to the fuzzer. For example, address sanitizers features (ASAN) will flag reads from uninitialized memory, use-after-free bugs, and buffer overflows. However, this can be easily extended to also cover logic bugs and deviating behavior from other reference implementations.

## Fuzzing

- Traditionally focused on C-style bugs
  - but can be applied to logic bugs itself given reference
- Particularly valuable for anything that parses adversary-controlled input
  - Deserialization code
  - File formats
  - Network protocols
  - ...
- Continuous fuzzing as part of CI/CD strategy (e.g. OSS Fuzz)

Slide 172

While fuzzing can be performed as a one-off endeavour, e.g. after implementing new features, many projects use 24/7 fuzzing services, e.g. Google's OSS Fuzz. This allows for longer runs and therefore more comprehensive coverage. In addition, it can serve as an opportunity to benchmark different fuzzing techniques.

## Fuzzing road-blocks

Some code is easier to fuzz than other. Tricky conditions that are unlikely to pass with random guessing are often referred to as "road blocks".

```
def handle_packet(pkt: Packet):
    tag = pkt[32:40]
    if hash(pkt[:32]) == tag:
        parse_packet(pkt[:32])
```

Countermeasures:

- Allow fuzzer to disable/skip checks
- Symbolic execution, back-propagation
- Additional entry points

Slide 173

One challenge in fuzzing are so-called *road blocks*. These are conditions in code that are unlikely to be passed by random guessing. For example, a protocol implementation might first verify a message's authentication tag before parsing the inner message. Hence, simply mutating the input string most likely will cause this early check to fail and not explore interesting code paths afterwards.

There are a few approaches for working around these. For simpler checks, e.g. length constraints, back-propagation of required properties and symbolic execution can provide an automatic solution. However, with protocol software it is more likely that we need to build a separate target of our library where some checks are optionally deactivated or simplified. Alternatively, we might expose new entry points of internal methods, e.g. the `parse_message` function in the above target, and run our fuzzer against these more narrow targets. However, this can lead to false positives for which no real-world exploit chain exists. Nevertheless, it will be worth to fix these inner issues as well given smart prioritisations among other findings.

## Performance optimizations

- Write a **correct** (naïve) implementation first. Celebrate.
- Make all tests pass
- Add benchmarks and ensure they are solid
- Do small step-by-step improvements
  - The tests give you confidence
  - Benchmarks help you to evaluate changes as you go

Slide 174

In your lab report make sure you discuss "costs". This can include maintenance burden, additional dependencies, reduction in compatibility with other libraries, more complex API interfaces, ... It is tempting to over-optimize on quantifiable metrics such as runtime and not defending more qualitative metrics.

## Benchmarking with `timeit`

```
import timeit
x = setup(...)
ts = timeit.repeat(
    'your_function(x)',
    globals=globals(),
    repeat=5, number=5
)
```

- Minimize the lines under test
- Consider mean, media, $p_{90}$, $p_{99}$, ...
- Reduce noise (GC, dynamic frequency, ...)
- "Warm-up" the code

Slide 175

When measuring the performance of individual operations, the built-in `timeit` module is a good starting point. In general, we want to reduce the measured section as much as possible and leave non-critical operations, e.g. parsing serialized data and logging[3], out of the measurement. However, it is important to understand that the performance of a single operation can vary widely depending on the context in which it is executed. Therefore, we want to reduce external noise (e.g. other running processes, variable CPU frequency, ...) as much as possible. In cryptography, many algorithms are randomised and therefore the performance can vary even between two runs of the same program. One stark example is the generation of RSA keys which typically relies on the Robin-Miller primality test which is probabilistic.

This blog post by Filippo Valsorda discusses how to benchmark such operations efficiently without having to run too many samples (RSA key generation is a notoriously slow operation). The idea is to find representative samples, commit them to the repository, and then test against these when comparing different code snapshots. This approach ensures that our benchmarks have sufficient coverage, but importantly, it keeps inter-run variance to a minimum which in turn allows running faster benchmarks with fewer samples.

Once we have a solid performance baseline, we can start to optimise. For this we are interested in understanding which parts of our code are slow. This is where profiling comes in. In Python, the `cProfile` module provides us with detailed stacktraces of which functions are called and how much time is spent in each function. All with relatively low overhead so that it does not distort our results. It's convention to use its Context Manager interface in a `with` statement. We stop recording by calling `pr.disable()`.

In the slide below we use `cProfile` to profile the performance of an X25519 implementation. Not surprisingly, the `__mul__` operation is the most time-consuming one and therefore an ideal target for optimisation. Instead of a text based output, we can call `pr.dump_stats` to save the profile to a file which can be analysed later. This is particularly useful when we want to compare the performance of different runs before and after our "optimisation". A common visual representation of the profile is a flame graph which can be generated using tools like `snakeviz`. In a flame graph, the functions are represented as boxes and the time spent in each function is represented by the width of the box. Importantly, the x-axis is the cumulative time spent in a function and its call stack—not the actual linear time. That is, multiple calls to the same function in a loop are represented as a single box.

[3] In particular, `print` statements notoriously turn CPU-bound code into IO-bound tasks and therefore invalidate our conclusions.

## Profiling with cProfile

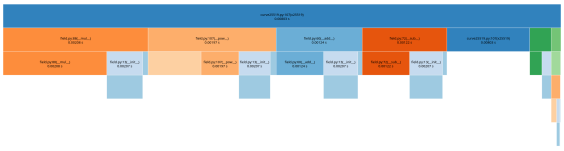
```
import cProfile, pstats
with cProfile.Profile() as pr:
    x25519(alice_sk, bob_pk)
    pr.disable()
    pstats.Stats(pr) \
        .strip_dirs() \
        .sort_stats('cumulative') \
        .print_stats(5)
```

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001    0.007    0.007 curve25519.py:102(x25519)
  1276    0.001    0.000    0.002    0.000 field.py:77(__mul__)
  1021    0.001    0.000    0.002    0.000 field.py:96(__pow__)
  4596    0.001    0.000    0.002    0.000 field.py:8(__init__)
  1020    0.001    0.000    0.001    0.000 field.py:49(__add__)
```

Slide 176

## Profiling with cProfile (Flamegraph)

```
import cProfile, pstats
with cProfile.Profile() as pr:
    x25519(alice_sk, bob_pk)
    pr.disable()
    pr.dump_stats('x25519.prof')
```



Slide 177

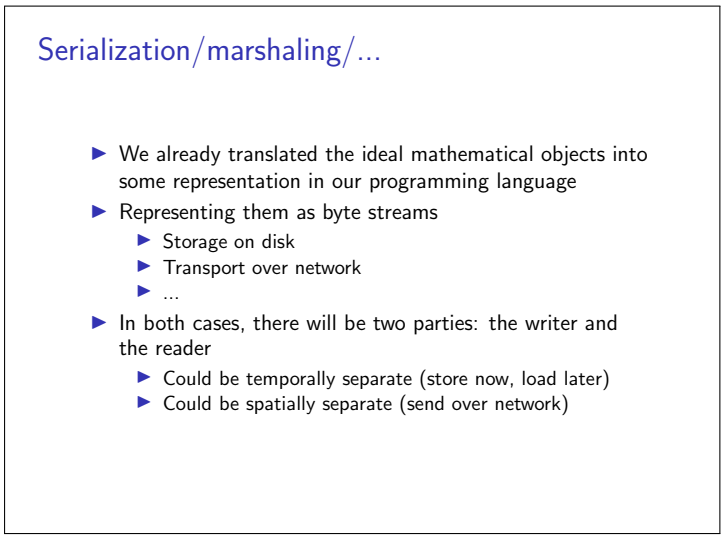
Slide 178 shows the same profile. The top half of the graph is the same as before, however, the bottom half is an inverse flamegraph. It aggregates not by the caller but by the callee. This allows us to quickly identify the functions where we spend most execution time overall and where we call them.

## Profiling with cProfile (Flamegraph)

Slide 178

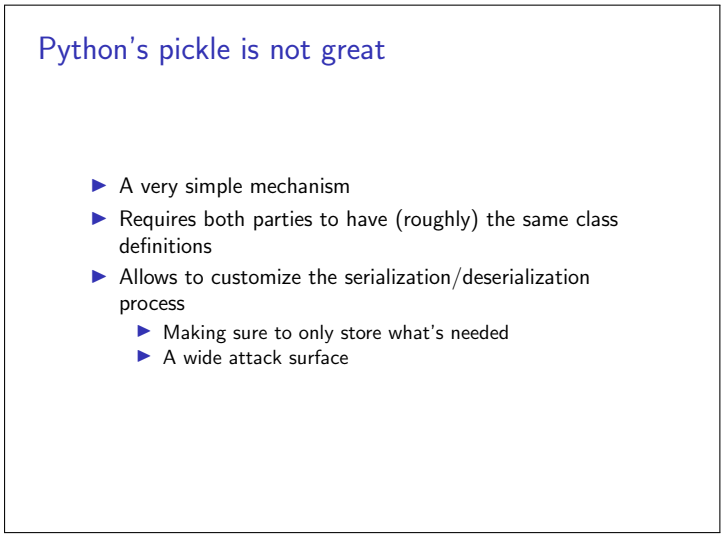
## 6.3 Serialization and marshaling

In the previous Section 6.2 we have explored the challenge of transforming our ideal mathematical objects into representations within the constraints of the programming language of our choosing. However, often we need to do an additional step – we need to turn them into byte streams so that we can exchange them over the network or persist them in files. Often it is tempting to model these after the representations in our implementation, i.e. the language serves as an intermediate step. However, this can complicate interoperability where these mechanisms are tied to a particular language.




Slide 179

The terms of serialization and marshaling are often used synonymously to mean "turning an in-memory object (graph) into a byte stream". However, in some contexts, e.g. Java, marshaling also means including the code of the object itself. Including arbitrary code in marshaling can make receiving potentially untrusted files very risky and we discuss this using Python's `pickle` framework as an example.




Slide 180

## Pickle example (1)

```
import pickle

class Foo:
    def __init__(self, name):
        self.name = name

foo = Foo('good')
with open('foo.pkl', 'wb') as f:
    pickle.dump(foo, f)
```

Slide 181

## Pickle example (2)

```
import pickle

class Foo: pass

with open('foo.pkl', 'rb') as f:
    foo = pickle.load(f)
```

Slide 182

## Pickle example (3)

```
import pickle

class EvilFoo:
    def __reduce__(self):
        return (
            exec,
            ('import os; os.system("uname -r")',)
        )

with open('evil.pkl', 'wb') as f:
    pickle.dump(EvilFoo(), f)
```

Slide 183

## Pickle example (4)

```
$ xxd -c8 evil.pkl
00000000: 8004 9538 0000 0000  ...8....
00000008: 0000 008c 0862 7569  .....bui
00000010: 6c74 696e 7394 8c04  ltins...
00000018: 6578 6563 9493 948c  exec....
00000020: 1c69 6d70 6f72 7420  .import
00000028: 6f73 3b20 6f73 2e73  os; os.s
00000030: 7973 7465 6d28 2268  ystem("h
00000038: 746f 7022 2994 8594  top")...
00000040: 5294 2e              R..
```

Slide 184

## Pickle example (5)

```python
import pickle

class Foo: pass

with open('evil.pkl', 'rb') as f:
    foo = pickle.load(f) # Boom
```

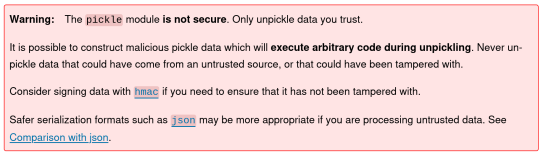
Slide 185

The first examples demonstrate why Python's pickle functionality enjoys great popularity: it is very easy to use and easily stores an object in any byte stream. Note that the deserialization does not rely on the constructor, but uses the `__getstate__` and `__setstate__` methods. These can be overwritten for customized behavior, e.g. not including secret data or large cached data that can be reconstructed. The ability to customize is where the danger lies.

We can also overwrite the `__reduce__` method to dictate how an object is being reconstructed by pickle. It returns both a function (reference) to call and a tuple of arguments. These instructions are actually included in the pickle byte stream – and hence can come as a surprise on the deserializing side as the included (arbitrary) code will be executed directly by the interpreter.

Importantly, the hexdump reveals that the `evil.pkl` file does not even contain a reference to the `EvilFoo` class. Unpickling this file will relatively directly execute the same code that we had written in the `__reduce__` method. If the `os` were in scope the initial `import` would not be needed.

## Do not use Python's pickle!



**Warning:** The `pickle` module **is not secure**. Only unpickle data you trust.

It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling**. Never unpickle data that could have come from an untrusted source, or that could have been tampered with.

Consider signing data with `hmac` if you need to ensure that it has not been tampered with.

Safer serialization formats such as `json` may be more appropriate if you are processing untrusted data. See Comparison with json.

- If you have to, make sure that nobody can tamper with the files (using a MAC, ...)
- The AI/ML community is rediscovering this
  - `model = torch.load(PATH, weights_only=False)`
  - Still an on-going issue

Slide 186

Interestingly, using pickle for loading and storing models is very wide-spread in the AI/ML community and is widely used to load third-party models from the Internet. The PyTorch tutorial on saving and loading models does not mention the potential security risks at all[4].

## Case study: Log4J

- Log4Shell (CVE-2021-44228) is considered one of the most serious vulnerabilities of the last years
- Messages could contain special tags `${type:arg}` to enhance the message
- Log4J allowed plugin-like behaviour with `${jndi:url}`
- Faults
  - Trusting untrusted user-controlled input
  - Message parsing leads to code loading and execution
  - JRE allows downloading code during runtime by default

Slide 187

[4] However, in some instances, e.g. when using `torch.nn.Model`, one can use a dictionary based format to store weights without much risk. The related GitHub discussion started in 2021, has not come to a satisfying conclusion. The main challenge is that many actually rely on pickle to build complex objects that are tedious to deserialize in new environments. Alternatives likes ONNX can be used instead.

## Better choices: length-encoding, protobuf, JSON

- Protobuf:
  - Describe types and messages
  - Compiler generates language bindings
- JSON
  - Simple and human readable
  - Type differences between languages (date format, ...)
- Custom formats (e.g. length encoded)

Slide 188

In the next slides we explore how the popular Serde library for Rust provides a framework for serialization and deserialization. Its macros generate the required code at compile time without relying on reflection APIs as many frameworks do in languages like, e.g. Java and Python. This provides both performance benefits and increased confidence about what code can be reached when loading from byte streams. Interestingly, Serde separates between the serialization approach (via the macros) and the encoding format. The latter is left to individual plugins and hence allows supporting many different formats.

In the following example, we use the `serde_json` plugin to serialize and deserialize to JSON and the `rmp_serde` plugin for MessagePack. MessagePack is a binary encoding format that is equivalent to JSON in terms of expressiveness, but more compact and faster to parse.

## Serde example (1)

```
use serde::{Deserialize, Serialize};

#[derive(Serialize, Deserialize)]
enum Message {
    Ok,
    Text{msg: String},
}
```

Slide 189

## Serde example (2)

```
serde_json::to_string(&msg)?;
// json: {"Text":{"msg":"Hello world!"}}
```

Slide 190

## Serde example (3)

```
rmp_serde::to_vec(&msg)?;
// msgp: 81A45465787491AC
//       48656C6C6F20776F
//       726C6421
```

Slide 191

The separation of the serialization approach and the data format allows us to change the mapping between language types and their abstract representation easily. In our example, we use an `enum` type and see that it uses a key-value pair to remember the used variant. However, we can change the behavior to make either format to use a separate `tag` key or to leave it `untagged` and simply use the first matching variant. The latter is very useful when parsing typical REST-API responses that can return different JSON variants.

## Serde example (4)

```
#[derive(Serialize, Deserialize)]
#[serde(tag = "type")]
enum Message {
    Ok,
    Text{msg: String},
}

// json: {"type":"Text","msg":"Hello world!"}
// msgp: 92A454657874AC48
//       656C6C6F20776F72
//       6C6421
```

Slide 192

## Serde example (5)

```rust
#[derive(Serialize, Deserialize)]
#[serde(untagged)]
enum Message {
    Ok,
    Text{msg: String},
}

// json: {"msg":"Hello world!"}
// msgp: 91AC48656C6C6F20
//       776F726C6421
```

Slide 193

## What we want from serialization tools

- Simple (complexity is danger)
- Expressive (avoids writing custom sub-parsers)
- Cross-platform and cross-language type agreement
- Backwards compatibility (new code)
- Streamable parsing
- Pluggable

Slide 194

## Postel's law

> *"2.10 Robustness principle: [...] be conservative in what you do, be liberal in what you accept from others."*
>
> — *RFC 761, Transmission Control Protocol*

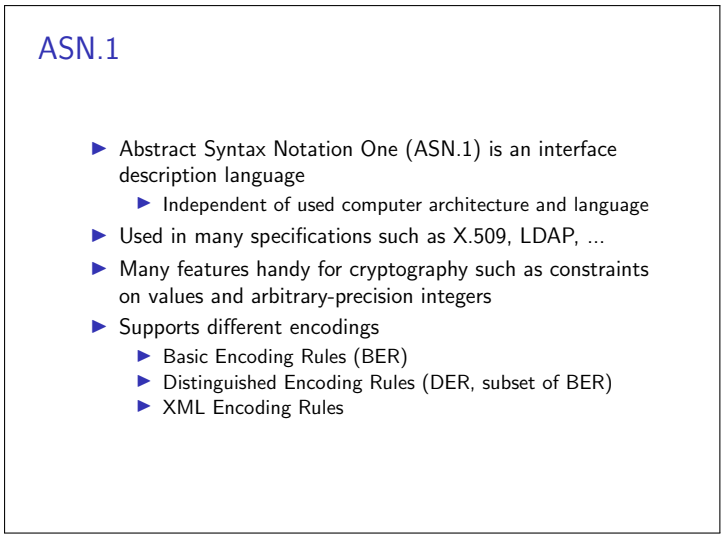
- Great idea for making systems work together
- Often bad for cryptographic systems
- Example: YAML v1.1
  - `name: peter` is a string
  - `name: yes` is a boolean
  - Requires very careful parsing and generating

Slide 195

In the general software engineering mindset, we are interested in robust protocols and data exchange. This works well where we are trying to make many different implementations talk to each other and not crash. However, being liberal with what we accept and (try to) parse, can lead to an increased attack surface. For instance, researchers found that Tor's protocol flexibility introduced new side-channels Ro-

chet and Pereira [2018]. And even seemingly simple configuration formats like YAML can have surprising behaviour – and in security we do not like surprises. In the following example, a list of three countries in ISO-3166-2 format will get parsed as the types string, bool, string. This is because earlier YAML versions allowed many words for boolean values, including: true, false, on, off, yes, and . . . no.


Slide 196




Slide 197

Many protocols and cryptographic specifications express their objects in ASN.1 [ITU-T, 2002a]. It's independent of the used computer architecture and programming language. Therefore, it is well suited for the ground truth description. For example, it does not refer to bytes (a platform specific word) but instead uses octets, groups of 8 bits.

The most common encoding for protocols is DER. DER is a subset of BER, but with extra constraints such that each object has exactly one encoded representation (compare the YAML example above) [ITU-T, 2002b]. Therefore, DER is very helpful when we have to compute signatures over other objects since there is no ambiguity of the message that is to be signed. Compare this to, e.g. JSON, which can be formatted in many different ways.

In practice some protocols sidestep this issue by computing a signature over the transmitted serialized form, instead of first parsing the bytes into an in-memory representation and then serializing it again. This works well for when we can retrieve the fully-encoded object in the incoming message. However, it does not work if we actually have to create such an object on-the-fly, e.g by combining information from multiple sources.

## ASN.1 example (1)

```
MyModule DEFINITIONS ::= BEGIN
    Message ::= CHOICE {
        ok OKMessage,
        text TextMessage
    }

    OKMessage ::= NULL

    TextMessage ::= SEQUENCE {
        message UTF8String (SIZE(0..1024))
    }
END
```

Slide 198

The above ASN.1 specification picks up our message example from Slide 189. One interesting feature of ASN.1 in the context of cryptographic protocols is its ability to define constraints, such as the length of the message. The following Python program generates three different outputs from our specification and our test message.

## ASN.1 example (2)

```
import asn1tools

for codec in ('der', 'xer', 'jer'):
    mod = asn1tools.compile_files('module.asn', codec)
    msg = mod.encode(
        'Message',
        ('text', {'message': 'Hello'})
    )

    with open(f"message.{codec}", 'wb') as f:
        f.write(msg)
```

Slide 199

## ASN.1 example (3)

```
$ xxd -c9 message.der
00000000: 3007 0c05 4865 6c6c 6f  0...Hello

<Message>
    <text><message>Hello</message></text>
</Message>

{"text": {"message": "Hello"}}
```

Slide 200

## Popular serialization formats

| | |
|---|---|
| JSON | ▶ Human readable and flexible<br>▶ Few built-in types (no support for, e.g. dates) |
| MessagePack | ▶ "Binary JSON encoding" (concise and fast)<br>▶ Compact encoding and less ambigious encoding |
| protobuf | ▶ Efficient binary format by Google<br>▶ Schema required (compiled) |
| ASN.1 | ▶ Complex standard for telecom/crypto<br>▶ Multiple encodings (BER, DER, etc.)<br>▶ Platform independent |
| CBOR | ▶ Concise and fast<br>▶ Extensible without schema |

Slide 201

## Other deserialization challenges

- Protection against resource exhaustion and Denial of Service (DoS) attacks
- Defense against ZIP bombs (e.g. compressed long, repetitive strings or recursive archives)
  - Limit expansion and deny nested compression
  - Less of a problem with streaming processing
- Handling recursive encodings and references
  - Avoid formats that allow clever references
  - Detect cycles and/or limit stack size

Slide 202

## 6.4 Randomness II

We revisit the topic of randomness from Section 146 and discuss some topics more in-depth. In particular, we discuss the challenges of deriving keys from passphrases and providing encryption algorithms with IV/nonce parameters reliably.

## PSA: Universally Unique Identifier (UUID)

`d37b6a75-0419-11f0-9ba1-f875a40a2c42`

- Created to uniquely identify objects (128 bits) and designed to avoid conflicts.
- Different versions exist:
  - V1: 48-bit MAC address + 60-bit timestamp
  - V4: 6-bit version + 122-bit random number
  - V7: 6-bit version + 48-bit timestamp + 74-bit counter/random
- Do not assume that a UUID is a valid cryptographic secret! For instance, it is not uniformly random and subsequent UUIDs can be predicted.

Slide 203

In many cryptographic protocols we need to derive keys from passphrases. While we have seen passphrase authentication with a remote party using SPAKE2 in Section 142, we now consider local key derivation, e.g. for full disk encryption.

## Password-based key derivation functions

- We cannot use the passphrase directly as a key, as it is not uniformly random. However, we can fix that using a KDF.
- Problem: the min-entropy is low and hence passwords are vulnerable to brute-force attacks
- Solutions:
  - Generate high-entropy passphrases for the user
  - Making the password derivation step intentionally expensive

Slide 204

One problem with passphrases is that those chosen by users often have low min-entropy, as they pick common words or short phrases Blocki et al. [2018]. Hence, one solution is to generate a passphrase for the user from a word list. Today, there are two popular word lists: the EFF word list and the BIP 39 word list which is used for some cryptocurrency applications.

## Generating high-entropy passphrases (EFF)

- EFF word list: $|L| = 6^5 = 7776$ words
  - Also called "dice list"
  - Chosen to avoid pairs of words that are similar (e.g. "build" and "built")
- Single word provides: $\log_2(6^5) = 12.92$ bits
- For 128-bit security we need 10 words:

"snowfield enamel subtext awkward viscous yippee hardly clamshell deploy anew"

Slide 205

## Generating high-entropy passphrases (BIP 39)

- BIP 39 word list: $|L| = 2048$ words
  - Each word uniquely identified by its first 4 letters
- Single word provides: $\log_2(2048) = 11$ bits
- For 128-bit security we need 12 words:

"wild artefact gossip float pelican novel toddler salute dish agent actor figure"

"wild arte goss floa peli nove todd salu dish agen acto figu"

Slide 206

Such long passphrases are not very practical for users and thus only feasible for some special cases. However, if we can make the guessing time for a passphrase high enough, we can get away with shorter passphrases. Password-based key derivation functions (PBKDFs) were designed for this purpose. They use iterative application of a hash function, to force a sequential number of operations, which makes them slow to compute and hard to parallelize.

The original PBKDF1 simply applied the hash function multiple times. However, this runs at risk that the evaluation sequences for different passwords converge once a common intermediate value is reached. For instance, if $H(H(\texttt{hello})) = \texttt{0x1234}$ and $H(H(H(\texttt{world}))) = \texttt{0x1234}$, then all following hashes in both cases will be identical—this can give an advantage to the adversary. Therefore, PBKDF2 typically uses a HMAC construction keyed with the password to avoid such shared evaluation chains.

However, simply applying a hash function iteratively is often not good enough in practice. Already a few ten-thousand iterations often hit a few milliseconds on a modern CPU and thus we cannot choose much higher levels for interactive applications. At the same time, an adversary with a GPU or ASIC can evaluate billions of hash operations per second. For instance, this hashcat benchmark suggests that an NVidia H100 GPU can compute up to $100 \cdot 10^9$ SHA-256 hash operations per second. Another indication is the hash rate of the Bitcoin network which is estimated to be north of 100m TH/s (100 million tera-hash operations per second) or $100 \cdot 10^{18}$ hash operations per second.

## Let us make things slower

- Hash function $H(pw)$: 10,000s of millions per second
- PBKDF1: $H(H(...H(pw)))$
- PBKDF2: combine intermediate results using XOR and allow for variable output length
  - Single guess cannot be parallelized
  - However, multiple guesses can be parallelized using GPUs / ASICs
- Adversary scales with computation speed

Slide 207

## Adversary with compute power (cloud GPU)

Assume:

- GPU computes 100 GHash/s (NVidia H100)
- Adversary rents 1,000 GPUs

In one day the adversary can make:

- $\approx$ 8.6 quintillion guesses ($\approx$ 63 bits)

Slide 208

## **LAB:** Design a memory hard function

**Challenge:** design a function that is memory hard! It should take a password and have the following requirements:

- Easy to compute if you have 100 MiB space
- Hard to compute if you have much less than that

5 min to think about it and then we'll gather and discuss solutions

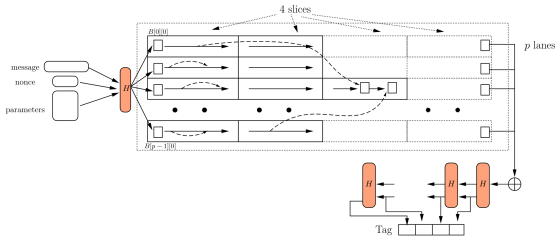
Slide 209

The Argon2 KDF [Biryukov et al., 2021] is a popular choice for memory-hard (password-based) key derivation and was selected as the winner of the 2015 Password Hashing Competition. Argon2 takes a memory parameter which specifies the amount of memory the algorithm will use. Its memory-hard property guarantees that the output is only efficiently to compute with the specified amount of memory. It does so by filling the memory with a pseudo-random function and then indexes into the memory and updating it multiple times based on the previous steps. Thus, an implementation without the specified amount of memory would have to then re-compute the requested memory blocks again.

Argon2 also has a time parameter which specifies the number of iterations and a parallelism parameter which specifies how many independent memory blocks are used and thus how many cores can be used. It comes in three variants: Argon2d, Argon2i, and Argon2id. Argon2d used data-dependent memory access, i.e. the indexing operation is also based on the password. This variant is believed to be more secure, but more likely leads to side-channel attacks. Argon2i uses data-independent memory access, i.e. the indexing operation is not based on the password. This mitigates side-channel attacks, but is potentially weaker. Argon2id is a hybrid of the two and uses data-independent memory access for the first half of the memory and data-dependent memory access for the second half. RFC 9106 recommends Argon2id as the default variant.

## Argon2 (RFC 9106)

- Parameters: memory, number of iterations, parallelization
- 2+1 version: Argon2d, Argon2i, Argon2id
- Optimized for multi-core and assembly




Graphic from https://www.password-hashing.net/argon2-specs.pdf

Slide 210

## Why optimize an intentionally slow step?

- Hardness relies on difficulty of the underlying problem. Not our implementation!
- Making it faster allows us to choose harder parameter within our acceptable bounds

→ We work against an adversary which has the fastest possible computation

Slide 211

## Cambridge HPC

Assume:

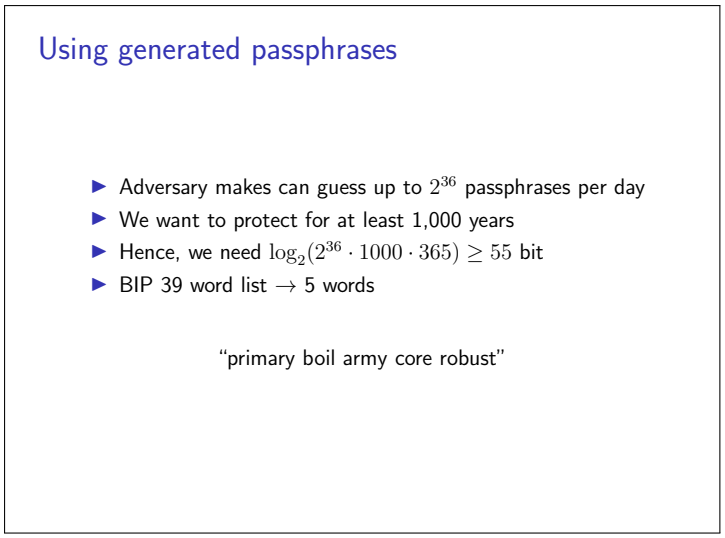
- Cambridge HPC: ∼500 PiB memory
- 10ms per guess, limited only by memory
- Set Argon2 memory parameter: 256 MiB

In one day the adversary can make:

- ≈ 104 billion guesses (≈ 36 bits)

Slide 212

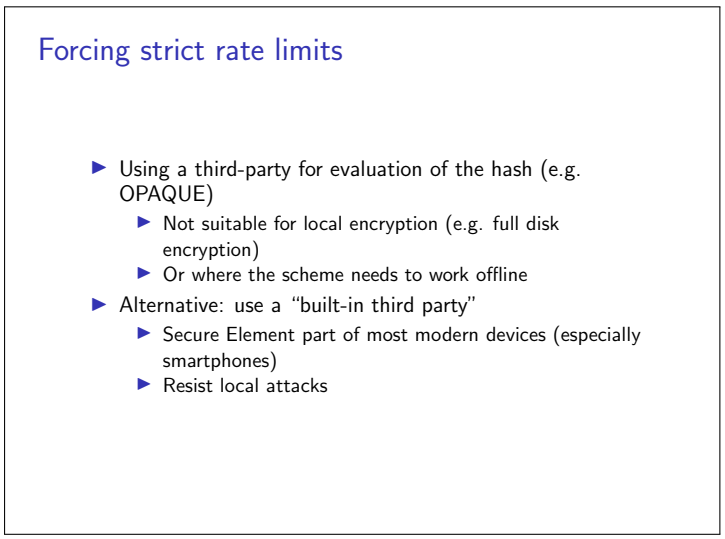
Having to build a high-performance computer as in Slide 212 is likely a lot more expensive than acquiring a few hundred ASICs as in Slide 208. Thus, the memory-hardness of Argon2 is a good property to have! With more confidence in how long it takes to verify a guess, we can then use shorter passphrases:




Slide 213

An alternative to purely local key derivation is to use a third-party for evaluation of the hash. For instance, in the OPAQUE protocol [Jarecki et al., 2018] the client needs the server to evaluate the hash of the password. The protocol uses an oblivious pseudorandom function (OPRF) such that the server does not learn anything about the password. This allows applications to enforce rate-limits on the server side. However, this setting is not suitable for local encryption (e.g. full disk encryption) or where the scheme needs to work offline. It is also not suitable for applications that need plausible deniability, i.e. we do not want leave traces of using it at all.

Some of these short-comings can be mitigated by using a "built-in third party" which is a secure element (SE) embedded in the device. The SE is a specialized, tamper-resistant processor that is designed to perform cryptographic operations securely. This allows smartphones to enforce rate-limits on otherwise insecurely short PIN codes. However, typically only the first-party operating system can utilize these features of the SE. By using the its limited computational bandwidth for generally-available cryptographic operations as an intentional bottleneck, we can enforce actual wall-time constraints for password guesses—without requiring modifications to the device or operating system.




Slide 214

In addition to PBKDFs, we also want to quickly discuss modern hash functions such as SHA-3 and BLAKE2. One important improvement of the newer generations of hash functions is that they are resistant to length extension attacks. This is a property that the SHA-1 and SHA-2 hash functions do not have—hence the importance of the HMAC construction (see Slide 18).

## Modern hash functions (SHA-3)

- Part of Keccak, published by NIST, 2015
- Based on a sponge construction
  - Different to Merkle–Damgård in SHA-1 and SHA-2
- Large (hidden) internal state prevents length extension

Slide 215

## Modern hash functions (BLAKE2)

- Faster than SHA-3, based on ChaCha
- Prevents length-extension attacks by compressing the last block differently
- Argon2 uses the variant BLAKE2b

Slide 216

## Nonces and IVs

- IVs generally assumed to be unpredictable (though not secret)
- Nonce only need to be unique, e.g. $0, 1, 2, \ldots$
- Nonce re-use is typically "catastrophic", i.e. allows an attacker to break the encryption

Example: if nonces are random, for AES-GCM (96-bit nonce) and $m = 2^{32}$ messages the chance of collision is:

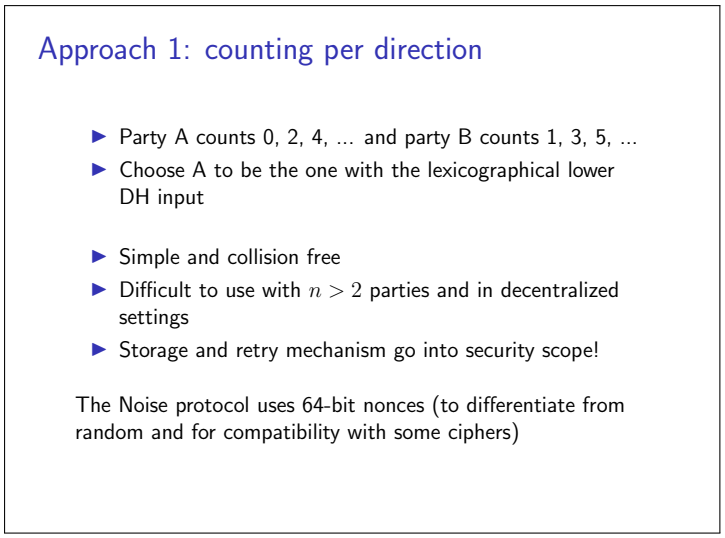
$$p \approx \frac{m^2}{2 \cdot n} = \frac{2^{2 \cdot 32}}{2 \cdot 2^{96}} = 2^{-33}$$

Slide 217

Many encryption schemes require a nonce or IV to be used to achieve IND-CPA security. However, reusing nonces or IVs is potentially catastrophic and can allow the attacker to break the scheme, i.e. decrypt messages. Commonly deployed schemes, such as AES-GCM, are vulnerable to this scenario and come with short 96-bit nonces. Where we choose such short nonces at random, the chance of collision

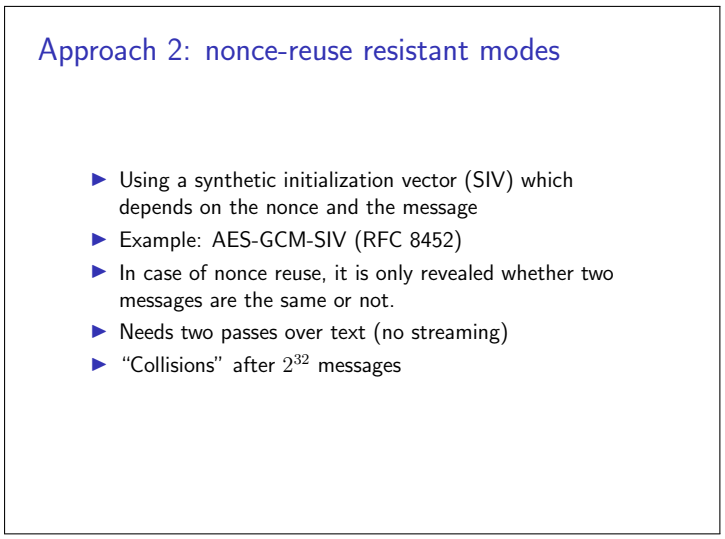
is $\approx 2^{-33}$ for $2^{32}$ messages as per square-root approximation of the birthday paradox: the number of possible nonces $n = 2^{96}$ are the bins, the number of messages $m = 2^{32}$ are the balls, and we are interested in the probability of two balls landing in the same bin. For real world systems, the probability of $2^{-33}$ for a catastrophic failure is uncomfortably high.

One simple approach to mitigate this is to have separate counters for each direction of the protocol. However, this requires additional storage and becomes complex for more than two parties. For instance, if A wants to send a message $M_2$ with nonce $N_2 = N_1 + 1$, it needs to ensure that it has persisted the new counter value $N_2$ to disk before sending the message. Otherwise, it might crash after sending and before persisting the value. In this case, A would likely re-use the same message when they retry after restarting.




Slide 218

Most schemes break exactly when the same nonce/IV is used with different messages under the same key. So, we can use a synthetic initialization vector (SIV) which depends on the nonce and the message. This way, the SIV is unique for each message and the scheme is secure even if the same nonce is used for different messages.




Slide 219

The final approach, which finds its way into many modern schemes, is to make the nonce space very large. This way, the chance of collision is negligible even if we choose the nonce randomly.

## Approach 3: extended nonces

- Make the nonce space very large so that collisions are unlikely
- Typically 192-bit, e.g. XChaCha20-Poly1305, XAES-256-GCM
- Increasingly popular, especially when trying to make misuse-safe APIs

Example: for XAES-256-GCM (192-bit nonce) and $2^{64}$ messages the chance of collision is:

$$p \approx \frac{2^{2 \cdot 64}}{2 \cdot 2^{192}} = 2^{-64}$$

Slide 220

## 6.5 API Design

In our final section, we touch on some high-level aspects of API design. There are few canonical resources on the topic. Instead, we often see that progress in this area is shared through new code and shared experiences from engineers in the form of blog posts, talks, and the like.

## **LAB:** Library design reflections

**Topic:** What third-party cryptography libraries have you worked with?

- Good aspects?
- "Interesting" aspects?
- Dangerous patterns or APIs?

Collect your thoughts for a few minutes and then we'll discuss!

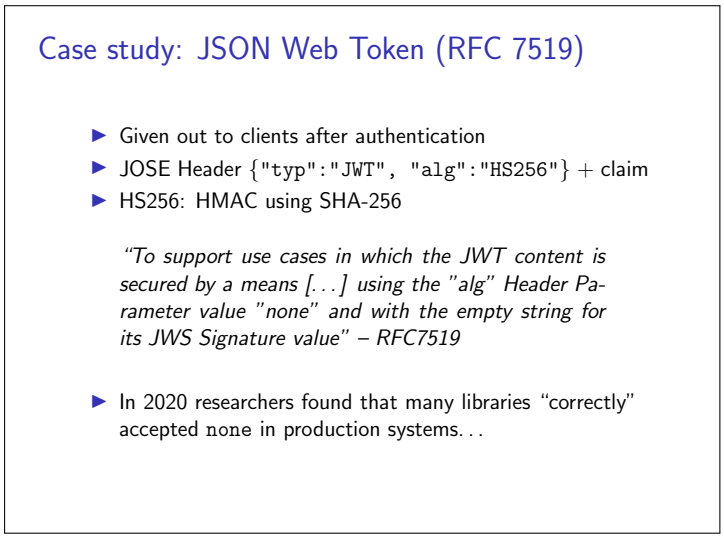
Slide 221

## Cryptographic agility

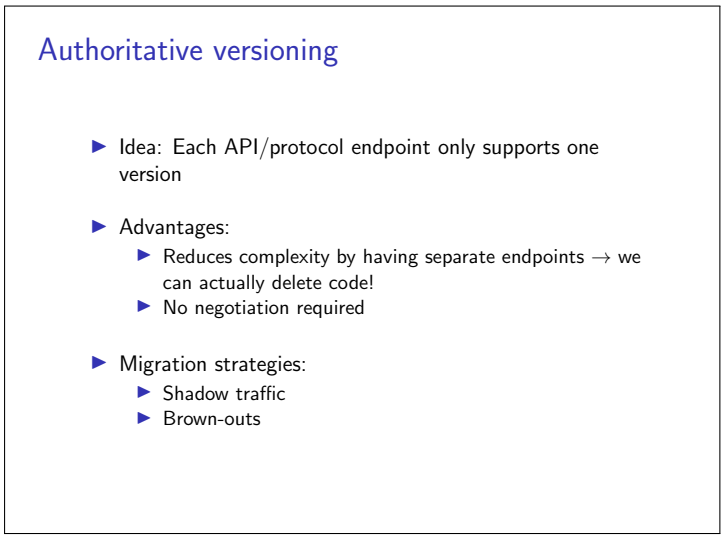
- Idea: allow upgrading the underlying cipher suites
  - Phase out old algorithms
  - Upgrade to new algorithms (e.g. PQC)
- Challenges:
  - Backwards compatibility prevents us from removing old algorithms, see e.g. SHA-1 in TLS
  - Negotiation happens early, i.e. before we establish authenticity.
  - New schemes might require larger underlying changes, e.g. allowed message size
  - Complexity! Specification, implementation, proofs, ...

Slide 222

Cryptographic agility is a property of a protocol that allows it to evolve over time. While this is a desirable property, e.g. in order to phase out weak algorithms, it is very challenging in practice. For instance, once a diverse set of clients and servers are deployed, it is likely that a non-negligible set of these deployments will never update. Thus, when agreeing on a commonly supported subset of algorithms, these stragglers prevent us from completely removing old algorithms, as this would break compatibility. Similarly, most protocols needs to negotiate the algorithms to use before establishing authenticity. As a result, machine-in-the-middle attacks can try to influence these initial messages and force the use of weaker (broken) algorithms. In his blog post on cryptographic agility and version negotiation, Adam Langley suggests: "have one joint and keep it well oiled".




Slide 223




Slide 224

An alternative to more complex version negotiation is to have implementations only implement one version. This immediately allows to remove new code from the newer version and prevents "zombie extensions" that are rarely used, under-tested, and add unnecessary complexity. In a classical client-server setting, this is straightforward as the load-balancer can route to the correct version. Upgrades to clients will only be shipped, once the new version is available. Once prevalence of the older version is negligible, the old version endpoint can be removed.

In practice, two deployment strategies have proven helpful: When testing the new version, have a small number of test clients communicate with the new version in parallel to the old one. This *shadow traffic* allows to detect issues with the new version and any troubles do not have user-visible impact. In a complex environment, once the decision is made to turn-off an old version, it is not necessary clear what other systems are relying on it. Hence, before turning it off entirely, we often have *brown-outs* where the version is temporarily deactivated, e.g. for an increasing number of hours each day, or clients emit very visible warnings.

## Opinionated libraries

- Avoid user misconfiguration and wrong usage
  - Limited choice
  - Secure defaults
  - High-level abstractions
- You have seen a few:
  - LibNacl, LibSodium, LibHydrogen
  - Noise protocol
  - Many Rust libraries
  - ...

Slide 225

Opinionated libraries deliberately limit user choices to prevent misuse. However, over time users often request access to lower-level primitives for building custom protocols. Library maintainer can face pressure to expose these functions, even when doing so increases the attack surface and maintenance burden.

## Case study: Low-level APIs in libsodium (2025)

- Originally designed for high-level operations only
  - Users shouldn't need to know internal algorithms
- Over time, low-level functions were exposed for protocol designers
  - "a lot of code to maintain"
  - Only `--enable-minimal` builds API that is fully supported
  - Allows library consumers to avoid adding additional dependencies
- 2025: Bug in low-level point validation function.
  - High-level APIs (`crypto_sign_*`) unaffected
  - Custom protocol implementations potentially vulnerable

Slide 226

This case study [Denis, 2025] illustrates the inherent tension in cryptographic library design. The high-level APIs that libsodium was originally designed around remained secure; they never needed the buggy validation function. The bug itself – an incomplete check in a point validation function – only affected users who bypassed the high-level APIs to call low-level functions directly.

## Pure functions

- All information and state are explicitly passed
  - Input/output parameters
  - Environment: file system, time, ...
- On embedded systems this might include
  - Memory allocation
  - Secure random generator
- Added benefit: drastically simplifies testing!

Slide 227

## Managing secret life times

- We want to minimize the lifetime of secrets in memory
  - Quickly transform into derived secrets
  - "Erase" memory
- Have the language help us with this
  - Rust: `Drop` handler
  - C++: RAII pattern
  - Java: `AutoCloseable`, `finalize`
  - ...
- Difficult in user interfaces which are often not under our control
  - Also, side-channels such as keyboard predictions

Slide 228

## Rust

- `Drop` handler can automatically erase memory
- Non trivial: making sure the compiler does not optimize this step away

```
use zeroize::{Zeroize, ZeroizeOnDrop};

#[derive(ZeroizeOnDrop)]
struct SessionKey { bytes: [u8; 16] }
```

Slide 229

### This is harder in non-native languages

- Java/... do not provide direct access to memory
- The GC might inadvertely copy our secrets

```
pw = byte[16];
read_pw(pw);

// do work

Arrays.fill(pw, 0);
```

Slide 230

Ensuring that secrets are effectively removed from memory is challenging in languages that abstract away the physical memory. That includes most languages that rely on garbage collection (e.g. Java, Python), as the GC might copy our secret to different memory locations during its operation. Hence, when we try to override the memory, we might only changes values at the new location. The example in Slide 230 is not effective, as the GC might have copied `pw` to a new location before we fill it with zeros.

However, most of these languages provide some methods to interact with physical memory addresses through Foreign Function Interfaces (FFI). In Java, we can also use the `ByteBuffer.allocateDirect` method to allocate memory that is not managed by the GC, but lives at a fixed physical address. Thus, we can override the memory later and be sure that our secrets are indeed removed from memory. In the example in Slide 231, we clear the buffer by rewinding the cursor to the beginning and writing zeros.

### This is harder in non-native languages

- Use `ByteBuffer` or equivalent to manage directly allocated memory

```
pw = ByteBuffer.allocateDirect(16);
read_pw(pw);

// do work

pw.rewind();
pw.put(new byte[16]);
```

Slide 231

## This is harder in non-native languages

Add safety by using `finalize` method which is (supposed to be) called just before an object is garbage collected.

- Variant A: perform the operation for the user (might be delayed or unreliable)
- Variant B: cause crashes in debug builds (see e.g. Android's strict mode)

Slide 232

## Locking memory

- Prevent swapping out to disk
  - Extra motivation why swap should always be encrypted
- On Linux we have to simple call `mlock` and `munlock`

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

Slide 233

## Case study: crash dump leak (MS STORM-0558)

- Microsoft has an isolated production environment for signing final artifacts
- A crash dump from this environment was exported for investigation
  - Usually, secrets are filtered out from crash dumps (this was broken)
  - Usually, secrets would be detected and revoked by other systems (this was broken)
- Attackers had access to the investigation machine which was outside the isolated environment
- Libraries did not automatically validate the key scope
- Result: attackers were able to access enterprise emails using forged tokens.

Slide 234

# References

Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In *Cryptographers' Track at the RSA Conference*, CT-RSA, pages 191–208. Springer, February 2005. doi:10.1007/978-3-540-30574-3_14.

Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy*, pages 526–540. IEEE, 2013.

Kent Beck. *Test driven development: by example*. Addison-Wesley Professional, 2022.

Daniel J Bernstein. Curve25519: New Diffie-Hellman speed records. In *9th International Conference on Theory and Practice in Public-Key Cryptography*, PKC 2006, pages 207–228. Springer, April 2006. doi:10.1007/11745853_14. URL https://cr.yp.to/ecdh/curve25519-20060209.pdf.

Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, September 2012. ISSN 2190-8508. doi:10.1007/s13389-012-0027-1. URL https://ed25519.cr.yp.to/ed25519-20110926.pdf.

Cliff L Biffle. The typestate pattern in Rust, June 2019. URL https://cliffle.com/blog/rust-typestate/.

Alex Biryukov, Daniel Dinu, Dmitry Khovratovich, and Simon Josefsson. Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications. RFC 9106, September 2021. URL https://www.rfc-editor.org/info/rfc9106.

Simon Blake-Wilson and Alfred Menezes. Unknown key-share attacks on the Station-to-Station (STS) protocol. In *2nd International Workshop on Practice and Theory in Public Key Cryptography*, PKC. Springer, March 1999. doi:10.1007/3-540-49162-7_12.

Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*, pages 1–12. Springer, 1998.

Daniel Bleichenbacher et al. Project Wycheproof. URL https://github.com/C2SP/wycheproof.

Jeremiah Blocki, Benjamin Harsha, and Samson Zhou. On the economics of offline password cracking. In *2018 IEEE Symposium on Security and Privacy (S&P)*, pages 853–871. IEEE, 2018.

Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280, May 2008. URL https://datatracker.ietf.org/doc/html/rfc5280.

Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use PGP. In *ACM Workshop on Privacy in the Electronic Society*, WPES, page 77–84. ACM, 2004. doi:10.1145/1029179.1029200.

Ran Canetti and Hugo Krawczyk. Security analysis of IKE's signature-based key-exchange protocol. In *22nd Annual International Cryptology Conference*, CRYPTO, 2002. doi:10.1007/3-540-45708-9_10. Full version at https://eprint.iacr.org/2002/120.

Pranav Dahiya, Ilia Shumailov, and Ross Anderson. Machine learning needs better randomness standards: Randomised smoothing and {PRNG-based} attacks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3657–3674, 2024.

Henry de Valence. It's 255:19am. Do you know what your validation criteria are?, October 2020. URL https://hdevalence.ca/blog/2020-10-04-its-25519am.

Henry de Valence, Jack Grigg, George Tankersley, Filippo Valsorda, and Isis Lovecruft. The ristretto255 group. IETF Internet-Draft, May 2020. URL https://www.ietf.org/archive/id/draft-irtf-cfrg-ristretto255-00.html.

Frank Denis. A vulnerability in libsodium. Blog post, December 2025. URL https://00f.net/2025/12/30/libsodium-vulnerability/.

Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992. ISSN 1573-7586. doi:10.1007/BF00124891.

Michael Driscoll. The illustrated TLS 1.3 connection: Every byte explained and reproduced, 2018. URL https://tls13.xargs.org/.

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic: with proofs, without compromises. *ACM SIGOPS Operating Systems Review*, 54(1):23–30, 2020.

fail0verflow. Console hacking 2010: PS3 epic fail. In *27th Chaos Communication Congress*, December 2010. URL https://media.ccc.de/v/27c3-4087-en-console_hacking_2010.

Daniel Genkin, Luke Valenta, and Yuval Yarom. May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 845–858. ACM, October 2017. doi:10.1145/3133956.3134029.

Feng Hao and Paul C. van Oorschot. SoK: Password-Authenticated Key Exchange – theory, practice, standardization and real-world lessons. In *ACM Asia Conference on Computer and Communications Security*, ASIACCS, pages 697–711. ACM, May 2022. doi:10.1145/3488932.3523256. URL https://eprint.iacr.org/2021/1492.pdf.

Daniel Hugenroth, Sam Cutler, Dominic Kendrick, Mario Savarese, Zeke Hunter-Green, Philip McMahon, Marjan Kalanaki, Diana A. Vasile, Sabina Bejasa-Dimmock, Luke Hoyland, and Alastair R. Beresford. CoverDrop White Paper. Technical Report UCAM-CL-TR-999, University of Cambridge, Computer Laboratory, June 2025. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-999.pdf.

ITU-T. X. 680 ISO/IEC 8824-1: 2002, Abstract Syntax Notation One (ASN. 1): Specification of basic notation, 2002a. URL https://www.itu.int/rec/T-REC-X.680-202102-I.

ITU-T. X. 690 ISO/IEC 8824-1: 2002, ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), canonical encoding rules (cer) and distinguished encoding rules (der)., 2002b. URL https://www.itu.int/rec/T-REC-X.690-202102-I.

Dennis Jackson, Cas Cremers, Katriel Cohn-Gordon, and Ralf Sasse. Seems legit: Automated analysis of subtle attacks on protocols that use signatures. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 2165–2180. ACM, 2019. doi:10.1145/3319535.3339813.

Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part III 37*, pages 456–486. Springer, 2018.

Simon Josefsson and Ilari Liusvaara. Edwards-curve digital signature algorithm (EdDSA). RFC 8032, January 2017. URL https://www.rfc-editor.org/rfc/rfc8032.html.

Hubert Kario. "constant time" compare in python, August 2018. URL https://securitypitfalls.wordpress.com/2018/08/03/constant-time-compare-in-python/.

Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, third edition, December 2020. doi:10.1201/9781351133036.

Martin Kleppmann. Implementing Curve25519/X25519: A tutorial on elliptic curve cryptography. Draft manuscript, October 2020. URL https://martin.kleppmann.com/papers/curve25519.pdf.

Hugo Krawczyk. SIGMA: the 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols. In *23rd Annual International Cryptology Conference*, CRYPTO, 2003. doi:10.1007/978-3-540-45146-4_24. URL http://iacr.org/archive/crypto2003/27290399/27290399.pdf.

Watson Ladd. SPAKE2, a password-authenticated key exchange. RFC 9382, September 2023. URL https://datatracker.ietf.org/doc/rfc9382/.

Adam Langley, Mike Hamburg, and Sean Turner. Elliptic curves for security. RFC 7748, January 2016. URL https://tools.ietf.org/html/rfc7748.

Ben Laurie. Certificate transparency. *ACM Queue*, 12(8):10–19, August 2014. doi:10.1145/2668152.2668154.

Neil Madden. CVE-2022-21449: Psychic signatures in Java, April 2022. URL https://neilmadden.blog/2022/04/19/psychic-signatures-in-java/.

Moxie Marlinspike and Trevor Perrin. The X3DH key agreement protocol, November 2016. URL https://signal.org/docs/specifications/x3dh/.

Karissa Rae McKelvey, Benjamin Royer, Chris (daiyi) Sun, Cade Diehm, and Peter van Hardenberg. Backchannel: A relationship-based digital identity system. Technical report, Ink & Switch, September 2021. URL https://www.inkandswitch.com/backchannel/.

Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *24th USENIX Security Symposium*, 2015. URL https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-melara.pdf.

NIST. A statistical test suite for random and pseudorandom number generators for cryptographic applications. 2010. URL https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-22r1a.pdf.

Lawrence C Paulson. *Isabelle: A generic theorem prover*. Springer, 1994.

Thomas Pornin. Why constant-time crypto?, 2017. URL https://www.bearssl.org/constanttime.html.

Eric Rescorla. The Transport Layer Security (TLS) protocol version 1.3. RFC 8446, August 2018. URL https://datatracker.ietf.org/doc/html/rfc8446.

Florentin Rochet and Olivier Pereira. Dropping on the edge: Flexibility and traffic confirmation in onion routing protocols. *Proceedings on Privacy Enhancing Technology*, 2018(2):27–46, 2018. URL https://petsymposium.org/2018/files/papers/issue2/popets-2018-0011.pdf.

Markus Schiffermuller. We found cryptography bugs in the elliptic library using Wycheproof. Trail of Bits Blog, November 2025. URL https://blog.trailofbits.com/2025/11/18/we-found-cryptography-bugs-in-the-elliptic-library-using-wycheproof/.

Simon Tatham. PuTTY vulnerability vuln-p521-bias, April 2024. URL https://www.chiark.greenend.org.uk/~sgtatham/putty/wishlist/vuln-p521-bias.html.

Serge Vaudenay. Security flaws induced by CBC padding—applications to SSL, IPSEC, WTLS, ... In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 534–545. Springer, 2002.

Brian Warner. SPAKE2 "random" elements, January 2016. URL https://www.lothar.com/blog/54-spake2-random-elements/. Archived at https://perma.cc/A93R-RDLE.

Brian Warner. SPAKE2 interoperability, July 2017. URL https://www.lothar.com/blog/57-SPAKE2-Interoperability/. Archived at https://perma.cc/WWM3-5UAV.