P79: Cryptography and Protocol Engineering

University of Cambridge
MPhil in Advanced Computer Science / Computer Science Tripos, Part III
Lent term 2024/25
https://www.cl.cam.ac.uk/teaching/2425/P79/

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

P79: Cryptography and Protocol Engineering

https://www.cl.cam.ac.uk/teaching/2425/P79/

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

University of Cambridge
Part III/MPhil in Advanced Computer Science

Motivation

- The Internet (and hence, the modern world) would not function without cryptography
- Cryptography has a reputation of being somehow magic

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

About this module

- Running first time in 2024/25
  - Thank you for being our guinea pigs :)
  - Sorry for teething troubles!

About this module

- Running first time in 2024/25
  - Thank you for being our guinea pigs :)
  - Sorry for teething troubles!
- 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

About this module

- Running first time in 2024/25
  - Thank you for being our guinea pigs :)
  - Sorry for teething troubles!
- 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

“Don’t roll your own crypto!”

- Cryptography implementations are very prone to subtle flaws that **completely break** the intended security properties

“Don’t roll your own crypto!”

- Cryptography implementations are very prone to subtle flaws that **completely break** the intended security properties
- Production software (where harm could result if it’s broken) should use **expert-audited**, preferably **formally verified** code
  - And those expert audits are not cheap

“Don’t roll your own crypto!”

- Cryptography implementations are very prone to subtle flaws that **completely break** the intended security properties
- Production software (where harm could result if it’s broken) should use **expert-audited**, preferably **formally verified** code
  - And those expert audits are not cheap
- But if you want to become one of those experts yourself, reading and writing crypto code is a part of the journey
- If you put your code on GitHub, please add a big **warning label**!

Module objectives

By the end of this module, you should hopefully...

- appreciate real-world cryptography
- know the mathematical notation and concepts used in crypto protocols
- be able to understand and implement research papers and crypto standards
- use crypto libraries correctly
- have tried some attacks on crypto protocols
- got a glimpse into recent research
- got better at technical writing (through lab reports)

Assessment

- Three lab report + code submissions:
  1. Elliptic curve cryptography (~~10~~ 17 Feb 2025)
  2. Authenticated key exchange (3 Mar 2025)
  3. Private information retrieval (24 Mar 2025)
- The worst of the 3 marks will be discarded; the remaining 2 marks count equally

Assessment

- Three lab report + code submissions:
  1. Elliptic curve cryptography (~~10~~ 17 Feb 2025)
  2. Authenticated key exchange (3 Mar 2025)
  3. Private information retrieval (24 Mar 2025)
- The worst of the 3 marks will be discarded; the remaining 2 marks count equally
- 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

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

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

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
- Simulate the network within a single process (no need for real networking, but do encode/decode to bytes)

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
- Simulate the network within a single process (no need for real networking, but do encode/decode to bytes)
- Don't bother building user interfaces

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
- Submit as a `.zip` archive that includes at least:
  - Your code
  - A `Dockerfile` that 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.

Code: what we look for

- **Correctness** $\gg$ **robustness** $\gg$ performance
  - Make it easy for us to verify this by including tests with known input output pairs

Code: what we look for

- **Correctness** $\gg$ **robustness** $\gg$ performance
  - Make it easy for us to verify this by including tests with known input output pairs
- **Tests**

Code: what we look for

- **Correctness** $\gg$ **robustness** $\gg$ performance
  - Make it easy for us to verify this by including tests with known input output pairs
- **Tests**
- **Design of your API:** typing, error handling, naming, consistency

Code: what we look for

- **Correctness** $\gg$ **robustness** $\gg$ performance
  - Make it easy for us to verify this by including tests with known input output pairs
- **Tests**
- **Design of your API:** typing, error handling, naming, consistency
- **Documentation:** high-level picture, do not comment every line, make meaningful comments, API language should be self-documenting

Code: what we look for

- **Correctness** $\gg$ **robustness** $\gg$ performance
    - Make it easy for us to verify this by including tests with known input output pairs
- **Tests**
- **Design of your API:** typing, error handling, 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, . . .

Lab reports

> *Remember, kids: the only difference between screwing around and science is writing it down.*
> – *Adam Savage*

2,000 words, explaining key aspects of your code:

- How it works
- Why it's correct

Lab reports

> *Remember, kids: the only difference between screwing around and science is writing it down.*
> *– Adam Savage*

2,000 words, explaining key aspects of your code:

- How it works
- Why it's correct
- Any findings from your work (e.g. limitations or trade-offs you found)
- What you'd need to change to make it production-quality

Lab reports

> *Remember, kids: the only difference between screwing around and science is writing it down.*
> *– Adam Savage*

2,000 words, explaining key aspects of your code:

- How it works
- Why it's correct
- Any findings from your work (e.g. limitations or trade-offs you found)
- What you'd need to change to make it production-quality
- Other insights, e.g. how it compares with other implementations (performance or otherwise)

Opportunity for your critical insights and creativity!

Lab report requirements

Written in LaTeX

No fixed structure, but should contain:

- Explanation of core ideas behind your code

Lab report requirements

Written in LaTeX

No fixed structure, but should contain:

- Explanation of core ideas behind your code
- How do you know that it is correct?
    - Minimum acceptable: “I copied it from the RFC”
    - Better: tests, types, derived formulas yourself
    - Ideal (but not required for this module): formal proof

Lab report requirements

Written in LaTeX

No fixed structure, but should contain:

- Explanation of core ideas behind your code
- How do you know that it is correct?
    - Minimum acceptable: “I copied it from the RFC”
    - Better: tests, types, derived formulas yourself
    - Ideal (but not required for this module): formal proof
- References to relevant literature
(citations not included in word limit)

Lab report requirements

Written in LaTeX

No fixed structure, but should contain:

- Explanation of core ideas behind your code
- How do you know that it is correct?
    - Minimum acceptable: “I copied it from the RFC”
    - Better: tests, types, derived formulas yourself
    - Ideal (but not required for this module): formal proof
- References to relevant literature
  (citations not included in word limit)

Assessment criteria:

- Correctness and clarity of explanations
- Critical reflection
- High marks require significant creative insight

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

Basic cryptography recap

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

University of Cambridge
Part III/MPhil in Advanced Computer Science

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).

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

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 python # or equivalent on your OS
python3 -m venv .venv # creates subdirectory ".venv"
source .venv/bin/activate
pip install pynacl
```

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

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

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 $H(x) = H(y)$

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

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...
```

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

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

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)

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)

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

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))
```

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**:

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”)

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

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)

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)

Asymmetric (public key) cryptography

Hash functions, symmetric ciphers

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

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. . .)

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

Asymmetric (public key) cryptography

Hash functions, symmetric ciphers

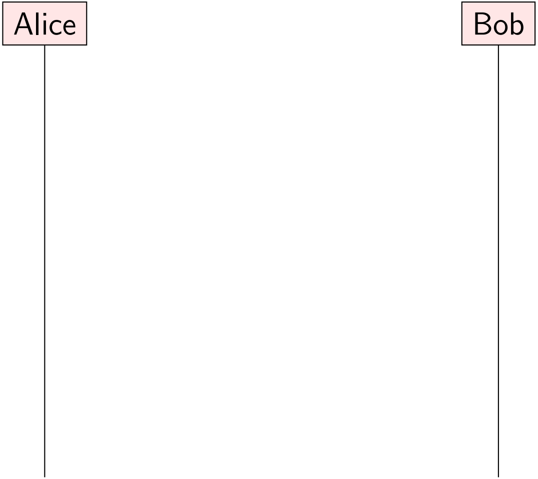
- Lots of bit shifts, XORs, lookup tables
- Not much underlying mathematical structure

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

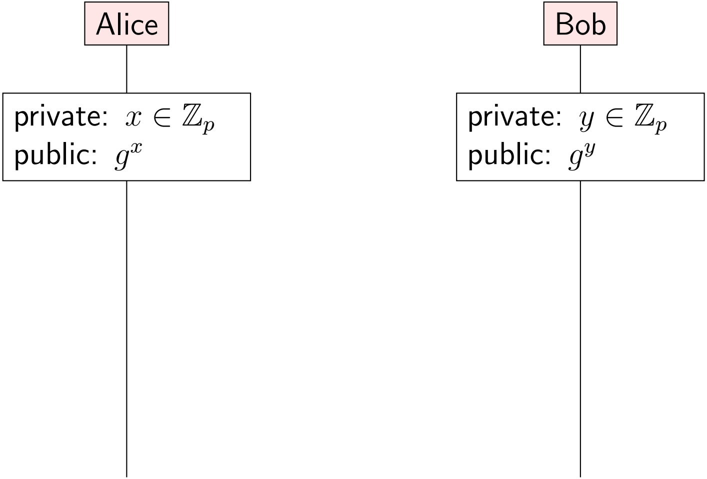
Diffie-Hellman

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



Diffie-Hellman

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



Alice

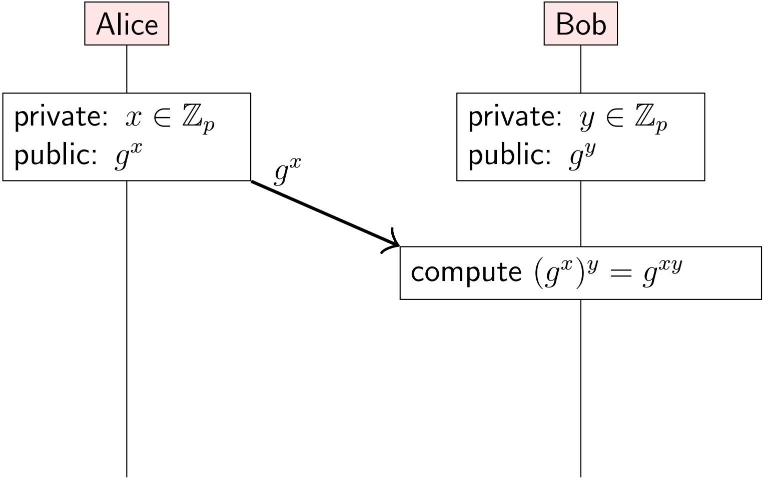
private: $x \in \mathbb{Z}_p$
public: g^x

Bob

private: $y \in \mathbb{Z}_p$
public: g^y

Diffie-Hellman

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



Alice

private: $x \in \mathbb{Z}_p$
public: g^x

g^x

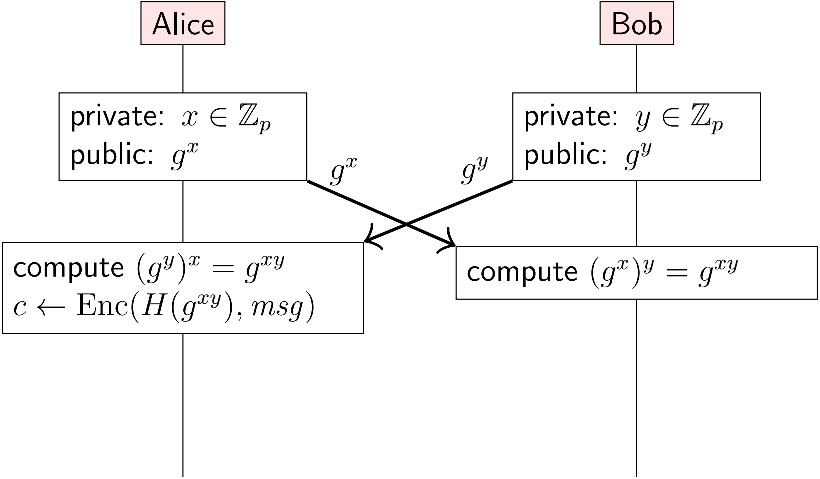
Bob

private: $y \in \mathbb{Z}_p$
public: g^y

compute $(g^x)^y = g^{xy}$

Diffie-Hellman

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



Alice

Bob

private: $x \in \mathbb{Z}_p$
public: g^x

private: $y \in \mathbb{Z}_p$
public: g^y

g^x

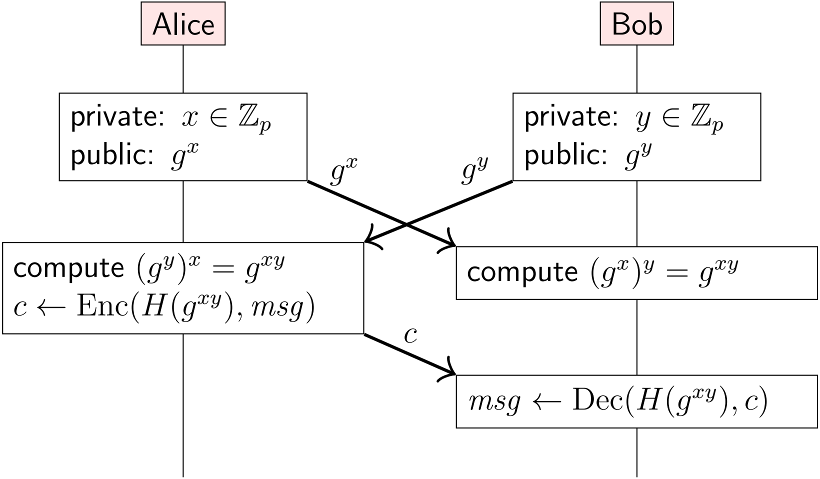
g^y

compute $(g^y)^x = g^{xy}$
$c \leftarrow \mathrm{Enc}(H(g^{xy}), msg)$

compute $(g^x)^y = g^{xy}$

Diffie-Hellman

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



Alice

Bob

private: $x \in \mathbb{Z}_p$
public: g^x

private: $y \in \mathbb{Z}_p$
public: g^y

g^x

g^y

compute $(g^y)^x = g^{xy}$
$c \leftarrow \mathrm{Enc}(H(g^{xy}), msg)$

compute $(g^x)^y = g^{xy}$

c

$msg \leftarrow \mathrm{Dec}(H(g^{xy}), c)$

Diffie-Hellman

- sk = private (secret) key, pk = public key
- $\mathrm{DH}(sk_A, pk_B) = \mathrm{DH}(sk_B, pk_A)$

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

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

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())
```

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

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

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

Asymmetric (public key) encryption

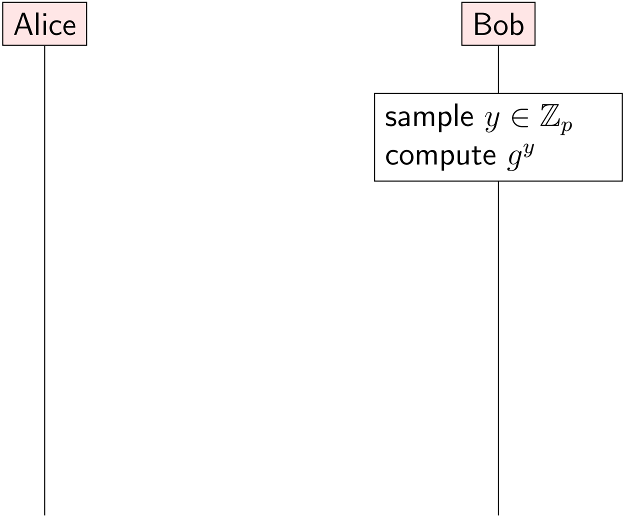
- $(pk, sk) \leftarrow \mathrm{Gen}()$ generates keypair (pk public, sk secret)
- $c \leftarrow \mathrm{Enc}(pk, msg)$ returns ciphertext c
- $msg \leftarrow \mathrm{Dec}(sk, c)$ decrypts c, returns msg or error
- Unauthenticated: anyone who knows pk can encrypt
- e.g. RSAES-OAEP, Hybrid Public Key Encryption
- 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

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))
```

Public key encryption from Diffie-Hellman

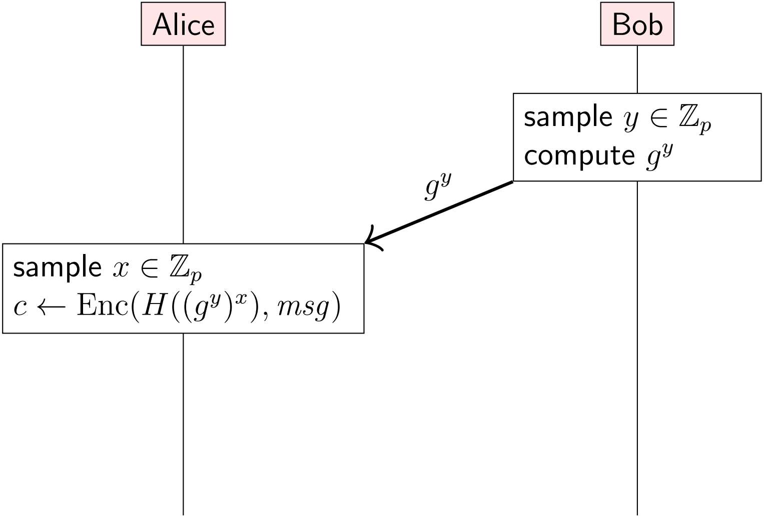


Alice

Bob

sample $y \in \mathbb{Z}_p$
compute g^y

Public key encryption from Diffie-Hellman



Alice

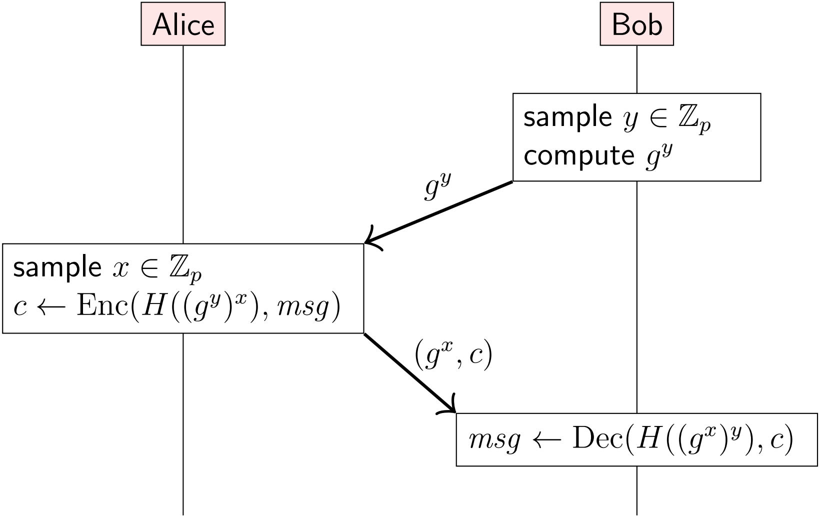
Bob

sample $y \in \mathbb{Z}_p$
compute g^y

g^y

sample $x \in \mathbb{Z}_p$
$c \leftarrow \mathrm{Enc}(H((g^y)^x), msg)$

Public key encryption from Diffie-Hellman



Alice

Bob

sample $y \in \mathbb{Z}_p$
compute g^y

g^y

sample $x \in \mathbb{Z}_p$
$c \leftarrow \mathrm{Enc}(H((g^y)^x), msg)$

(g^x, c)

$msg \leftarrow \mathrm{Dec}(H((g^x)^y), c)$

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

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

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'
```

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

Lab time

- Clone the sample code: `git clone https://github.com/lambdapioneer/p79-sample.git`
- Install Python 3.12 and Docker
- Run everything: `./run.sh`
- Fix the tests
- . . .
- Critique!

Elliptic Curve Cryptography (ECC)

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

University of Cambridge
Part III/MPhil in Advanced Computer Science

Introducing Elliptic Curve Cryptography

- Very widely used – protects majority of Internet traffic
- Key agreement: Elliptic Curve Diffie Hellman (ECDH)
- Digital signatures: Elliptic Curve Digital Signature Algorithm (ECDSA) or Edwards Curve Digital Signature Algorithm (EdDSA)

Introducing Elliptic Curve Cryptography

- Very widely used – protects majority of Internet traffic
- Key agreement: Elliptic Curve Diffie Hellman (ECDH)
- Digital signatures: Elliptic Curve Digital Signature Algorithm (ECDSA) or Edwards Curve Digital Signature Algorithm (EdDSA)
- Lots of funky advanced stuff also possible
- Faster than RSA, DSA; smaller keys and signatures (at same security level)

Introducing Elliptic Curve Cryptography

- Very widely used – protects majority of Internet traffic
- Key agreement: Elliptic Curve Diffie Hellman (ECDH)
- Digital signatures: Elliptic Curve Digital Signature Algorithm (ECDSA) or Edwards Curve Digital Signature Algorithm (EdDSA)
- Lots of funky advanced stuff also possible
- Faster than RSA, DSA; smaller keys and signatures (at same security level)

In this module:

- We will use ECC for the first two assignments
- You need to implement it from scratch, using only Python's built-in primitives
- Suggested reading: Martin's Curve25519 tutorial

(Abelian) Groups

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

	additive	**multiplicative**
closed: $\forall a, b \in E.\ a \bullet b \in E$	$a + b \in E$	$ab \in E$

(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: $\forall a, b \in E.\ a \bullet b = b \bullet a$	$a + b = b + a$	$ab = ba$

(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: $\forall a, b, c \in E.$ $(a \bullet b) \bullet c = a \bullet (b \bullet c)$	$(a + b) + c =$ $a + (b + c)$	$(ab)c = a(bc)$

(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: $\exists id \in E.\ \forall a \in E.\ a \bullet id = a$	$a + 0 = a$	$a \cdot 1 = a$

(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: $\forall a \in E.\ \exists b \in E.\ a \bullet b = id$	$a + (-a) = 0$	$a \cdot a^{-1} = 1$

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 $-a = n - a$

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 $\gcd(a, n) = 1$

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

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
```

Fields

A set E and two operations $+$, $\cdot$ such that:

- $(E, +)$ is an abelian group with identity 0
- $(E \setminus \{0\}, \cdot)$ is an abelian group with identity 1
- Distributive: $a \cdot (b + c) = ab + ac$

Fields

A set E and two operations $+$, $\cdot$ such that:

- $(E, +)$ is an abelian group with identity 0
- $(E \setminus \{0\}, \cdot)$ is an abelian group with identity 1
- Distributive: $a \cdot (b + c) = ab + ac$

For convenience we will write:

- $a - b = a + (-b)$ where $-b$ is the additive inverse
- $\frac{a}{b} = a \cdot b^{-1}$ where b^{-1} is the multiplicative inverse

Arithmetic works like what you learnt in secondary school.

Fields

A set E and two operations $+$, $\cdot$ such that:

- $(E, +)$ is an abelian group with identity 0
- $(E \setminus \{0\}, \cdot)$ is an abelian group with identity 1
- Distributive: $a \cdot (b + c) = ab + ac$

For convenience we will write:

- $a - b = a + (-b)$ where $-b$ is the additive inverse
- $\frac{a}{b} = a \cdot b^{-1}$ where b^{-1} is the multiplicative inverse

Arithmetic works like what you learnt in secondary school.

Finite field (Galois field) uses a finite set:

- We'll use $\mathbb{F}_p$: integers modulo p where **order** p is prime
- Also written $GF(p)$
- Fields $\mathbb{F}_n$ also exist when $n = p^k$, p prime, $k > 1$

Implementing finite fields

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

- In particular, $p = 2^{255} - 19 =$ `0x7fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffed` (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

Implementing finite fields

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

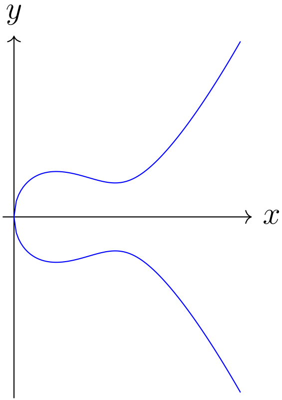
- In particular, $p = 2^{255} - 19 =$ `0x7fffffffffffffffffffffffffffffff` `ffffffffffffffffffffffffffffffed` (255 bits long)
- Python integers have no fixed size: 255-bit (or bigger) ints are no problem
  - Addition `(a + b) % p` is fine
  - Subtraction `(a - b) % p` is fine
  - Multiplication `(a * 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)

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

Montgomery curves (a family of elliptic curves)



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

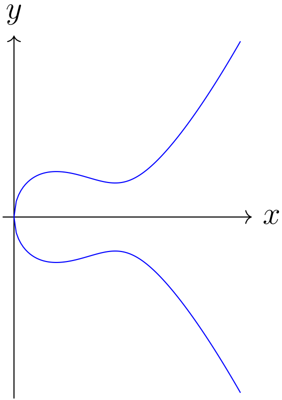
$$y^2 = x^3 + ax^2 + x$$

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 $a = -1.9$.

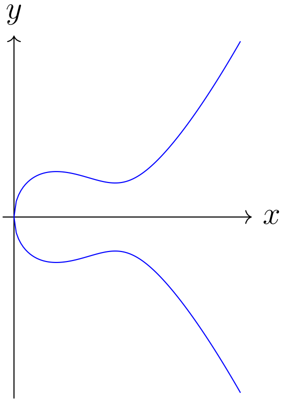
Constructing a group from a curve



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

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

Constructing a group from a curve

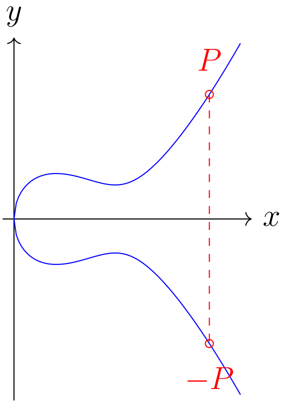


We will now define a **group** whose elements E are **points on the curve** (plus one special element ∞ called "point at infinity").

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

Define identity element as ∞

Constructing a group from a curve



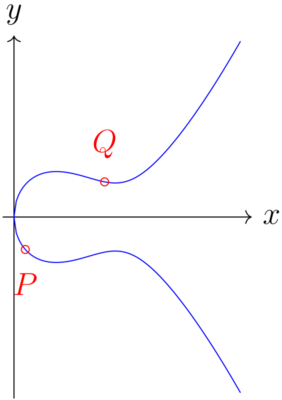
We will now define a **group** whose elements E are **points on the curve** (plus one special element ∞ called "point at infinity").

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

Define identity element as ∞

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

Constructing a group from a curve



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

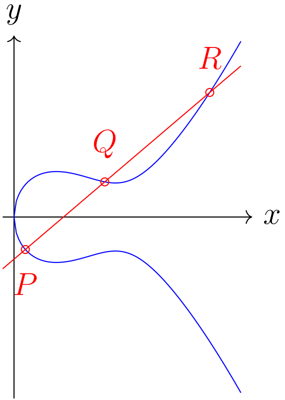
$$E = \{(x, y) \mid y^2 = x^3 + ax^2 + x\} \cup \{\infty\}$$

Define identity element as ∞

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

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

Constructing a group from a curve



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

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

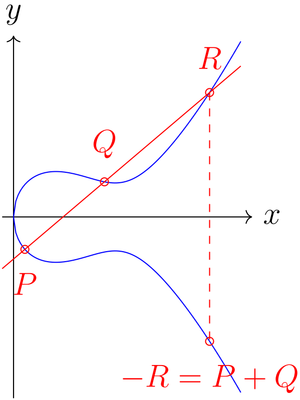
Define identity element as ∞

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

Constructing a group from a curve



We will now define a **group** whose elements E are **points on the curve** (plus one special element ∞ called “point at infinity”).

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

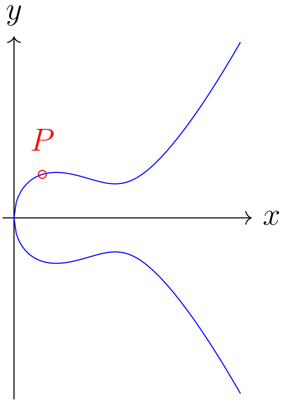
Define identity element as ∞

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 $P + Q$

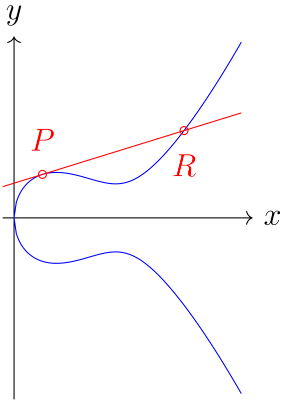
Adding a point to itself (doubling)



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

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

Adding a point to itself (doubling)

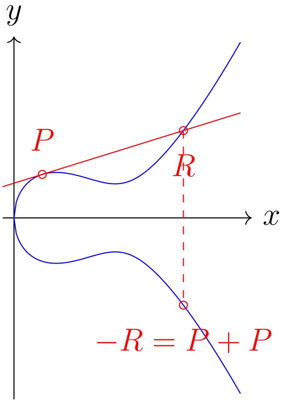


The definition on the last slide works when $P \neq Q$ and $P \neq -Q$.

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

- Draw a tangent to the curve at P
- It intersects the curve at R

Adding a point to itself (doubling)

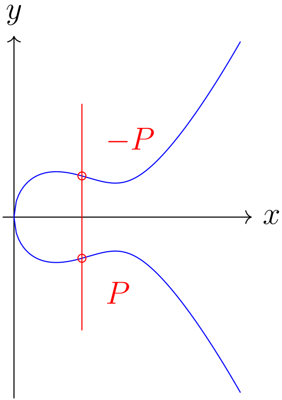


The definition on the last slide works when $P \neq Q$ and $P \neq -Q$.

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

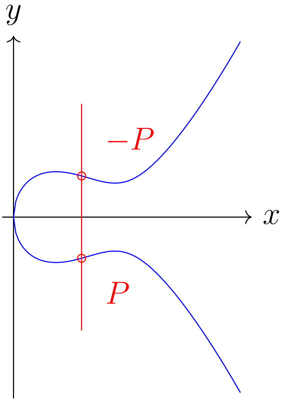
- Draw a tangent to the curve at P
- It intersects the curve at R
- Mirror R by x axis to get $P + P$

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).

Handling vertical lines

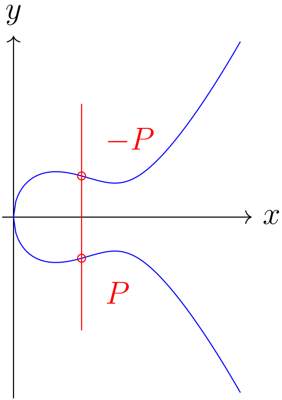


The final case we need to handle is $P + Q$ where $Q = -P$ (i.e. P and Q have the same x coordinate, but different y coordinates).

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

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

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 ∞ as identity element.

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

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.)

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

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

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

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

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

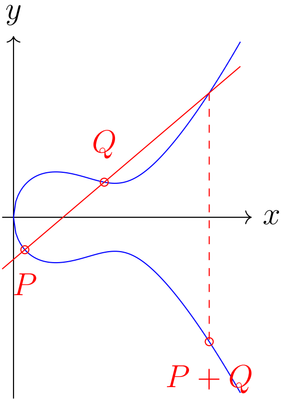
$$y_3 = \frac{(2x_1 + x_2 + a)(y_2 - y_1)}{x_2 - x_1} - \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^3 - y_1$$

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

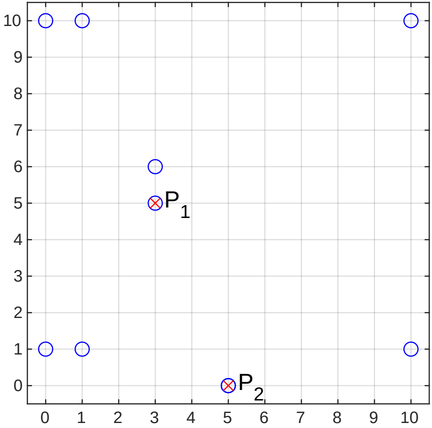
$$x_3 = \left(\frac{3x_1^2 + 2ax_1 + 1}{2y_1}\right)^2 - a - 2x_1$$

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

Elliptic curve over a finite field



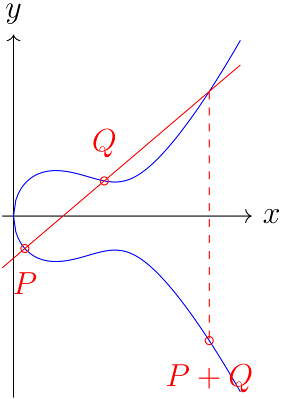
Curve over $\mathbb{R}$



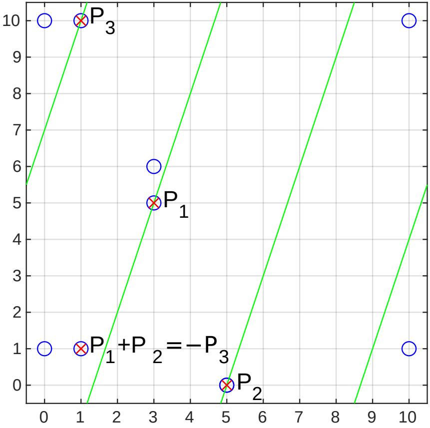
Curve over $\mathbb{F}_{11}$

Image by Markus Kuhn

Elliptic curve over a finite field



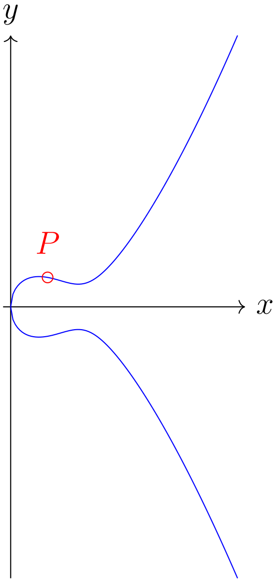
Curve over $\mathbb{R}$



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

Image by Markus Kuhn

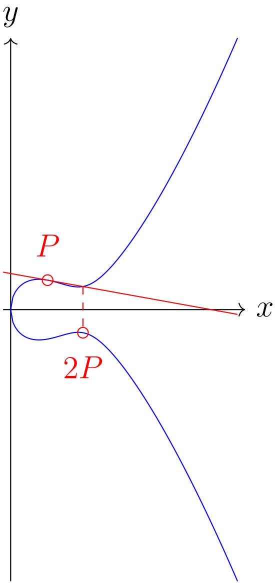
Repeated addition of a point to itself



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

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

Repeated addition of a point to itself

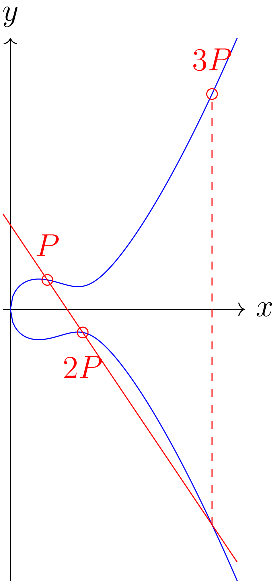


Define **scalar multiplication** of a point $P \in E$ as:

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

If you look at the sequence of points P, $2P$, $3P$, $\ldots$

Repeated addition of a point to itself

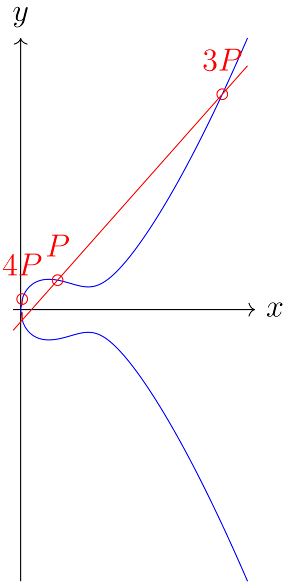


Define **scalar multiplication** of a point $P \in E$ as:

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

If you look at the sequence of points P, $2P$, $3P$, ...

Repeated addition of a point to itself

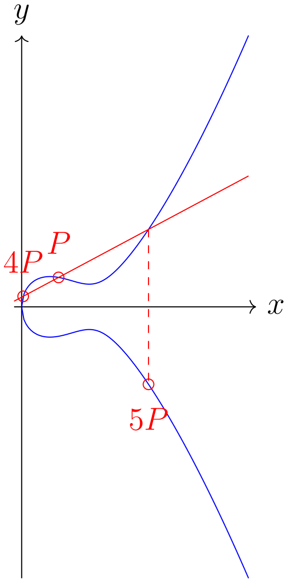


Define **scalar multiplication** of a point $P \in E$ as:

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

If you look at the sequence of points P, $2P$, $3P$, …

Repeated addition of a point to itself

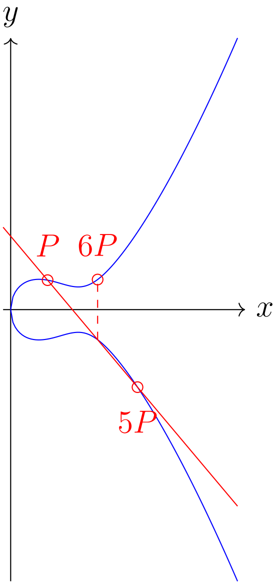


Define **scalar multiplication** of a point $P \in E$ as:

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

If you look at the sequence of points P, $2P$, $3P$, …

Repeated addition of a point to itself

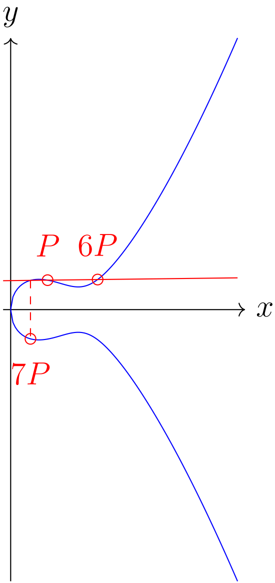


Define **scalar multiplication** of a point $P \in E$ as:

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

If you look at the sequence of points P, $2P$, $3P$, ...

Repeated addition of a point to itself

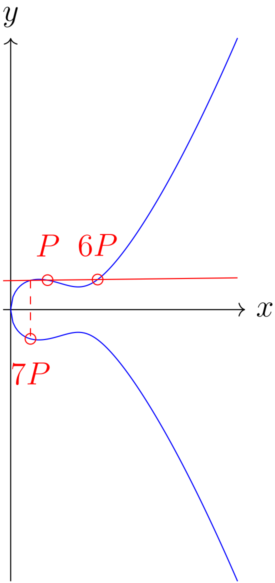


Define **scalar multiplication** of a point $P \in E$ as:

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

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

Repeated addition of a point to itself



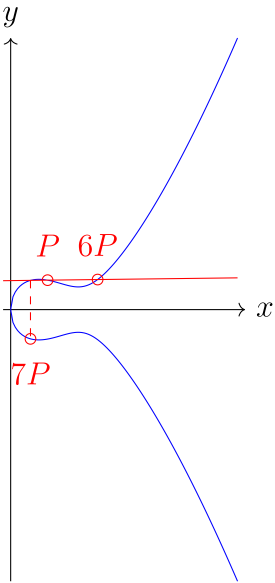
Define **scalar multiplication** of a point $P \in E$ as:

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

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.

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!)

Multiplying a point by a number

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

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

Multiplying a point by a number

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

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

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:

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

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

Generator of a group

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

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

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.

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, $2P$, $3P$, $\ldots$

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

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, $2P$, $3P$, $\ldots$

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. $|\langle P \rangle| = |P|$

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, $2P$, $3P$, $\ldots$

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.

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, $2P$, $3P$, $\ldots$

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).

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}

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

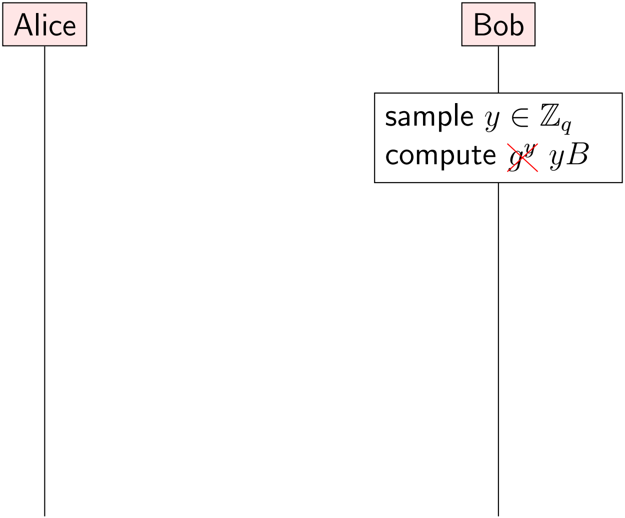
- compute k given kB and B
- compute k given g^k and g

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

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

Elliptic Curve Diffie-Hellman (ECDH)

Public parameter: base point $B \in E$ with order q



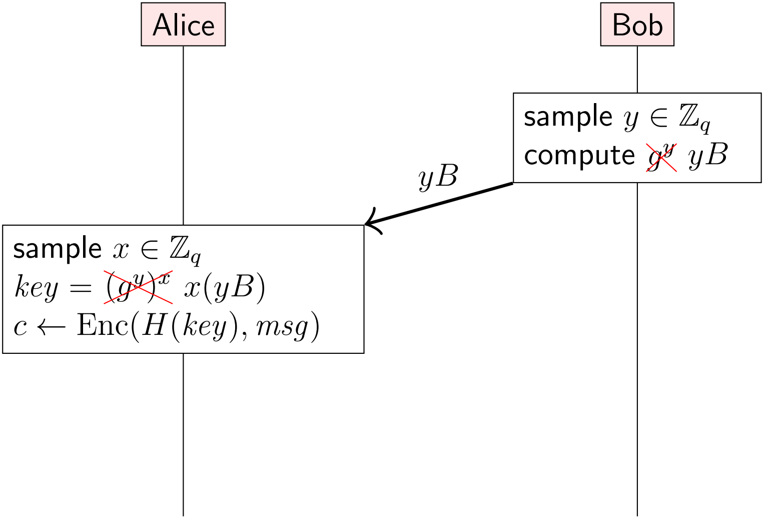
Alice

Bob

sample $y \in \mathbb{Z}_q$
compute $\cancel{g^y}\ yB$

Elliptic Curve Diffie-Hellman (ECDH)

Public parameter: base point $B \in E$ with order q



Alice

Bob

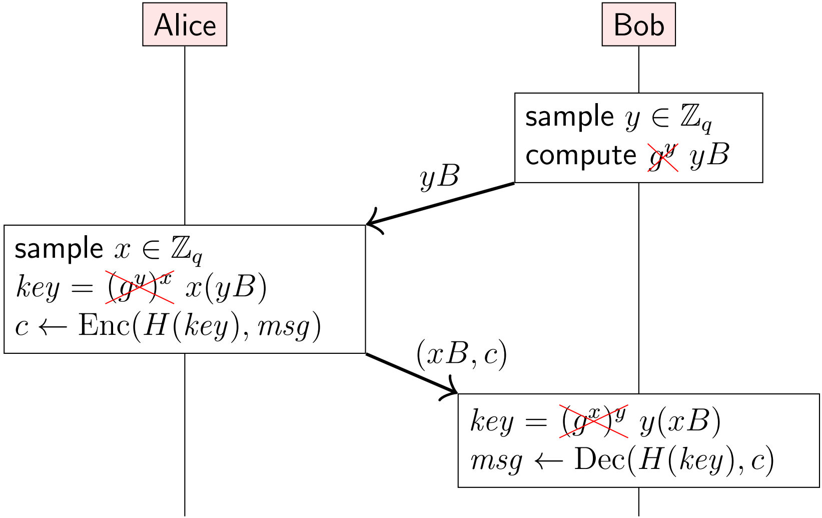
sample $y \in \mathbb{Z}_q$
compute $\cancel{g^y}$ yB

yB

sample $x \in \mathbb{Z}_q$
$key = \cancel{(g^y)^x}\ x(yB)$
$c \leftarrow \mathrm{Enc}(H(key), msg)$

Elliptic Curve Diffie-Hellman (ECDH)

Public parameter: base point $B \in E$ with order q



Alice

Bob

sample $y \in \mathbb{Z}_q$
compute $\xcancel{g^y}\ yB$

yB

sample $x \in \mathbb{Z}_q$
$key = \xcancel{(g^y)^x}\ x(yB)$
$c \leftarrow \mathrm{Enc}(H(key), msg)$

(xB, c)

$key = \xcancel{(g^x)^y}\ y(xB)$
$msg \leftarrow \mathrm{Dec}(H(key), c)$

X25519: Diffie-Hellman using Curve25519

One of the supported EC groups in TLS 1.3.

- $\mathrm{X25519}(sk_A, base) = pk_A$
- $\mathrm{X25519}(sk_A, pk_B) = \mathrm{X25519}(sk_B, pk_A)$

X25519: Diffie-Hellman using Curve25519

One of the supported EC groups in TLS 1.3.

- $\text{X25519}(sk_A, base) = pk_A$
- $\text{X25519}(sk_A, pk_B) = \text{X25519}(sk_B, pk_A)$
- Private keys are 32 bytes, public keys also 32 bytes
- Designed to have simple constant-time implementation
- Fast: only computes x coordinate, not y coordinate
- Allows use of *Montgomery ladder* instead of group law

X25519: Diffie-Hellman using Curve25519

One of the supported EC groups in TLS 1.3.

- $\mathrm{X25519}(sk_A, base) = pk_A$
- $\mathrm{X25519}(sk_A, pk_B) = \mathrm{X25519}(sk_B, pk_A)$
- Private keys are 32 bytes, public keys also 32 bytes
- Designed to have simple constant-time implementation
- Fast: only computes x coordinate, not y coordinate
- Allows use of *Montgomery ladder* instead of group law
- No need to validate whether byte string is a valid curve point (which other protocols require)
- Secure even though underlying curve is not prime-order

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 $\geq 2^{254}$)

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

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 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

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

Assignment 1, Task 1

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

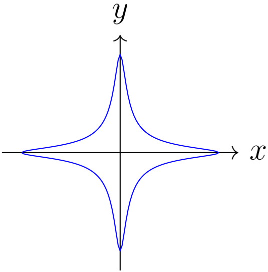
Do it two ways and check they agree:

- 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.

Hope you have fun!

Twisted Edwards Curves



$x^2 + y^2 = 1 - 300x^2y^2$

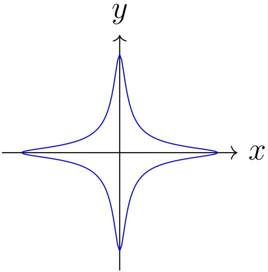
plotted over $\mathbb{R}$

Edwards25519 is the twisted Edwards curve

$$-x^2 + y^2 = 1 + dx^2y^2$$

with $x, y \in \mathbb{F}_p$, $p = 2^{255} - 19$, and $d = -\frac{121665}{121666}$.

Twisted Edwards Curves



$x^2 + y^2 = 1 - 300x^2y^2$

plotted over $\mathbb{R}$

Edwards25519 is the twisted Edwards curve

$$-x^2 + y^2 = 1 + dx^2y^2$$

with $x, y \in \mathbb{F}_p$, $p = 2^{255} - 19$, and $d = -\frac{121665}{121666}$.

There is a 1:1 mapping ("birational equivalence") between points on this curve and Curve25519.

Advantage over elliptic curve: the group law is simpler ⇒ faster to compute (with same security properties).

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)

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.

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.

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 ($32 \rightarrow 64$ bytes).

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").

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.)

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.

Point decompression

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

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

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 $p = 2^{255} - 19$:

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

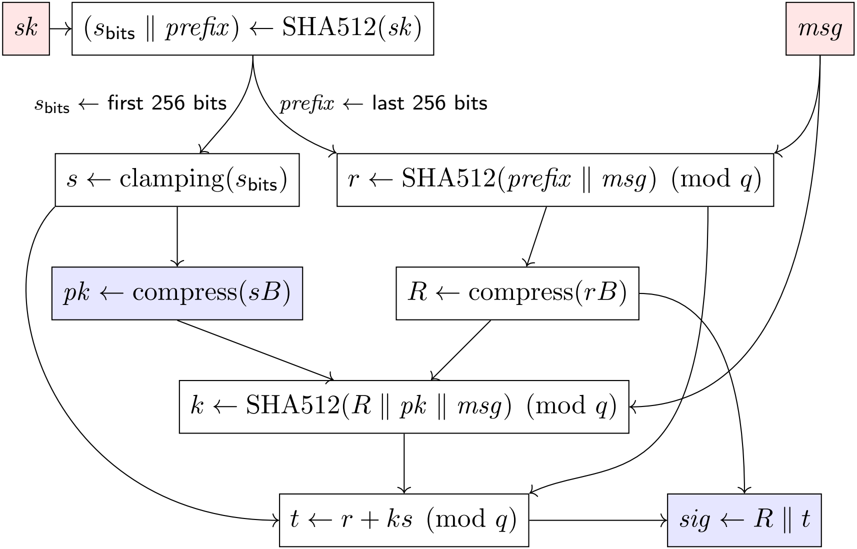
See RFC 8032, Tonelli–Shanks algorithm.

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

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 $\Rightarrow$ 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)

Ed25519 signing



sk

$(s_{\text{bits}} \parallel \mathit{prefix}) \leftarrow \text{SHA512}(sk)$

msg

$s_{\text{bits}} \leftarrow$ first 256 bits

$\mathit{prefix} \leftarrow$ last 256 bits

$s \leftarrow \text{clamping}(s_{\text{bits}})$

$r \leftarrow \text{SHA512}(\mathit{prefix} \parallel msg) \pmod q$

$pk \leftarrow \text{compress}(sB)$

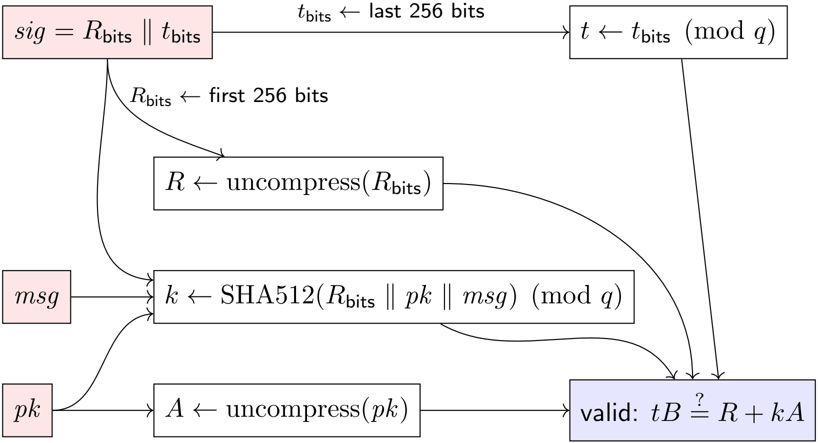
$R \leftarrow \text{compress}(rB)$

$k \leftarrow \text{SHA512}(R \parallel pk \parallel msg) \pmod q$

$t \leftarrow r + ks \pmod q$

$sig \leftarrow R \parallel t$

Ed25519 signature verification



$sig = R_{\text{bits}} \parallel t_{\text{bits}}$

$t_{\text{bits}} \leftarrow$ last 256 bits

$t \leftarrow t_{\text{bits}} \pmod q$

$R_{\text{bits}} \leftarrow$ first 256 bits

$R \leftarrow \mathrm{uncompress}(R_{\text{bits}})$

msg

$k \leftarrow \mathrm{SHA512}(R_{\text{bits}} \parallel pk \parallel msg) \pmod q$

pk

$A \leftarrow \mathrm{uncompress}(pk)$

valid: $tB \stackrel{?}{=} R + kA$

Assignment 1, Task 2

Implement Ed25519 from scratch, relying only on bignums for field arithmetic.

You can find example code and test vectors in RFC 8032.

Due date for code and lab report: ~~10~~ 17 Feb 2025

Software Engineering for Cryptography

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

University of Cambridge
Part III/MPhil in Advanced Computer Science

Why Do We Have Standards?

- Standards are important for interoperability
  - Different implementations need to work together
  - Writing it in text requires that everything is specified

Why Do We Have Standards?

- Standards are important for interoperability
  - Different implementations need to work together
  - Writing it in text requires that everything is specified
- Standards are important for security
  - Well-reviewed specifications
  - Clear security requirements and properties

Major Standardization Bodies

IETF typically protocols

- TLS 1.3 (RFC 8446)
- COSE (RFC 8152)

NIST typically algorithms

- AES (FIPS 197)
- SHA-3 (FIPS 202)

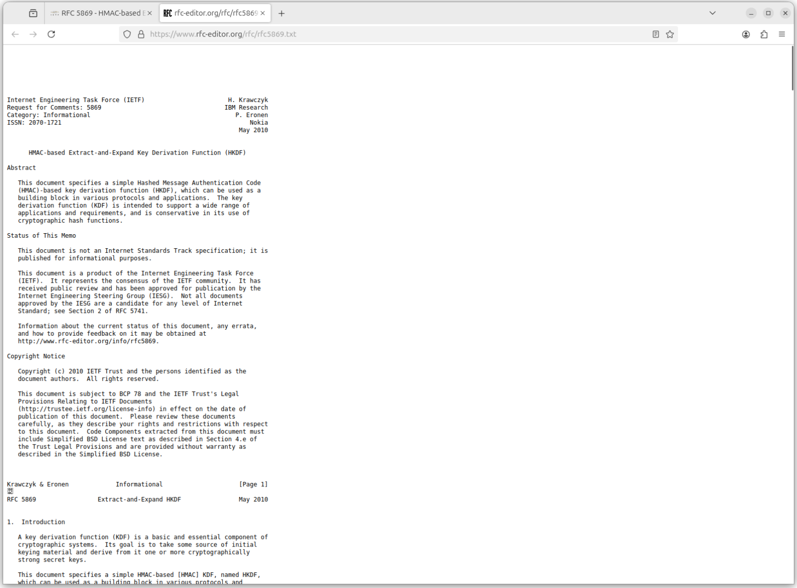
IEEE typically hardware/protocols

- IEEE 802.11i (WPA2/WPA3)

Reading RFCs

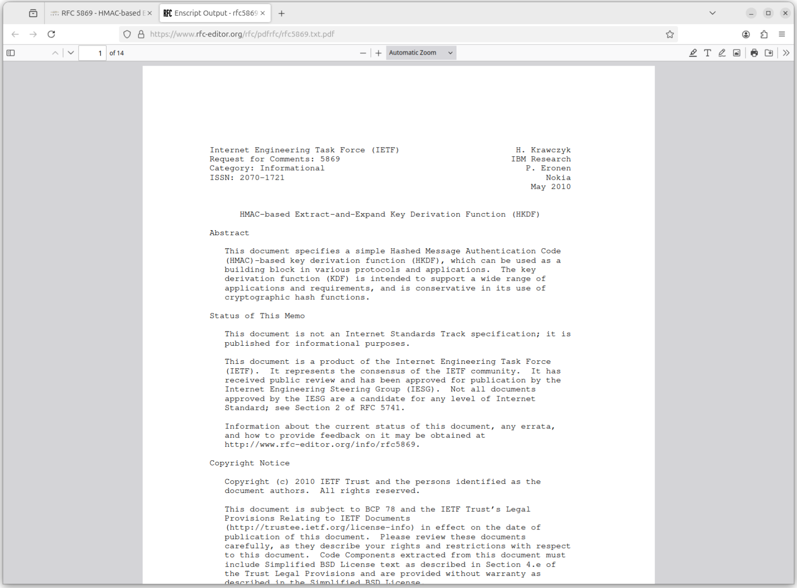
- Example: RFC 5869 - HMAC-based Extract-and-Expand Key Derivation Function (HKDF)
- Let's walk through how to read and understand an RFC
- They are available at https://datatracker.ietf.org/doc/html/rfc5869

RFC Formats - Text



```
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.

Status of This Memo

   This document is not an Internet Standards Track specification; it is
   published for informational purposes.

   This document is a product of the Internet Engineering Task Force
   (IETF).  It represents the consensus of the IETF community.  It has
   received public review and has been approved for publication by the
   Internet Engineering Steering Group (IESG).  Not all documents
   approved by the IESG are a candidate for any level of Internet
   Standard; see Section 2 of RFC 5741.

   Information about the current status of this document, any errata,
   and how to provide feedback on it may be obtained at
   http://www.rfc-editor.org/info/rfc5869.

Copyright Notice

   Copyright (c) 2010 IETF Trust and the persons identified as the
   document authors.  All rights reserved.

   This document is subject to BCP 78 and the IETF Trust's Legal
   Provisions Relating to IETF Documents
   (http://trustee.ietf.org/license-info) in effect on the date of
   publication of this document.  Please review these documents
   carefully, as they describe your rights and restrictions with respect
   to this document.  Code Components extracted from this document must
   include Simplified BSD License text as described in Section 4.e of
   the Trust Legal Provisions and are provided without warranty as
   described in the Simplified BSD License.



Krawczyk & Eronen             Informational                     [Page 1]

RFC 5869                Extract-and-Expand HKDF                 May 2010


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
```

RFC Formats - PDF



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.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

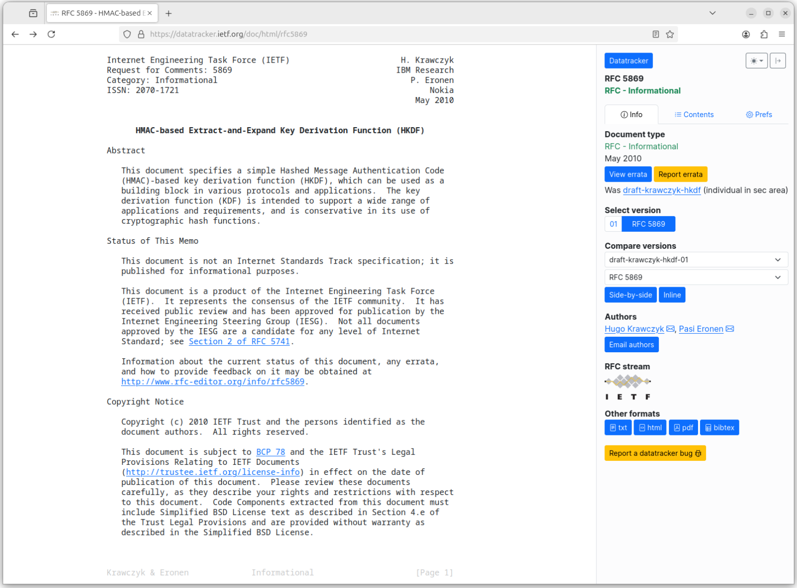
Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc5869.

Copyright Notice

Copyright (c) 2010 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

RFC Formats - HTML



Internet Engineering Task Force (IETF)
Request for Comments: 5869
Category: Informational
ISSN: 2070-1721

H. Krawczyk
IBM Research
P. Eronen
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.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc5869.

Copyright Notice

Copyright (c) 2010 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Krawczyk & Eronen Informational [Page 1]

RFC 5869
RFC - Informational

Document type
RFC - Informational
May 2010

Was draft-krawczyk-hkdf (individual in sec area)

Select version

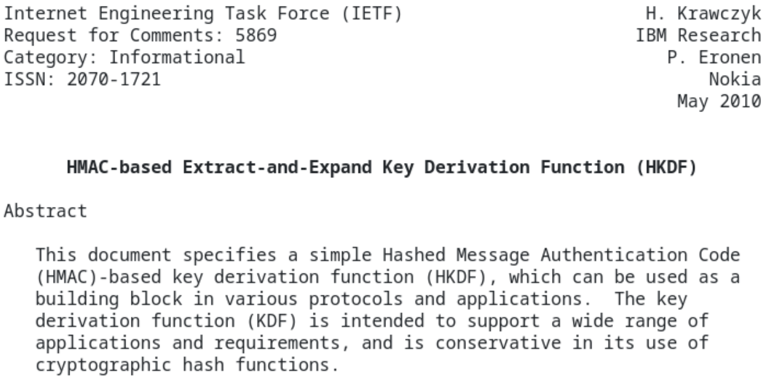
Compare versions
draft-krawczyk-hkdf-01
RFC 5869

Authors
Hugo Krawczyk, Pasi Eronen

RFC stream

Other formats

RFC Header (RFC 5869)



Internet Engineering Task Force (IETF)
Request for Comments: 5869
Category: Informational
ISSN: 2070-1721

H. Krawczyk
IBM Research
P. Eronen
Nokia
May 2010

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

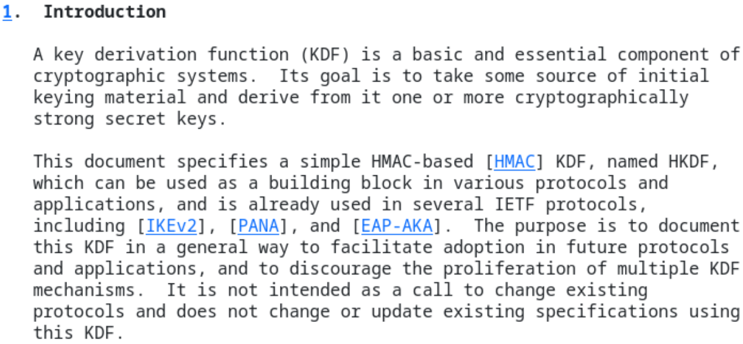
Abstract

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.

RFC Categories

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

Introduction (RFC 5869)

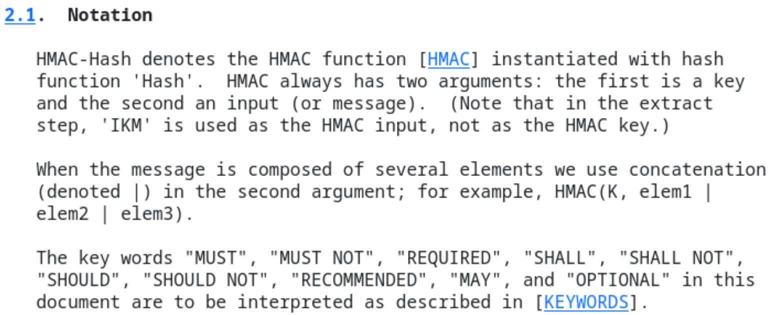


1. Introduction

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.

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].

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)

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.”*

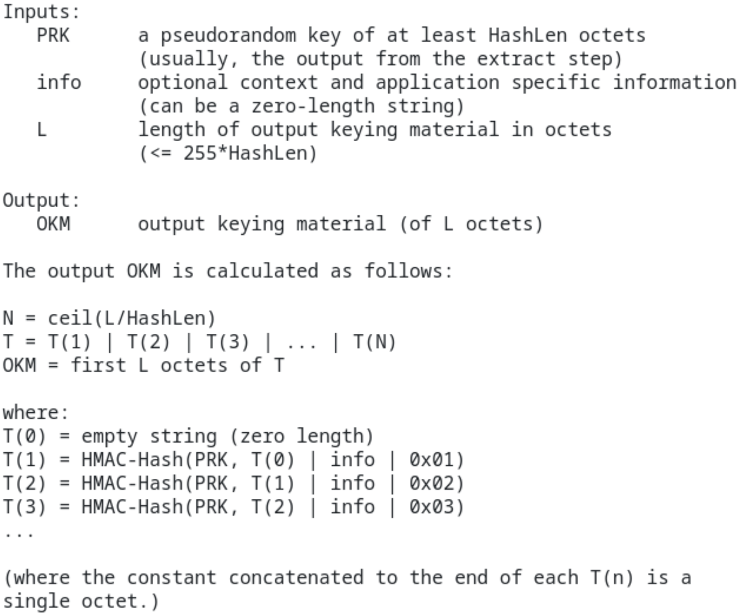
Notation

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."*

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.)
```

Algorithm Section (RFC 5869)

- Contains detailed technical specification
- Describes the protocol/algorithm step by step

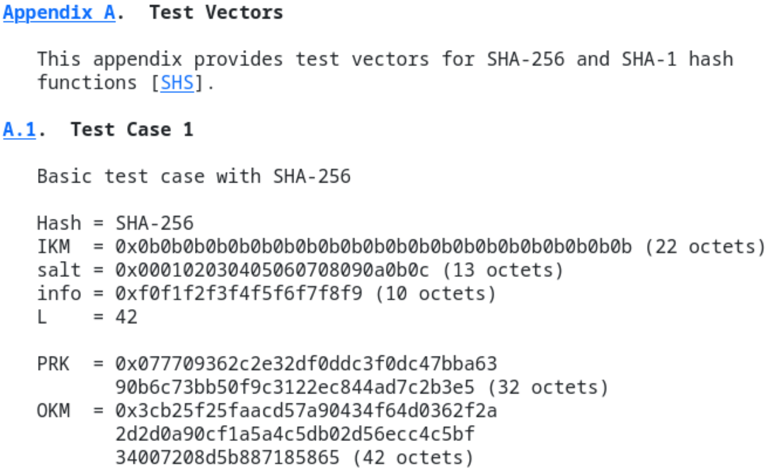
Algorithm Section (RFC 5869)

- 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

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

Test Vectors



Appendix A. Test Vectors

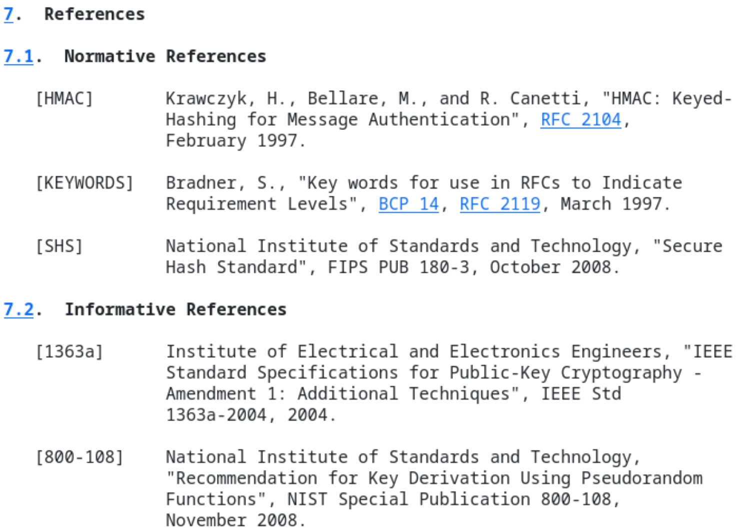
This appendix provides test vectors for SHA-256 and SHA-1 hash functions [SHS].

A.1. Test Case 1

Basic test case with SHA-256

```
Hash = SHA-256
IKM  = 0x0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b (22 octets)
salt = 0x000102030405060708090a0b0c (13 octets)
info = 0xf0f1f2f3f4f5f6f7f8f9 (10 octets)
L    = 42

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

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.

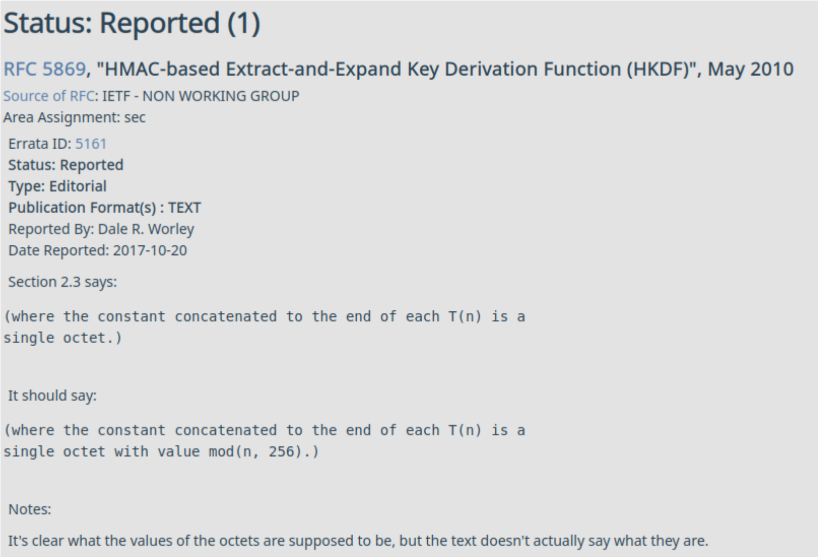
[SHS] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-3, October 2008.

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.

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.

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

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

NIST Post-Quantum Cryptography Standardization

- Started in 2016 to standardize quantum-resistant cryptographic algorithms
- Goal: Protect against both classical and quantum computer attacks

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

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

Case Study: Dual_EC_DRBG

- NIST SP 800-90A standardized Dual_EC_DRBG in 2006

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

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

Case Study: Choice of p in X25519

> *"I chose my prime* $2^{255} - 19$ *according to the following criteria: primes as close as possible to a power of 2 save time in field operations (as in, e.g, [9]), with no effect on (conjectured) security level; primes slightly below* $32k$ *bits, for some* k*, allow public keys to be easily transmitted in 32-bit words, with no serious concerns regarding wasted space;* $k = 8$ *provides a comfortable security level. I considered the primes* $2^{255} + 95$, $2^{255} - 19$, $2^{255} - 31$, $2^{254} + 79$, $2^{253} + 51$, *and* $2^{253} + 39$, *and selected* $2^{255} - 19$ *because 19 is smaller than 31, 39, 51, 79, 95"*

Bernstein 2006, p. 13

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 $\{4 \cdot$ prime$, 8 \cdot$ 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

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)

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.

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
);
```

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
}
```

C-style error handling

However, it's error prone:

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

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
}
```

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
}
```

C-style error handling: lessons learned

- Making error checking optional is dangerous

C-style error handling: lessons learned

- Making error checking optional is dangerous
- Relying on humans to follow patterns is infeasible

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. . .

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.

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
    }
}
```

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
}
```

C-style error handling: writing better call sites

```c
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
}
```

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;
}
```

Exception-based error handling

Declared as such:

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

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
    }
}
```

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
    }
}
```

Exception-based error handling

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

Exception-based error handling

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

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?
    }
}
```

Result types for error handling

Declared as such:

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

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"))
    }
}
```

Result types for error handling (ergonomics)

Using `let-else` for error handling:

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

Result types for error handling (ergonomics)

Using `let-else` for error handling:

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

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?

Error messages: be helpful

- `decryption failed`
  - No information about what went wrong
  - No guidance on how to fix it

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

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

Error messages: but not too helpful

```
decryption failed:  key must be 16 or 32 bytes,
got value "secretkey" which is 9 bytes long
```

Error messages: but not too helpful

```
decryption failed:  key must be 16 or 32 bytes,
got value "secretkey" which is 9 bytes long
```

- This leads us to our next topic: **leaky implementations**

Leaky implementations

Idealised formal world:

- Operations are solely defined by their mathematical properties
- Perfect black boxes
- Often assuming infinite resources

Leaky implementations

Idealised formal world:

- Operations are solely defined by their mathematical properties
- Perfect black boxes
- Often assuming infinite resources

Real world:

- Implementations are not perfect
- Adversary can learn exact hardware/software steps
- Work against a finite set of resources

Side-channels

- Timing side-channels
- Error messages
- Memory access patterns
- Energy consumption
- Electromagnetic emissions
- ...

Every bit of information can be used by an adversary. Especially if accessible in a repeated manner with adversary-controlled input.

Timing side-channels

- If the execution time of an algorithm depends on a secret value, measuring the time may leak that value, e.g. private key (**timing side-channel**)

Timing side-channels

- If the execution time of an algorithm depends on a secret value, measuring the time may leak that value, e.g. private key (**timing side-channel**)
- Memory access patterns can be revealed by timing (cache hits/misses)

Timing side-channels

- If the execution time of an algorithm depends on a secret value, measuring the time may leak that value, e.g. private key (**timing side-channel**)
- Memory access patterns can be revealed by timing (cache hits/misses)

Implementations should be **constant-time** to avoid this:

- No branches (if/else, break, . . .) conditional on a secret

Timing side-channels

- If the execution time of an algorithm depends on a secret value, measuring the time may leak that value, e.g. private key (**timing side-channel**)
- Memory access patterns can be revealed by timing (cache hits/misses)

Implementations should be **constant-time** to avoid this:

- No branches (if/else, break, . . .) conditional on a secret
- No memory access (array lookups) dependent on a secret

Timing side-channels

- If the execution time of an algorithm depends on a secret value, measuring the time may leak that value, e.g. private key (**timing side-channel**)
- Memory access patterns can be revealed by timing (cache hits/misses)

Implementations should be **constant-time** to avoid this:

- No branches (if/else, break, . . .) conditional on a secret
- No memory access (array lookups) dependent on a secret
- Individual CPU instructions (e.g. multiplying two 32-bit numbers) typically assumed to be constant-time

Timing side-channels

- If the execution time of an algorithm depends on a secret value, measuring the time may leak that value, e.g. private key (**timing side-channel**)
- Memory access patterns can be revealed by timing (cache hits/misses)

Implementations should be **constant-time** to avoid this:

- No branches (if/else, break, . . .) conditional on a secret
- No memory access (array lookups) dependent on a secret
- Individual CPU instructions (e.g. multiplying two 32-bit numbers) typically assumed to be constant-time

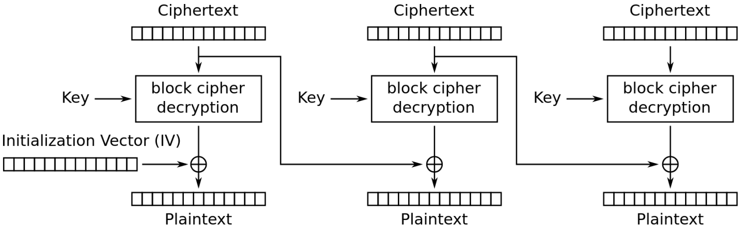
Code you write in this module **need not** be constant-time. However, your lab report should discuss how you could make it constant-time.

Padding oracle attack

- Assume AES-CBC with PKCS#7 padding
  - Padding pattern: `?? ?? ?? ?? 04 04 04 04`

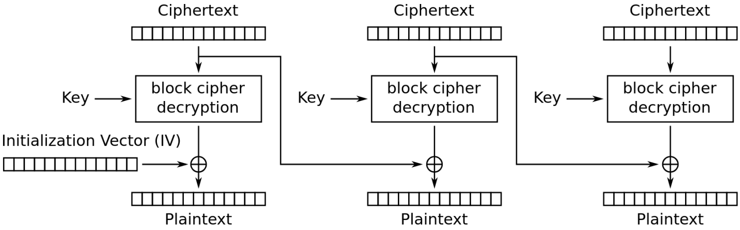
Padding oracle attack

- 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 $C_0 = IV$



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 $C_0 = IV$

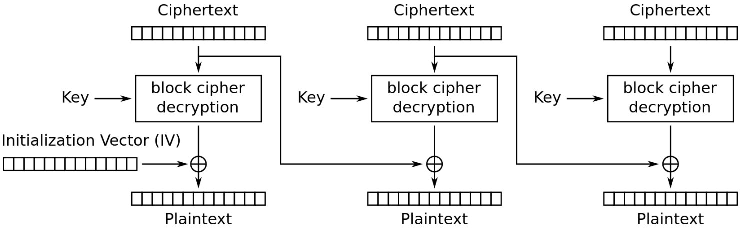


- Server returns either *nothing* or `PADDING_ERR`

Figure: Wikipedia, public domain.

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.

Padding oracle attacks in the real-world

- Initial attack in 1998 by Bleichenbacher against RSA → fixed

Padding oracle attacks in the real-world

- Initial attack in 1998 by Bleichenbacher against RSA → fixed
- More efficient attack against CBC-mode AES in 2002 by Vaudenay → fixed

Padding oracle attacks in the real-world

- Initial attack in 1998 by Bleichenbacher against RSA → fixed
- More efficient attack against CBC-mode AES in 2002 by Vaudenay → fixed
- Lucky Thirteen attack against TLS in 2013 (AlFardan and Paterson) by using the same technique but with a timing side-channel → fixed

Padding oracle attacks in the real-world

- Initial attack in 1998 by Bleichenbacher against RSA → fixed
- More efficient attack against CBC-mode AES in 2002 by Vaudenay → fixed
- Lucky Thirteen attack against TLS in 2013 (AlFardan and Paterson) by using the same technique but with a timing side-channel → fixed
- This introduced another timing side-channel CVE-2016-2107 → fixed

Padding oracle attacks in the real-world

- Initial attack in 1998 by Bleichenbacher against RSA → fixed
- More efficient attack against CBC-mode AES in 2002 by Vaudenay → fixed
- Lucky Thirteen attack against TLS in 2013 (AlFardan and Paterson) by using the same technique but with a timing side-channel → fixed
- This introduced another timing side-channel CVE-2016-2107 → fixed
- to be continued...

Not leaking information in Internet-connected services really is a **hard problem**!

What is a secret key actually?

Let's return to our X25519 secret key: $x \in \mathbb{F}_p$. But what exactly *is* x in the real world?

What is a secret key actually?

Let's return to our X25519 secret key: $x \in \mathbb{F}_p$. But what exactly *is* x in the real world?

- A number in the set $\mathbb{F}_p$?

What is a secret key actually?

Let's return to our X25519 secret key: $x \in \mathbb{F}_p$. But what exactly *is* x in the real world?

- A number in the set $\mathbb{F}_p$?
- The binary representation of that number?

What is a secret key actually?

Let's return to our X25519 secret key: $x \in \mathbb{F}_p$. But what exactly *is* x in the real world?

- A number in the set $\mathbb{F}_p$?
- The binary representation of that number?
- A Python object?

What is a secret key actually?

Let's return to our X25519 secret key: $x \in \mathbb{F}_p$. But what exactly *is* x in the real world?

- A number in the set $\mathbb{F}_p$?
- The binary representation of that number?
- A Python object?
- The serialized bytes?

What is a secret key actually?

Let's return to our X25519 secret key: $x \in \mathbb{F}_p$. But what exactly *is* x in the real world?

- A number in the set $\mathbb{F}_p$?
- The binary representation of that number?
- A Python object?
- The serialized bytes?
- Also the logic to interpret the bytes?
- . . .

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

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]
```

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

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
```

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
```

Going past one type

This comes with challenges:

- Cross-library interfaces require widely accepted types/patterns.
- Imperfect representations of all cryptographic properties.
- At some point, types become too bothersome
  - Developers go back to raw `byte[]` arrays

Going past one type

This comes with challenges:

- Cross-library interfaces require widely accepted types/patterns.
- Imperfect representations of all cryptographic properties.
- At some point, types become too bothersome
  - Developers go back to raw `byte[]` arrays

It's not an exact science.

Going past one type

This comes with challenges:

- Cross-library interfaces require widely accepted types/patterns.
- Imperfect representations of all cryptographic properties.
- At some point, types become too bothersome
  - Developers go back to raw `byte[]` arrays

It's not an exact science.
Don't be too clever.

Going past one type

This comes with challenges:

- Cross-library interfaces require widely accepted types/patterns.
- Imperfect representations of all cryptographic properties.
- At some point, types become too bothersome
    - Developers go back to raw `byte[]` arrays

It's not an exact science.
Don't be too clever.
Empathy and pragmatism win.

More complex types

Consider an encrypted AEAD message. It typically contains:

- A nonce or IV
- Associated data
- Encrypted data
- A tag

Strong types help here, as they ensure elements are not accidentally mixed up. But does the callee actually need to know about this? High-level API $\leftrightarrow$ low-level API

Static Type Checking with mypy

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

```
# Install mypy:
pip install mypy

# Run mypy:
mypy your_file.py
mypy your_code/
```

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"])
```

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
```

Advanced Type Features

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

```
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]]
```

Self-referential Types

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

```
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)
```

Pattern: type state (1, motivating example)

```
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(())
    }
}
```

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})
    }
}
```

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}})
    }
}
```

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);
    }
}
```

Pattern: type state (5)

Benefits:

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

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

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

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

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

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

Pattern: type promotions

```rust
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 ...
}
```

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> {
    // ...
}
```

Authenticated key exchange

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

University of Cambridge
Part III/MPhil in Advanced Computer Science

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.

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

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

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.

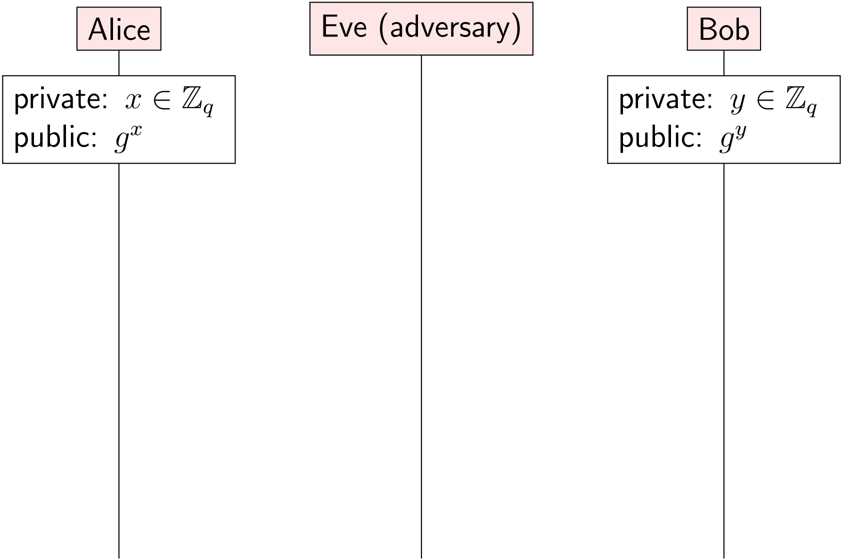
Threat model: assume what each party may do / not do, and what the network may do / not do

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!

Unauthenticated Diffie-Hellman



Alice

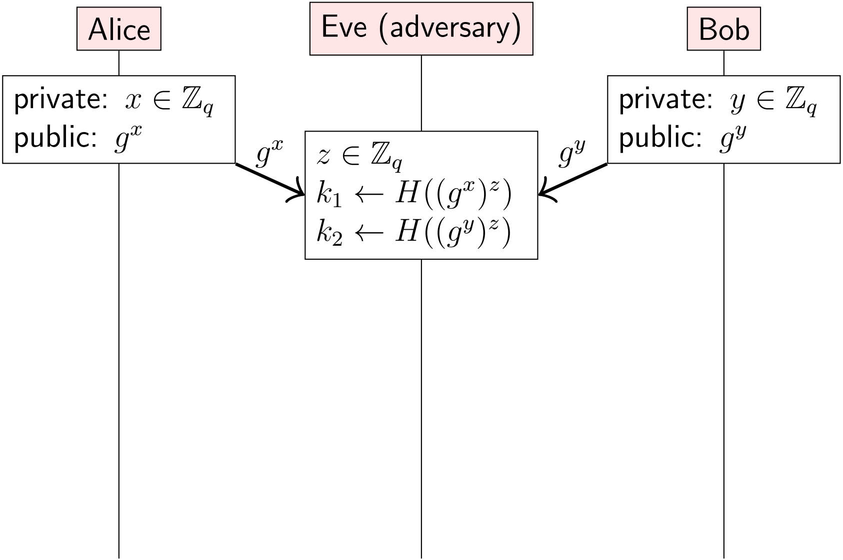
private: $x \in \mathbb{Z}_q$
public: g^x

Eve (adversary)

Bob

private: $y \in \mathbb{Z}_q$
public: g^y

Unauthenticated Diffie-Hellman



Alice

private: $x \in \mathbb{Z}_q$
public: g^x

Eve (adversary)

g^x

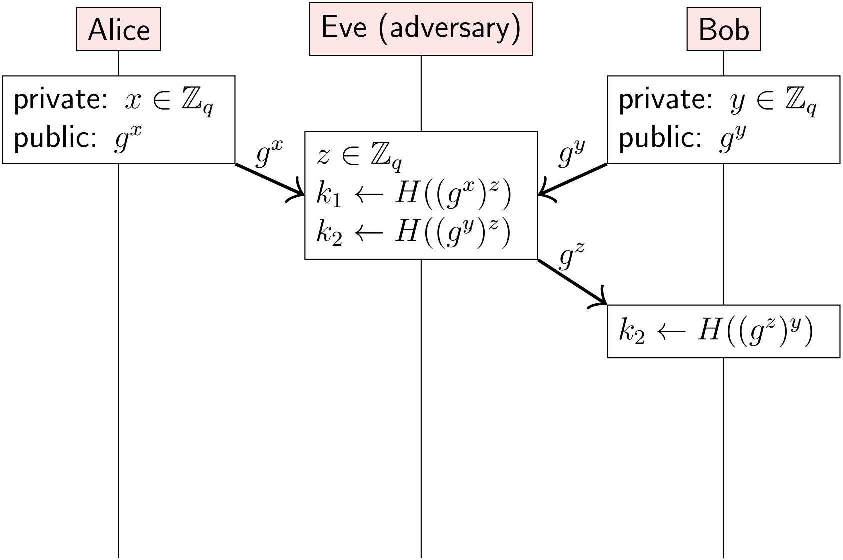
$z \in \mathbb{Z}_q$
$k_1 \leftarrow H((g^x)^z)$
$k_2 \leftarrow H((g^y)^z)$

g^y

Bob

private: $y \in \mathbb{Z}_q$
public: g^y

Unauthenticated Diffie-Hellman



Alice

Eve (adversary)

Bob

private: $x \in \mathbb{Z}_q$
public: g^x

private: $y \in \mathbb{Z}_q$
public: g^y

g^x

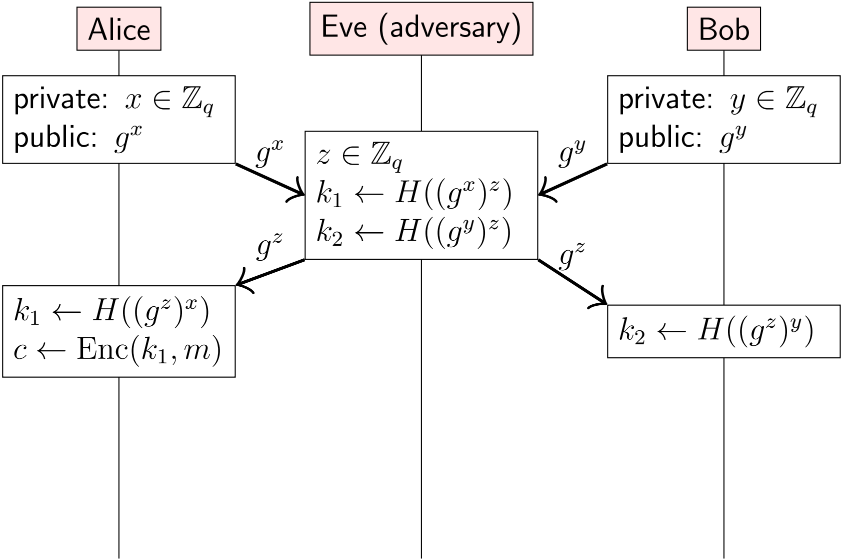
g^y

$z \in \mathbb{Z}_q$
$k_1 \leftarrow H((g^x)^z)$
$k_2 \leftarrow H((g^y)^z)$

g^z

$k_2 \leftarrow H((g^z)^y)$

Unauthenticated Diffie-Hellman



Alice

Eve (adversary)

Bob

private: $x \in \mathbb{Z}_q$
public: g^x

private: $y \in \mathbb{Z}_q$
public: g^y

g^x

g^y

$z \in \mathbb{Z}_q$
$k_1 \leftarrow H((g^x)^z)$
$k_2 \leftarrow H((g^y)^z)$

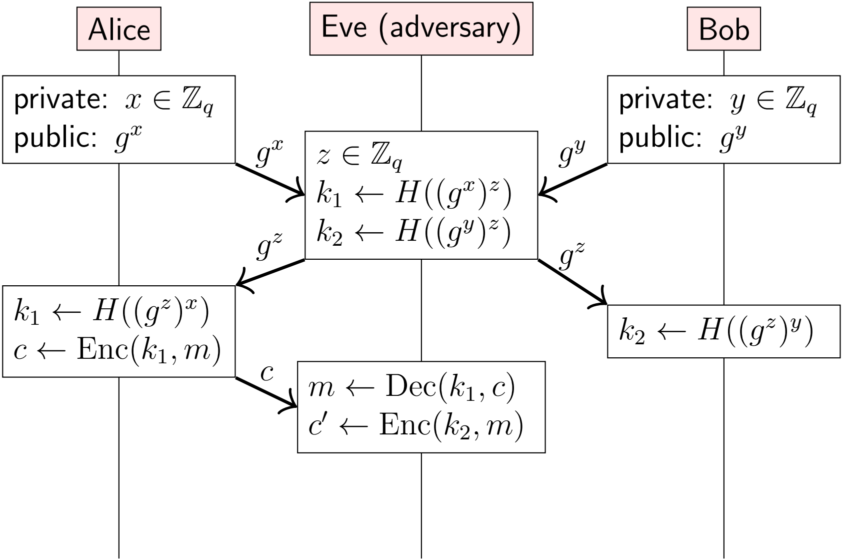
g^z

g^z

$k_1 \leftarrow H((g^z)^x)$
$c \leftarrow \mathrm{Enc}(k_1, m)$

$k_2 \leftarrow H((g^z)^y)$

Unauthenticated Diffie-Hellman



Alice

Eve (adversary)

Bob

private: $x \in \mathbb{Z}_q$
public: g^x

private: $y \in \mathbb{Z}_q$
public: g^y

g^x

g^y

$z \in \mathbb{Z}_q$
$k_1 \leftarrow H((g^x)^z)$
$k_2 \leftarrow H((g^y)^z)$

g^z

g^z

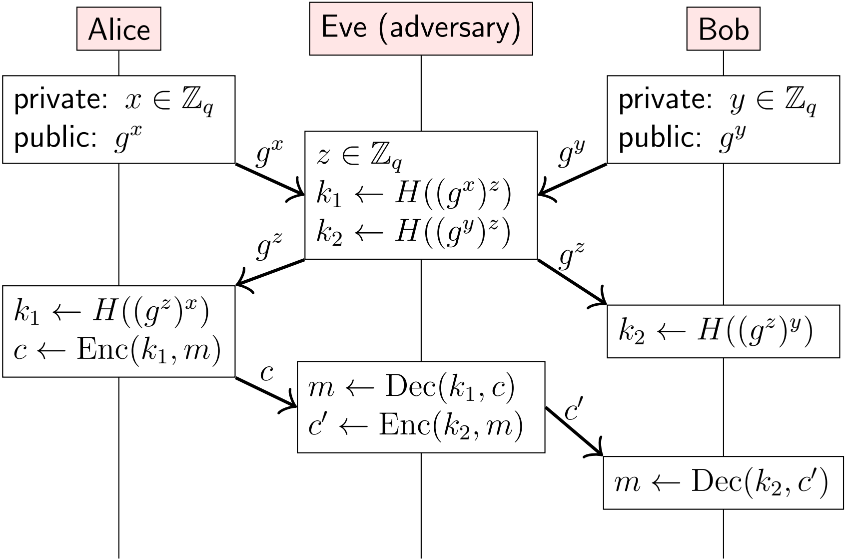
$k_1 \leftarrow H((g^z)^x)$
$c \leftarrow \mathrm{Enc}(k_1, m)$

$k_2 \leftarrow H((g^z)^y)$

c

$m \leftarrow \mathrm{Dec}(k_1, c)$
$c' \leftarrow \mathrm{Enc}(k_2, m)$

Unauthenticated Diffie-Hellman



Alice

Eve (adversary)

Bob

private: $x \in \mathbb{Z}_q$
public: g^x

private: $y \in \mathbb{Z}_q$
public: g^y

g^x

g^y

$z \in \mathbb{Z}_q$
$k_1 \leftarrow H((g^x)^z)$
$k_2 \leftarrow H((g^y)^z)$

g^z

g^z

$k_1 \leftarrow H((g^z)^x)$
$c \leftarrow \mathrm{Enc}(k_1, m)$

$k_2 \leftarrow H((g^z)^y)$

c

$m \leftarrow \mathrm{Dec}(k_1, c)$
$c' \leftarrow \mathrm{Enc}(k_2, m)$

c'

$m \leftarrow \mathrm{Dec}(k_2, c')$

Mutual authentication in protocols

How do two communicating parties convince each other that they are genuine?

Mutual authentication in protocols

How do two communicating parties convince each other that they are genuine?

Two usual forms of mutual authentication:

- **Authenticated Key Exchange** (AKE):
  each party has a private key; other party knows the corresponding public key

Mutual authentication in protocols

How do two communicating parties convince each other that they are genuine?

Two usual forms of mutual authentication:

- **Authenticated Key Exchange** (AKE):
  each party has a private key; other party knows the corresponding public key
- **Password-Authenticated Key Exchange** (PAKE):
  the two parties share a (low-entropy) password; prove they know it without revealing it (e.g. wifi password)

Mutual authentication in protocols

How do two communicating parties convince each other that they are genuine?

Two usual forms of mutual authentication:

- **Authenticated Key Exchange** (AKE):
  each party has a private key; other party knows the corresponding public key
- **Password-Authenticated Key Exchange** (PAKE):
  the two parties share a (low-entropy) password; prove they know it without revealing it (e.g. wifi password)

On the web it's usually asymmetric:

1. **Server authenticates itself to client** using public key

Mutual authentication in protocols

How do two communicating parties convince each other that they are genuine?

Two usual forms of mutual authentication:

- **Authenticated Key Exchange** (AKE):
  each party has a private key; other party knows the corresponding public key
- **Password-Authenticated Key Exchange** (PAKE):
  the two parties share a (low-entropy) password; prove they know it without revealing it (e.g. wifi password)

On the web it's usually asymmetric:

1. **Server authenticates itself to client** using public key
2. **Client authenticates itself to server** by sending password over encrypted+authenticated connection

Public Key Infrastructure (PKI)

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

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

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

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

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

Public Key Infrastructure (PKI)

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

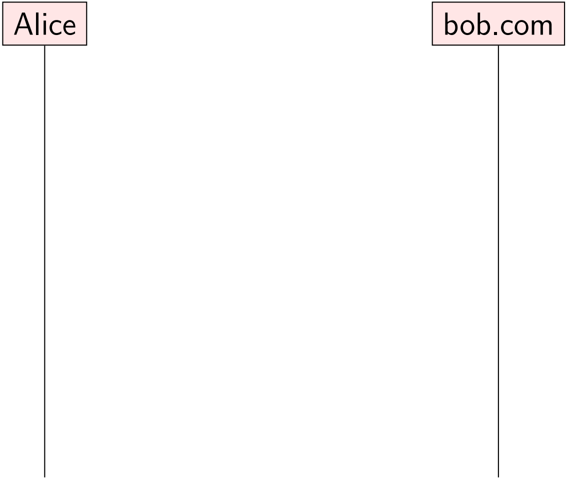
- **Web** (WebPKI):
  - Certificate is (domain name, pubkey, validity dates)
  - Certificate is signed by a **certificate authority** (CA)
  - Several CAs' public keys are built into the web browser
  - New website: CA checks via HTTP request or DNS whether you own the domain
- **Secure messaging** (Signal, 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)

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_{CA}



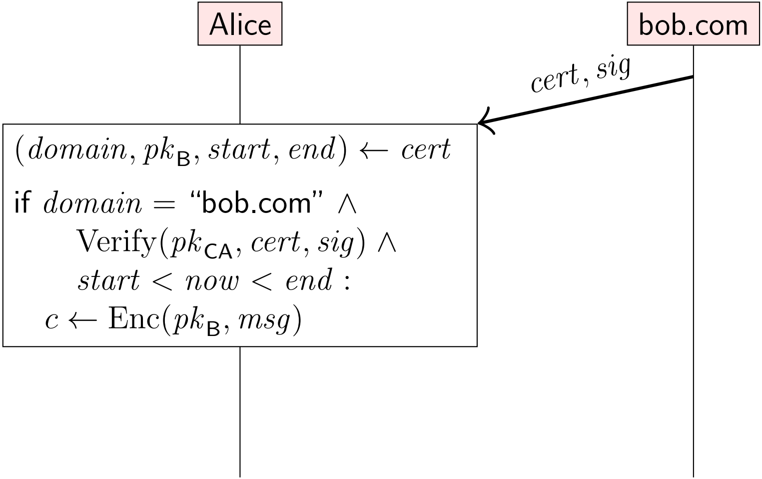
Alice

bob.com

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_{CA}

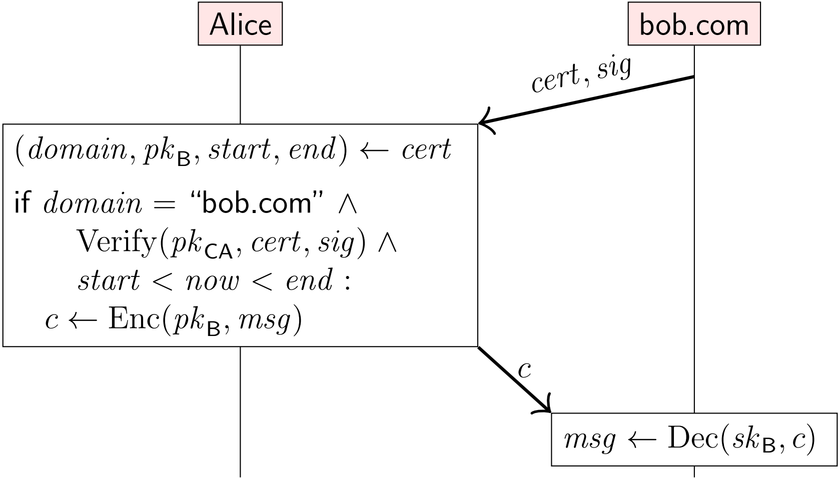


```
$(domain, pk_{B}, start, end) \leftarrow cert$
if $domain = \text{“bob.com”} \wedge$
        $\mathrm{Verify}(pk_{CA}, cert, sig) \wedge$
        $start < now < end$ :
    $c \leftarrow \mathrm{Enc}(pk_{B}, msg)$
```

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_{CA}



Alice

bob.com

$cert, sig$

```
$(domain, pk_{\mathsf{B}}, start, end) \leftarrow cert$
if $domain = \text{“bob.com”} \wedge$
        $\mathrm{Verify}(pk_{\mathsf{CA}}, cert, sig) \wedge$
        $start < now < end$ :
    $c \leftarrow \mathrm{Enc}(pk_{\mathsf{B}}, msg)$
```

c

$msg \leftarrow \mathrm{Dec}(sk_{\mathsf{B}}, c)$

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

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

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

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)

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!

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

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:

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

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

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**

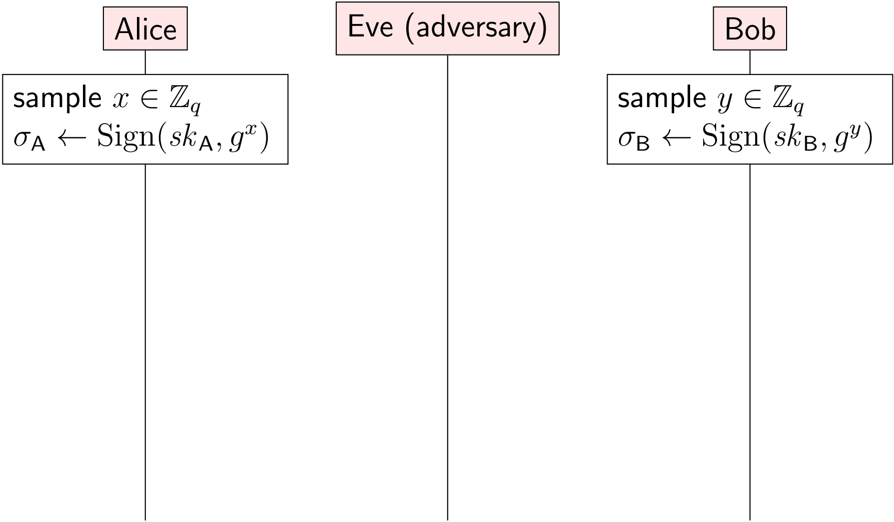
Requirements for authenticated key exchange

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)

Badly Authenticated Diffie-Hellman (1)



Alice

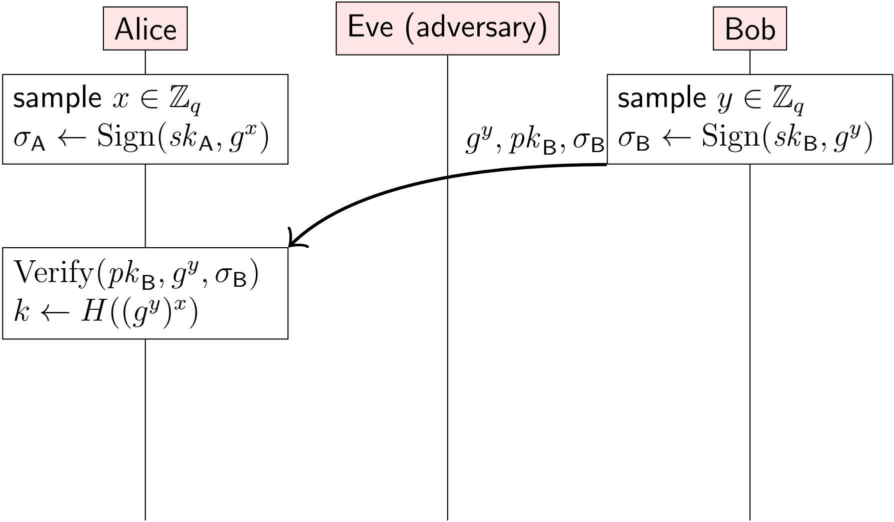
Eve (adversary)

Bob

sample $x \in \mathbb{Z}_q$
$\sigma_{\mathsf{A}} \leftarrow \mathrm{Sign}(sk_{\mathsf{A}}, g^x)$

sample $y \in \mathbb{Z}_q$
$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^y)$

Badly Authenticated Diffie-Hellman (1)



Alice

Eve (adversary)

Bob

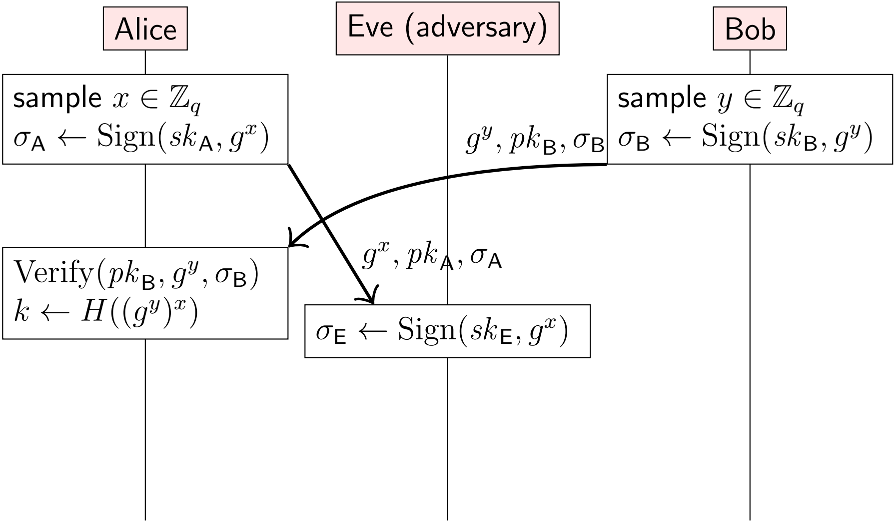
sample $x \in \mathbb{Z}_q$
$\sigma_\mathsf{A} \leftarrow \mathrm{Sign}(sk_\mathsf{A}, g^x)$

sample $y \in \mathbb{Z}_q$
$\sigma_\mathsf{B} \leftarrow \mathrm{Sign}(sk_\mathsf{B}, g^y)$

$g^y, pk_\mathsf{B}, \sigma_\mathsf{B}$

$\mathrm{Verify}(pk_\mathsf{B}, g^y, \sigma_\mathsf{B})$
$k \leftarrow H((g^y)^x)$

Badly Authenticated Diffie-Hellman (1)



Alice

Eve (adversary)

Bob

sample $x \in \mathbb{Z}_q$
$\sigma_{\mathsf{A}} \leftarrow \mathrm{Sign}(sk_{\mathsf{A}}, g^x)$

sample $y \in \mathbb{Z}_q$
$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^y)$

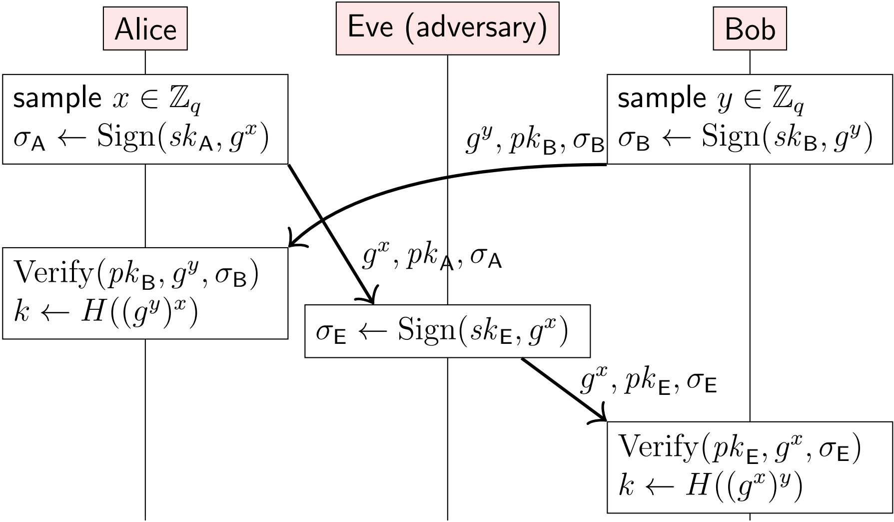
$g^y, pk_{\mathsf{B}}, \sigma_{\mathsf{B}}$

$g^x, pk_{\mathsf{A}}, \sigma_{\mathsf{A}}$

$\mathrm{Verify}(pk_{\mathsf{B}}, g^y, \sigma_{\mathsf{B}})$
$k \leftarrow H((g^y)^x)$

$\sigma_{\mathsf{E}} \leftarrow \mathrm{Sign}(sk_{\mathsf{E}}, g^x)$

Badly Authenticated Diffie-Hellman (1)



Alice

Eve (adversary)

Bob

sample $x \in \mathbb{Z}_q$
$\sigma_{\mathsf{A}} \leftarrow \mathrm{Sign}(sk_{\mathsf{A}}, g^x)$

sample $y \in \mathbb{Z}_q$
$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^y)$

$g^y, pk_{\mathsf{B}}, \sigma_{\mathsf{B}}$

$g^x, pk_{\mathsf{A}}, \sigma_{\mathsf{A}}$

$\mathrm{Verify}(pk_{\mathsf{B}}, g^y, \sigma_{\mathsf{B}})$
$k \leftarrow H((g^y)^x)$

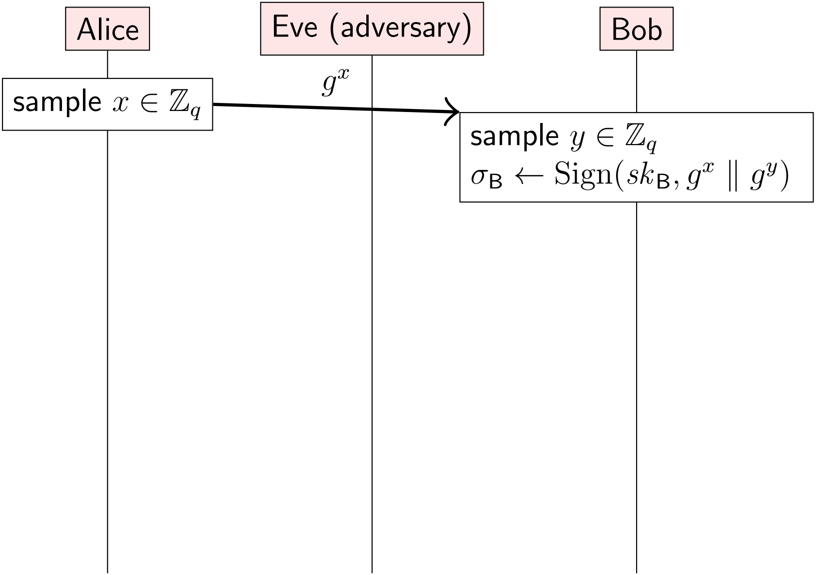
$\sigma_{\mathsf{E}} \leftarrow \mathrm{Sign}(sk_{\mathsf{E}}, g^x)$

$g^x, pk_{\mathsf{E}}, \sigma_{\mathsf{E}}$

$\mathrm{Verify}(pk_{\mathsf{E}}, g^x, \sigma_{\mathsf{E}})$
$k \leftarrow H((g^x)^y)$

Alice is talking to Bob, but Bob thinks he's talking to Eve

Badly Authenticated Diffie-Hellman (2)



Alice

Eve (adversary)

Bob

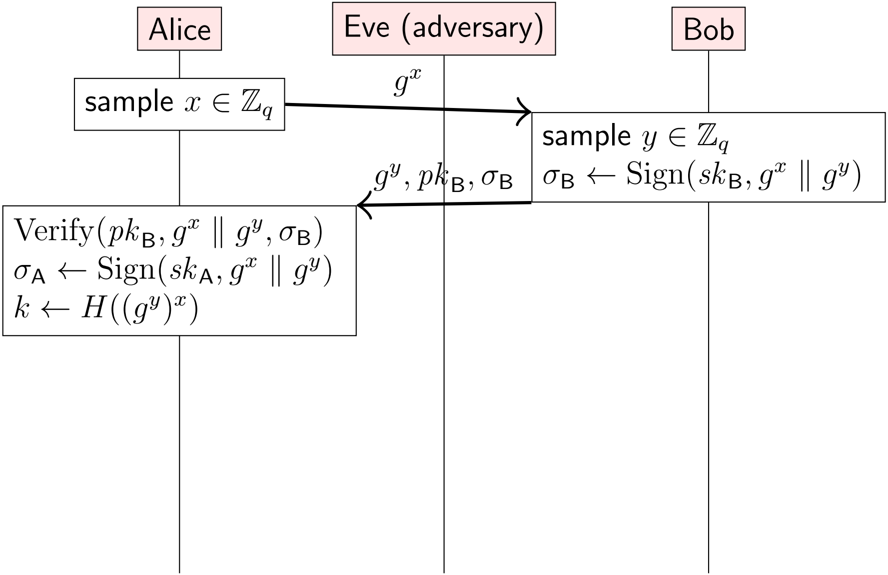
sample $x \in \mathbb{Z}_q$

g^x

sample $y \in \mathbb{Z}_q$

$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^x \parallel g^y)$

Badly Authenticated Diffie-Hellman (2)



Alice

Eve (adversary)

Bob

sample $x \in \mathbb{Z}_q$

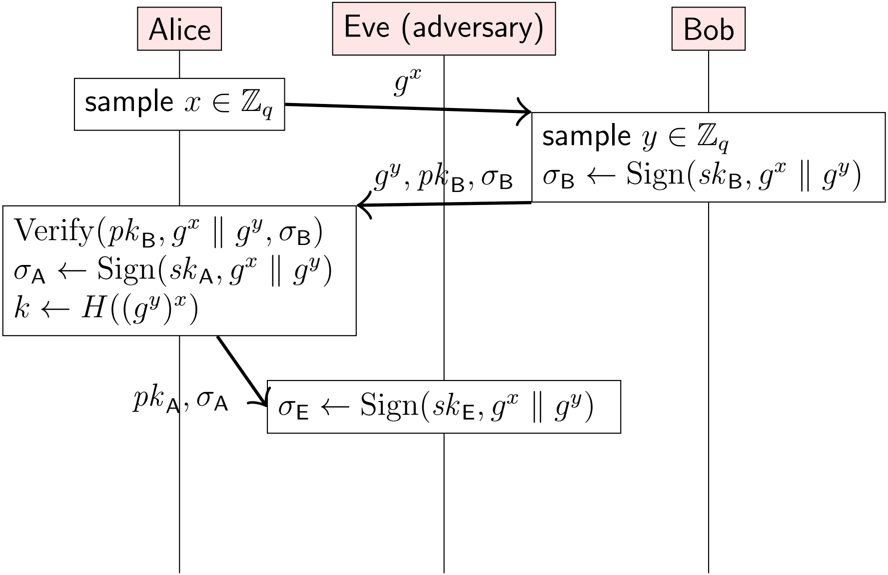
g^x

sample $y \in \mathbb{Z}_q$
$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^x \parallel g^y)$

$g^y, pk_{\mathsf{B}}, \sigma_{\mathsf{B}}$

$\mathrm{Verify}(pk_{\mathsf{B}}, g^x \parallel g^y, \sigma_{\mathsf{B}})$
$\sigma_{\mathsf{A}} \leftarrow \mathrm{Sign}(sk_{\mathsf{A}}, g^x \parallel g^y)$
$k \leftarrow H((g^y)^x)$

Badly Authenticated Diffie-Hellman (2)



Alice

Eve (adversary)

Bob

sample $x \in \mathbb{Z}_q$

g^x

sample $y \in \mathbb{Z}_q$

$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^x \parallel g^y)$

$g^y, pk_{\mathsf{B}}, \sigma_{\mathsf{B}}$

$\mathrm{Verify}(pk_{\mathsf{B}}, g^x \parallel g^y, \sigma_{\mathsf{B}})$

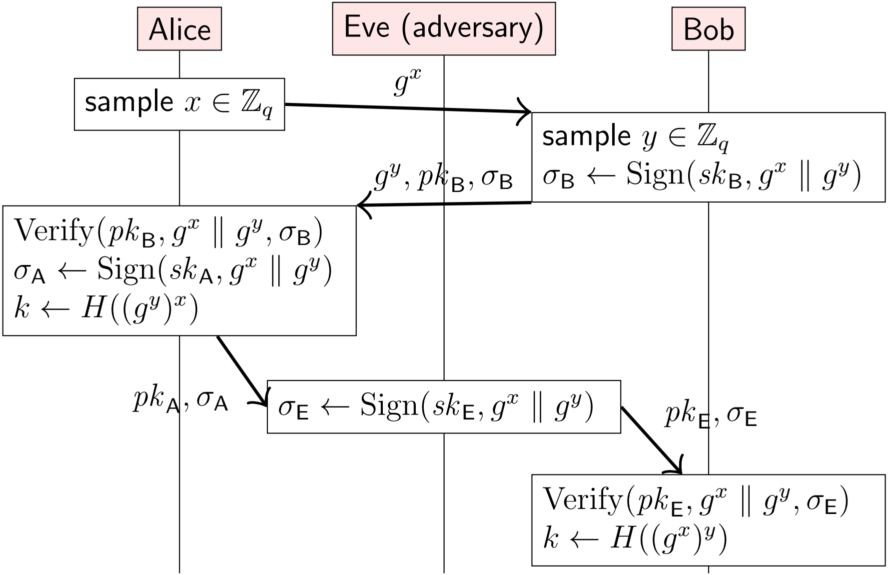
$\sigma_{\mathsf{A}} \leftarrow \mathrm{Sign}(sk_{\mathsf{A}}, g^x \parallel g^y)$

$k \leftarrow H((g^y)^x)$

$pk_{\mathsf{A}}, \sigma_{\mathsf{A}}$

$\sigma_{\mathsf{E}} \leftarrow \mathrm{Sign}(sk_{\mathsf{E}}, g^x \parallel g^y)$

Badly Authenticated Diffie-Hellman (2)



Alice

Eve (adversary)

Bob

sample $x \in \mathbb{Z}_q$

g^x

sample $y \in \mathbb{Z}_q$

$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^x \parallel g^y)$

$g^y, pk_{\mathsf{B}}, \sigma_{\mathsf{B}}$

$\mathrm{Verify}(pk_{\mathsf{B}}, g^x \parallel g^y, \sigma_{\mathsf{B}})$

$\sigma_{\mathsf{A}} \leftarrow \mathrm{Sign}(sk_{\mathsf{A}}, g^x \parallel g^y)$

$k \leftarrow H((g^y)^x)$

$pk_{\mathsf{A}}, \sigma_{\mathsf{A}}$

$\sigma_{\mathsf{E}} \leftarrow \mathrm{Sign}(sk_{\mathsf{E}}, g^x \parallel g^y)$

$pk_{\mathsf{E}}, \sigma_{\mathsf{E}}$

$\mathrm{Verify}(pk_{\mathsf{E}}, g^x \parallel g^y, \sigma_{\mathsf{E}})$

$k \leftarrow H((g^x)^y)$

Again Bob thinks he's talking to Eve

Message Authentication Code (MAC)

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

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

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

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!

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)

Hash-based MAC (HMAC) – RFC 2104

- $H(key \parallel msg)$ is not a secure MAC when H is SHA-256: length extension attacks!

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

Hash-based MAC (HMAC) – RFC 2104

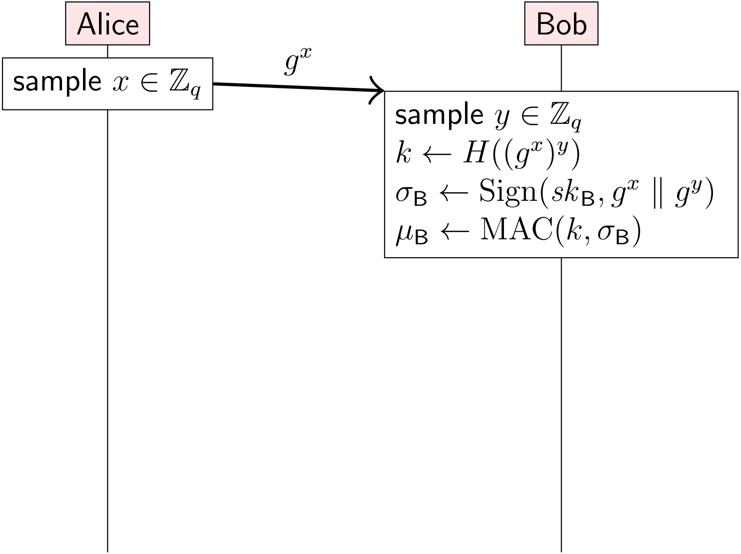
- $H(key \parallel msg)$ is not a secure MAC when H is SHA-256: length extension attacks!
- $\mathrm{HMAC}(key, msg) =$ $H((key \oplus outerPad) \parallel H((key \oplus innerPad) \parallel msg))$ where $\parallel$ is concatenation, $\oplus$ is bit-wise XOR
- Some hash functions (e.g. BLAKE2/3) have a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

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

```python
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))
```

MAC variant of STS protocol



Alice

Bob

sample $x \in \mathbb{Z}_q$

g^x

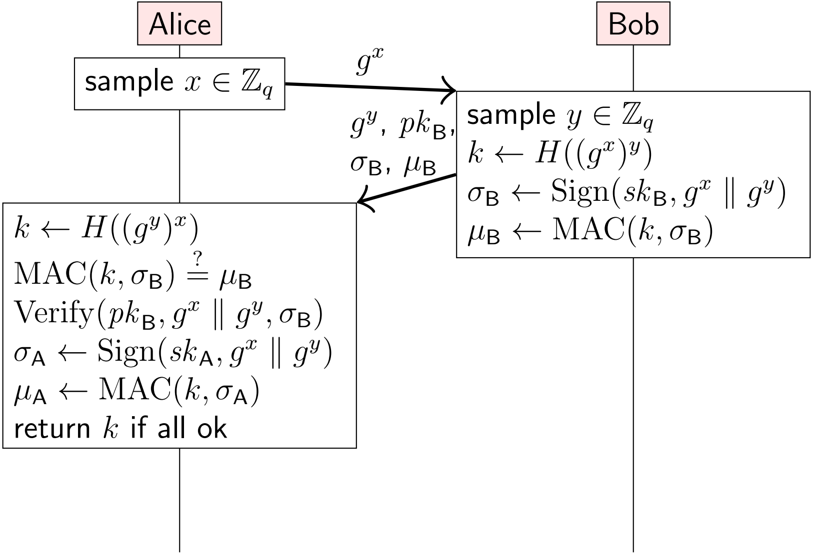
sample $y \in \mathbb{Z}_q$

$k \leftarrow H((g^x)^y)$

$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^x \parallel g^y)$

$\mu_{\mathsf{B}} \leftarrow \mathrm{MAC}(k, \sigma_{\mathsf{B}})$

MAC variant of STS protocol



Alice

sample $x \in \mathbb{Z}_q$

g^x

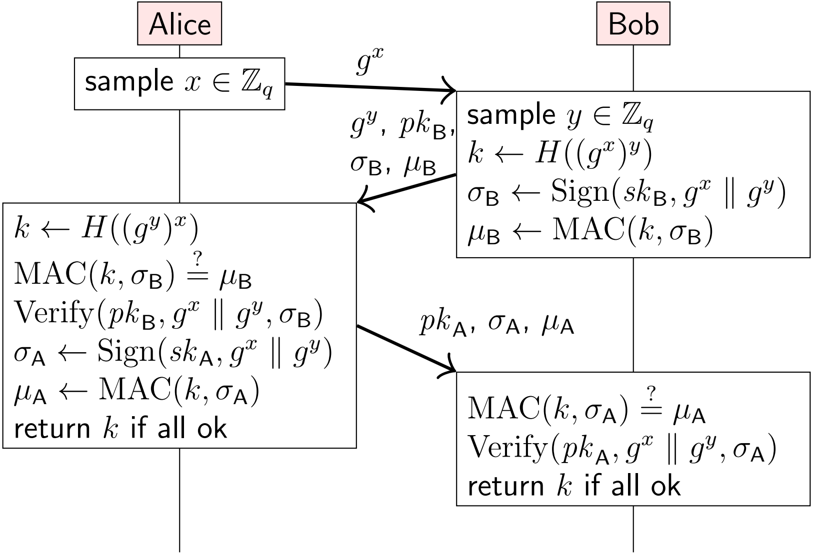
Bob

sample $y \in \mathbb{Z}_q$
$k \leftarrow H((g^x)^y)$
$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^x \parallel g^y)$
$\mu_{\mathsf{B}} \leftarrow \mathrm{MAC}(k, \sigma_{\mathsf{B}})$

g^y, pk_{B}, σ_{B}, μ_{B}

$k \leftarrow H((g^y)^x)$
$\mathrm{MAC}(k, \sigma_{\mathsf{B}}) \overset{?}{=} \mu_{\mathsf{B}}$
$\mathrm{Verify}(pk_{\mathsf{B}}, g^x \parallel g^y, \sigma_{\mathsf{B}})$
$\sigma_{\mathsf{A}} \leftarrow \mathrm{Sign}(sk_{\mathsf{A}}, g^x \parallel g^y)$
$\mu_{\mathsf{A}} \leftarrow \mathrm{MAC}(k, \sigma_{\mathsf{A}})$
return k if all ok

MAC variant of STS protocol



Alice

sample $x \in \mathbb{Z}_q$

g^x

Bob

sample $y \in \mathbb{Z}_q$

$k \leftarrow H((g^x)^y)$

$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^x \parallel g^y)$

$\mu_{\mathsf{B}} \leftarrow \mathrm{MAC}(k, \sigma_{\mathsf{B}})$

g^y, pk_{B}, σ_{B}, μ_{B}

$k \leftarrow H((g^y)^x)$

$\mathrm{MAC}(k, \sigma_{\mathsf{B}}) \overset{?}{=} \mu_{\mathsf{B}}$

$\mathrm{Verify}(pk_{\mathsf{B}}, g^x \parallel g^y, \sigma_{\mathsf{B}})$

$\sigma_{\mathsf{A}} \leftarrow \mathrm{Sign}(sk_{\mathsf{A}}, g^x \parallel g^y)$

$\mu_{\mathsf{A}} \leftarrow \mathrm{MAC}(k, \sigma_{\mathsf{A}})$

return k if all ok

pk_{A}, σ_{A}, μ_{A}

$\mathrm{MAC}(k, \sigma_{\mathsf{A}}) \overset{?}{=} \mu_{\mathsf{A}}$

$\mathrm{Verify}(pk_{\mathsf{A}}, g^x \parallel g^y, \sigma_{\mathsf{A}})$

return k if all ok

Weaknesses in the STS protocol

1. The same k is used as MAC and as session key
    - Violates single-purpose principle

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_{A} and register $(\text{“Eve”}, pk_{\mathsf{A}})$ with the PKI?
    - If so, we have an identity misbinding attack again
    - To prevent, PKI must check whether Eve knows sk_{A}

Weaknesses in the STS protocol

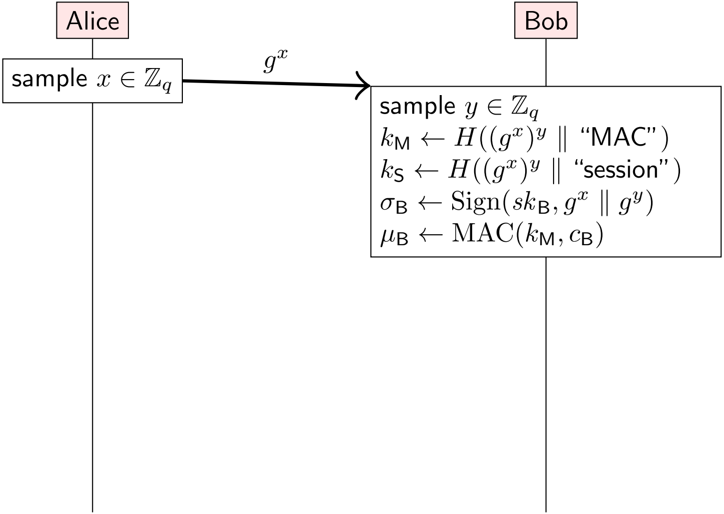
1. The same k is used as MAC and as session key
    - Violates single-purpose principle
2. Identifies the public key of the other side, but not the human-readable name
    - Could Eve take Alice's public key pk_{A} and register $(\text{“Eve”}, pk_{\mathsf{A}})$ with the PKI?
    - If so, we have an identity misbinding attack again
    - To prevent, PKI must check whether Eve knows sk_{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_{new}, i.e. $\mathrm{Verify}(pk_{\mathsf{new}}, m, \sigma_{\mathsf{A}}) =$ true

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_{A} and register $(\text{"Eve"}, pk_{\mathsf{A}})$ with the PKI?
    - If so, we have an identity misbinding attack again
    - To prevent, PKI must check whether Eve knows sk_{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_{new}, i.e. $\mathrm{Verify}(pk_{\mathsf{new}}, m, \sigma_{\mathsf{A}}) = \text{true}$
    - Then Eve registers pk_{new} with PKI and replaces pk_{A} with pk_{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!

SIGMA protocol

Let $c_{\mathsf{A}} = (\text{“Alice”}, pk_{\mathsf{B}}, start, end)$ + PKI signature; c_{B} similar.
Let g be a generator of a group with prime order $|g| = q$.



Alice

Bob

sample $x \in \mathbb{Z}_q$

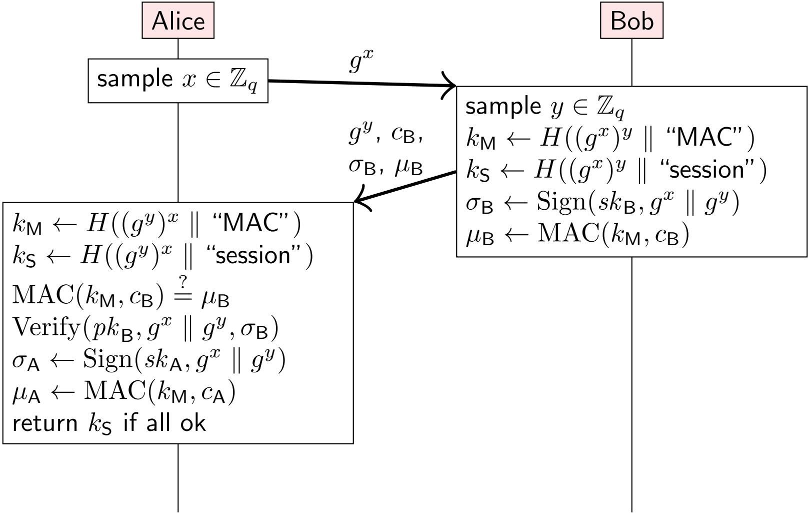
$\xrightarrow{g^x}$

sample $y \in \mathbb{Z}_q$
$k_{\mathsf{M}} \leftarrow H((g^x)^y \parallel \text{“MAC”})$
$k_{\mathsf{S}} \leftarrow H((g^x)^y \parallel \text{“session”})$
$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^x \parallel g^y)$
$\mu_{\mathsf{B}} \leftarrow \mathrm{MAC}(k_{\mathsf{M}}, c_{\mathsf{B}})$

SIGMA protocol

Let $c_{\mathsf{A}} = (\text{“Alice”}, pk_{\mathsf{B}}, start, end)$ + PKI signature; c_{B} similar.

Let g be a generator of a group with prime order $|g| = q$.



Alice

sample $x \in \mathbb{Z}_q$

g^x

Bob

sample $y \in \mathbb{Z}_q$

$k_{\mathsf{M}} \leftarrow H((g^x)^y \parallel \text{“MAC”})$

$k_{\mathsf{S}} \leftarrow H((g^x)^y \parallel \text{“session”})$

$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^x \parallel g^y)$

$\mu_{\mathsf{B}} \leftarrow \mathrm{MAC}(k_{\mathsf{M}}, c_{\mathsf{B}})$

g^y, c_{B}, σ_{B}, μ_{B}

$k_{\mathsf{M}} \leftarrow H((g^y)^x \parallel \text{“MAC”})$

$k_{\mathsf{S}} \leftarrow H((g^y)^x \parallel \text{“session”})$

$\mathrm{MAC}(k_{\mathsf{M}}, c_{\mathsf{B}}) \overset{?}{=} \mu_{\mathsf{B}}$

$\mathrm{Verify}(pk_{\mathsf{B}}, g^x \parallel g^y, \sigma_{\mathsf{B}})$

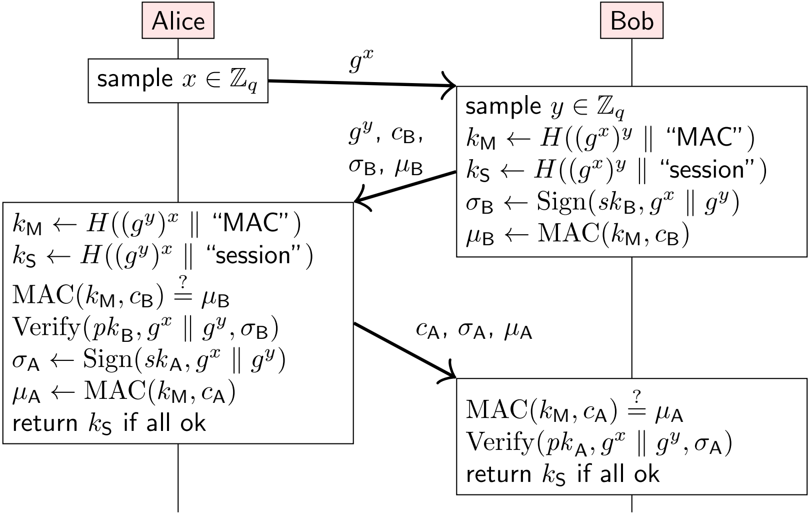
$\sigma_{\mathsf{A}} \leftarrow \mathrm{Sign}(sk_{\mathsf{A}}, g^x \parallel g^y)$

$\mu_{\mathsf{A}} \leftarrow \mathrm{MAC}(k_{\mathsf{M}}, c_{\mathsf{A}})$

return k_{S} if all ok

SIGMA protocol

Let $c_{\mathsf{A}} = (\text{“Alice”}, pk_{\mathsf{B}}, \mathit{start}, \mathit{end}) + \text{PKI signature}$; c_{B} similar.
Let g be a generator of a group with prime order $|g| = q$.



Alice

sample $x \in \mathbb{Z}_q$

g^x

Bob

sample $y \in \mathbb{Z}_q$
$k_{\mathsf{M}} \leftarrow H((g^x)^y \parallel \text{“MAC”})$
$k_{\mathsf{S}} \leftarrow H((g^x)^y \parallel \text{“session”})$
$\sigma_{\mathsf{B}} \leftarrow \mathrm{Sign}(sk_{\mathsf{B}}, g^x \parallel g^y)$
$\mu_{\mathsf{B}} \leftarrow \mathrm{MAC}(k_{\mathsf{M}}, c_{\mathsf{B}})$

g^y, c_{B}, σ_{B}, μ_{B}

$k_{\mathsf{M}} \leftarrow H((g^y)^x \parallel \text{“MAC”})$
$k_{\mathsf{S}} \leftarrow H((g^y)^x \parallel \text{“session”})$
$\mathrm{MAC}(k_{\mathsf{M}}, c_{\mathsf{B}}) \stackrel{?}{=} \mu_{\mathsf{B}}$
$\mathrm{Verify}(pk_{\mathsf{B}}, g^x \parallel g^y, \sigma_{\mathsf{B}})$
$\sigma_{\mathsf{A}} \leftarrow \mathrm{Sign}(sk_{\mathsf{A}}, g^x \parallel g^y)$
$\mu_{\mathsf{A}} \leftarrow \mathrm{MAC}(k_{\mathsf{M}}, c_{\mathsf{A}})$
return k_{S} if all ok

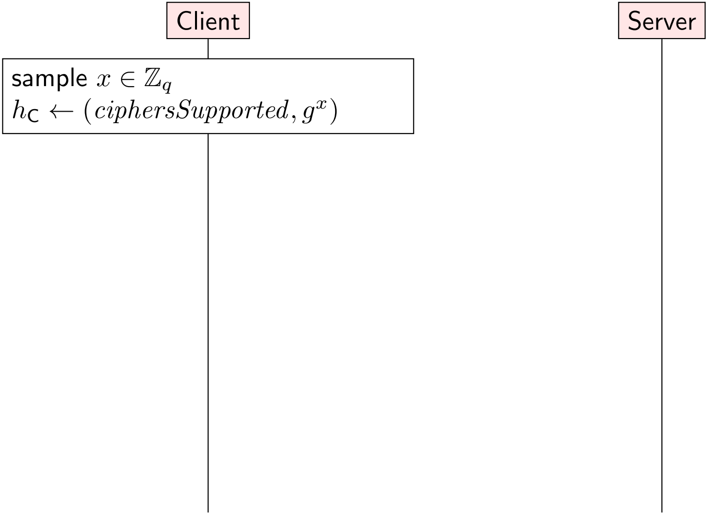
c_{A}, σ_{A}, μ_{A}

$\mathrm{MAC}(k_{\mathsf{M}}, c_{\mathsf{A}}) \stackrel{?}{=} \mu_{\mathsf{A}}$
$\mathrm{Verify}(pk_{\mathsf{A}}, g^x \parallel g^y, \sigma_{\mathsf{A}})$
return k_{S} if all ok

Finally secure?!

TLS handshake, simplified

Let $c_{\mathsf{S}} = (\text{“example.com”}, pk_{\mathsf{S}}, start, end)$ + PKI signature



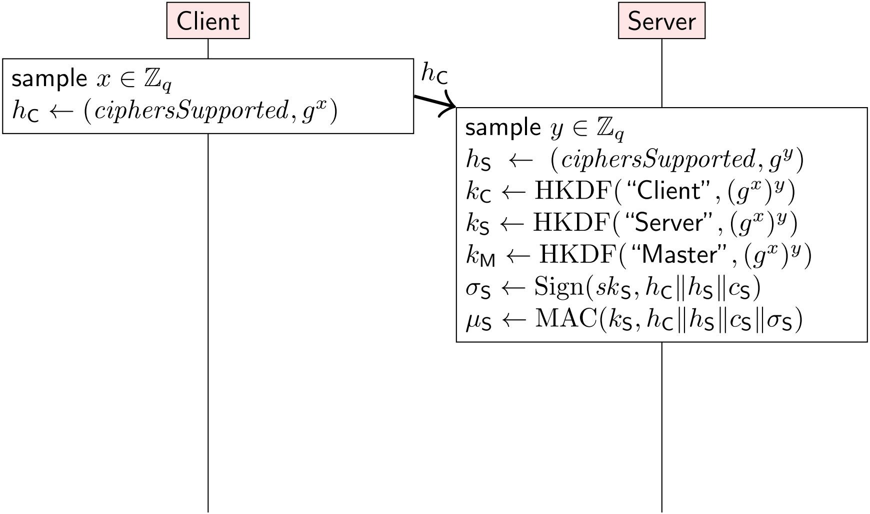
Client

Server

sample $x \in \mathbb{Z}_q$
$h_{\mathsf{C}} \leftarrow (ciphersSupported, g^x)$

TLS handshake, simplified

Let $c_{\mathsf{S}} = (\text{“example.com”}, pk_{\mathsf{S}}, start, end)$ + PKI signature



Client

sample $x \in \mathbb{Z}_q$
$h_{\mathsf{C}} \leftarrow (ciphersSupported, g^x)$

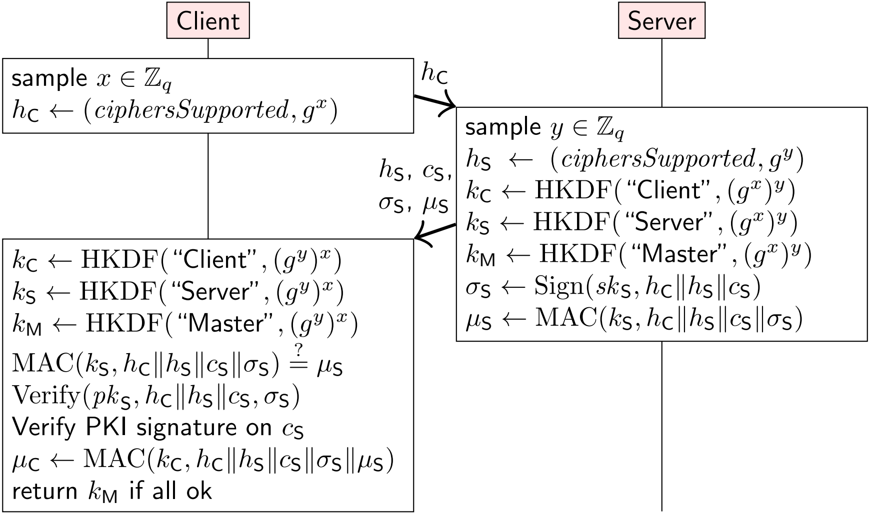
$$\xrightarrow{h_{\mathsf{C}}}$$

Server

sample $y \in \mathbb{Z}_q$
$h_{\mathsf{S}} \leftarrow (ciphersSupported, g^y)$
$k_{\mathsf{C}} \leftarrow \mathrm{HKDF}(\text{“Client”}, (g^x)^y)$
$k_{\mathsf{S}} \leftarrow \mathrm{HKDF}(\text{“Server”}, (g^x)^y)$
$k_{\mathsf{M}} \leftarrow \mathrm{HKDF}(\text{“Master”}, (g^x)^y)$
$\sigma_{\mathsf{S}} \leftarrow \mathrm{Sign}(sk_{\mathsf{S}}, h_{\mathsf{C}} \| h_{\mathsf{S}} \| c_{\mathsf{S}})$
$\mu_{\mathsf{S}} \leftarrow \mathrm{MAC}(k_{\mathsf{S}}, h_{\mathsf{C}} \| h_{\mathsf{S}} \| c_{\mathsf{S}} \| \sigma_{\mathsf{S}})$

TLS handshake, simplified

Let $c_\mathsf{S} = (\text{“example.com”}, pk_\mathsf{S}, start, end)$ + PKI signature



Client

Server

Client:

sample $x \in \mathbb{Z}_q$

$h_\mathsf{C} \leftarrow (ciphersSupported, g^x)$

Client → Server: h_C

Server:

sample $y \in \mathbb{Z}_q$

$h_\mathsf{S} \leftarrow (ciphersSupported, g^y)$

$k_\mathsf{C} \leftarrow \mathrm{HKDF}(\text{“Client”}, (g^x)^y)$

$k_\mathsf{S} \leftarrow \mathrm{HKDF}(\text{“Server”}, (g^x)^y)$

$k_\mathsf{M} \leftarrow \mathrm{HKDF}(\text{“Master”}, (g^x)^y)$

$\sigma_\mathsf{S} \leftarrow \mathrm{Sign}(sk_\mathsf{S}, h_\mathsf{C} \| h_\mathsf{S} \| c_\mathsf{S})$

$\mu_\mathsf{S} \leftarrow \mathrm{MAC}(k_\mathsf{S}, h_\mathsf{C} \| h_\mathsf{S} \| c_\mathsf{S} \| \sigma_\mathsf{S})$

Server → Client: h_S, c_S, σ_S, μ_S

Client:

$k_\mathsf{C} \leftarrow \mathrm{HKDF}(\text{“Client”}, (g^y)^x)$

$k_\mathsf{S} \leftarrow \mathrm{HKDF}(\text{“Server”}, (g^y)^x)$

$k_\mathsf{M} \leftarrow \mathrm{HKDF}(\text{“Master”}, (g^y)^x)$

$\mathrm{MAC}(k_\mathsf{S}, h_\mathsf{C} \| h_\mathsf{S} \| c_\mathsf{S} \| \sigma_\mathsf{S}) \stackrel{?}{=} \mu_\mathsf{S}$

$\mathrm{Verify}(pk_\mathsf{S}, h_\mathsf{C} \| h_\mathsf{S} \| c_\mathsf{S}, \sigma_\mathsf{S})$

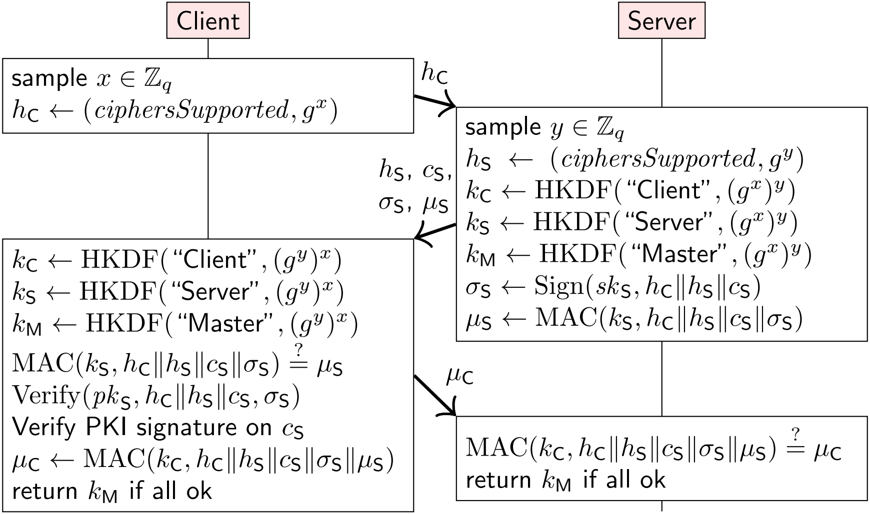
Verify PKI signature on c_S

$\mu_\mathsf{C} \leftarrow \mathrm{MAC}(k_\mathsf{C}, h_\mathsf{C} \| h_\mathsf{S} \| c_\mathsf{S} \| \sigma_\mathsf{S} \| \mu_\mathsf{S})$

return k_M if all ok

TLS handshake, simplified

Let $c_{\mathsf{S}} = (\text{“example.com”}, pk_{\mathsf{S}}, start, end)$ + PKI signature



Client

sample $x \in \mathbb{Z}_q$

$h_{\mathsf{C}} \leftarrow (ciphersSupported, g^x)$

h_{C}

Server

sample $y \in \mathbb{Z}_q$

$h_{\mathsf{S}} \leftarrow (ciphersSupported, g^y)$

$k_{\mathsf{C}} \leftarrow \mathrm{HKDF}(\text{“Client”}, (g^x)^y)$

$k_{\mathsf{S}} \leftarrow \mathrm{HKDF}(\text{“Server”}, (g^x)^y)$

$k_{\mathsf{M}} \leftarrow \mathrm{HKDF}(\text{“Master”}, (g^x)^y)$

$\sigma_{\mathsf{S}} \leftarrow \mathrm{Sign}(sk_{\mathsf{S}}, h_{\mathsf{C}} \| h_{\mathsf{S}} \| c_{\mathsf{S}})$

$\mu_{\mathsf{S}} \leftarrow \mathrm{MAC}(k_{\mathsf{S}}, h_{\mathsf{C}} \| h_{\mathsf{S}} \| c_{\mathsf{S}} \| \sigma_{\mathsf{S}})$

h_{S}, c_{S}, σ_{S}, μ_{S}

$k_{\mathsf{C}} \leftarrow \mathrm{HKDF}(\text{“Client”}, (g^y)^x)$

$k_{\mathsf{S}} \leftarrow \mathrm{HKDF}(\text{“Server”}, (g^y)^x)$

$k_{\mathsf{M}} \leftarrow \mathrm{HKDF}(\text{“Master”}, (g^y)^x)$

$\mathrm{MAC}(k_{\mathsf{S}}, h_{\mathsf{C}} \| h_{\mathsf{S}} \| c_{\mathsf{S}} \| \sigma_{\mathsf{S}}) \overset{?}{=} \mu_{\mathsf{S}}$

$\mathrm{Verify}(pk_{\mathsf{S}}, h_{\mathsf{C}} \| h_{\mathsf{S}} \| c_{\mathsf{S}}, \sigma_{\mathsf{S}})$

Verify PKI signature on c_{S}

$\mu_{\mathsf{C}} \leftarrow \mathrm{MAC}(k_{\mathsf{C}}, h_{\mathsf{C}} \| h_{\mathsf{S}} \| c_{\mathsf{S}} \| \sigma_{\mathsf{S}} \| \mu_{\mathsf{S}})$

return k_{M} if all ok

μ_{C}

$\mathrm{MAC}(k_{\mathsf{C}}, h_{\mathsf{C}} \| h_{\mathsf{S}} \| c_{\mathsf{S}} \| \sigma_{\mathsf{S}} \| \mu_{\mathsf{S}}) \overset{?}{=} \mu_{\mathsf{C}}$

return k_{M} if all ok

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!

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

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)

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

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, . . .)

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)

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

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**?

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

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)

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**?

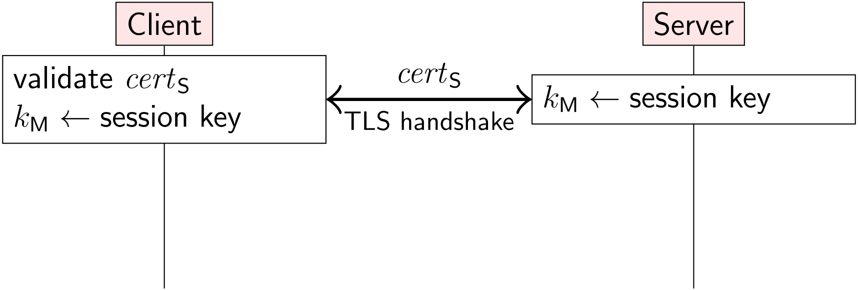
PAKE: no global names, no pubkeys, just a shared password

- 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

Passwords on the web: Not a PAKE

Let $cert_{\mathsf{S}} = (\text{“example.com”}, pk_{\mathsf{S}}, start, end)$ + PKI signature



Client

validate $cert_{\mathsf{S}}$

$k_{\mathsf{M}} \leftarrow$ session key

$cert_{\mathsf{S}}$

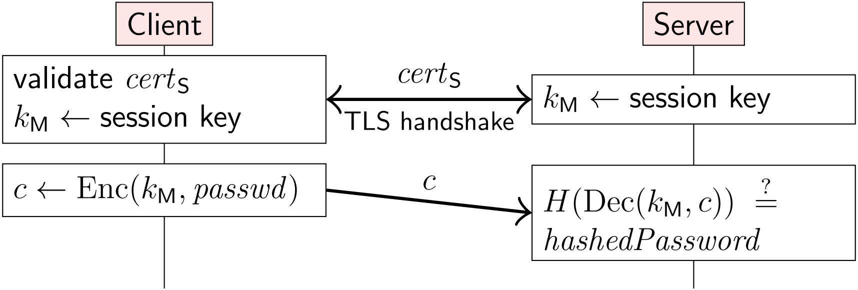
TLS handshake

Server

$k_{\mathsf{M}} \leftarrow$ session key

Passwords on the web: Not a PAKE

Let $cert_{\mathsf{S}} = (\text{“example.com”}, pk_{\mathsf{S}}, start, end)$ + PKI signature



Client

validate $cert_{\mathsf{S}}$
$k_{\mathsf{M}} \leftarrow$ session key

Server

$k_{\mathsf{M}} \leftarrow$ session key

$cert_{\mathsf{S}}$
TLS handshake

$c \leftarrow \mathrm{Enc}(k_{\mathsf{M}}, passwd)$

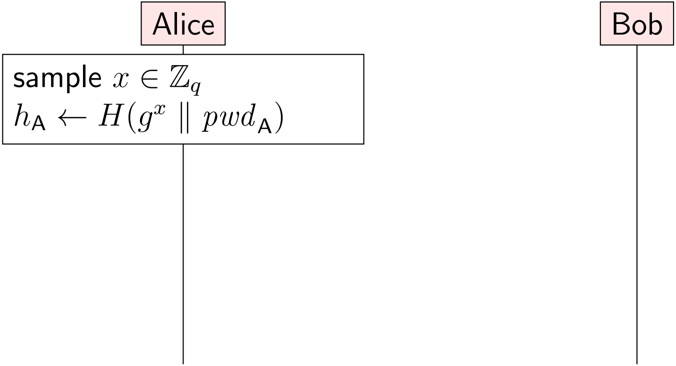
c

$H(\mathrm{Dec}(k_{\mathsf{M}}, c)) \stackrel{?}{=} hashedPassword$

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

An insecure PAKE

Say Alice knows a password pwd_{A} and Bob knows pwd_{B}. We want them to agree on a shared secret k if $pwd_{\mathsf{A}} = pwd_{\mathsf{B}}$.



Alice

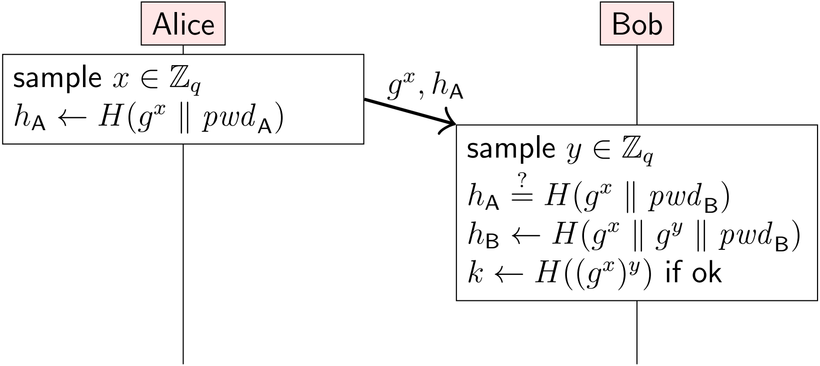
Bob

sample $x \in \mathbb{Z}_q$

$h_{\mathsf{A}} \leftarrow H(g^x \parallel pwd_{\mathsf{A}})$

An insecure PAKE

Say Alice knows a password pwd_{A} and Bob knows pwd_{B}. We want them to agree on a shared secret k if $pwd_{\mathsf{A}} = pwd_{\mathsf{B}}$.



Alice

sample $x \in \mathbb{Z}_q$

$h_{\mathsf{A}} \leftarrow H(g^x \parallel pwd_{\mathsf{A}})$

g^x, h_{A}

Bob

sample $y \in \mathbb{Z}_q$

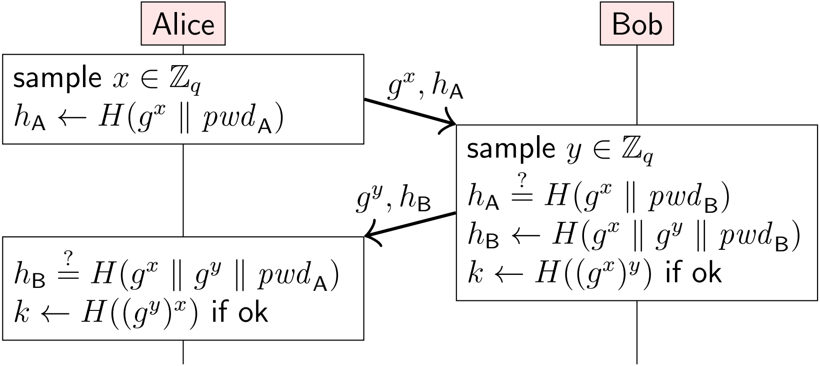
$h_{\mathsf{A}} \overset{?}{=} H(g^x \parallel pwd_{\mathsf{B}})$

$h_{\mathsf{B}} \leftarrow H(g^x \parallel g^y \parallel pwd_{\mathsf{B}})$

$k \leftarrow H((g^x)^y)$ if ok

An insecure PAKE

Say Alice knows a password pwd_{A} and Bob knows pwd_{B}. We want them to agree on a shared secret k if $pwd_{\mathsf{A}} = pwd_{\mathsf{B}}$.



Alice

sample $x \in \mathbb{Z}_q$

$h_{\mathsf{A}} \leftarrow H(g^x \parallel pwd_{\mathsf{A}})$

g^x, h_{A}

Bob

sample $y \in \mathbb{Z}_q$

$h_{\mathsf{A}} \overset{?}{=} H(g^x \parallel pwd_{\mathsf{B}})$

$h_{\mathsf{B}} \leftarrow H(g^x \parallel g^y \parallel pwd_{\mathsf{B}})$

$k \leftarrow H((g^x)^y)$ if ok

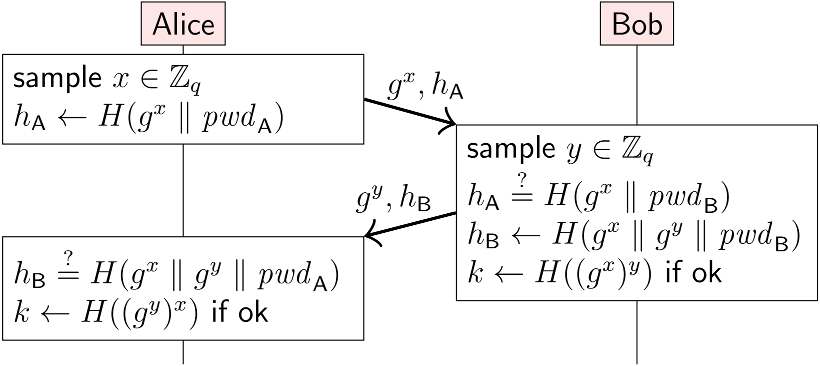
g^y, h_{B}

$h_{\mathsf{B}} \overset{?}{=} H(g^x \parallel g^y \parallel pwd_{\mathsf{A}})$

$k \leftarrow H((g^y)^x)$ if ok

An insecure PAKE

Say Alice knows a password pwd_{A} and Bob knows pwd_{B}. We want them to agree on a shared secret k if $pwd_{\mathsf{A}} = pwd_{\mathsf{B}}$.



Alice

sample $x \in \mathbb{Z}_q$

$h_{\mathsf{A}} \leftarrow H(g^x \parallel pwd_{\mathsf{A}})$

g^x, h_{A}

Bob

sample $y \in \mathbb{Z}_q$

$h_{\mathsf{A}} \overset{?}{=} H(g^x \parallel pwd_{\mathsf{B}})$

$h_{\mathsf{B}} \leftarrow H(g^x \parallel g^y \parallel pwd_{\mathsf{B}})$

$k \leftarrow H((g^x)^y)$ if ok

g^y, h_{B}

$h_{\mathsf{B}} \overset{?}{=} H(g^x \parallel g^y \parallel pwd_{\mathsf{A}})$

$k \leftarrow H((g^y)^x)$ if ok

Problem: adversary who has intercepted g^x and h_{A} can run an offline brute-force search, trying many passwords p until they find one such that $H(g^x \parallel p) = h_{\mathsf{A}}$

The SPAKE2 protocol

Let E be a group of order $|E| = hq$. Let $g \in E$ be a generator of prime order $|g| = q$. Let $M, N \in E$ be elements whose discrete log is unknown.

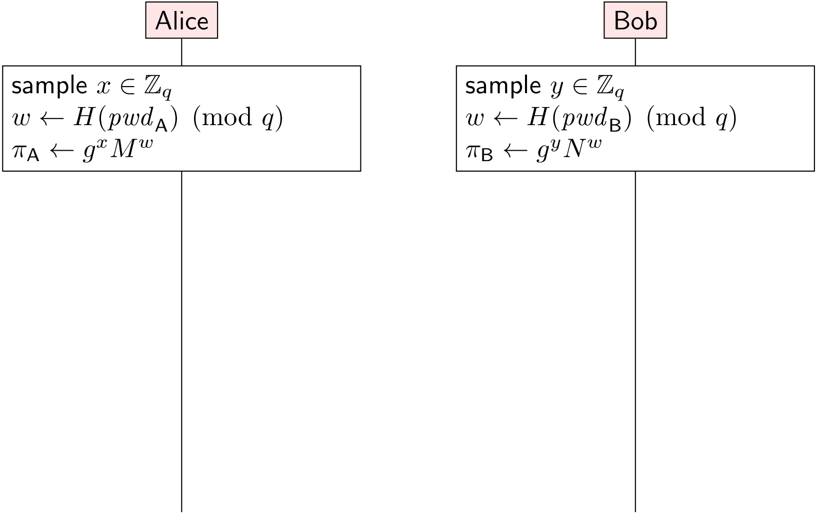
Alice

Bob

sample $x \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{A}}) \pmod{q}$
$\pi_{\mathsf{A}} \leftarrow g^x M^w$

The SPAKE2 protocol

Let E be a group of order $|E| = hq$. Let $g \in E$ be a generator of prime order $|g| = q$. Let $M, N \in E$ be elements whose discrete log is unknown.



Alice

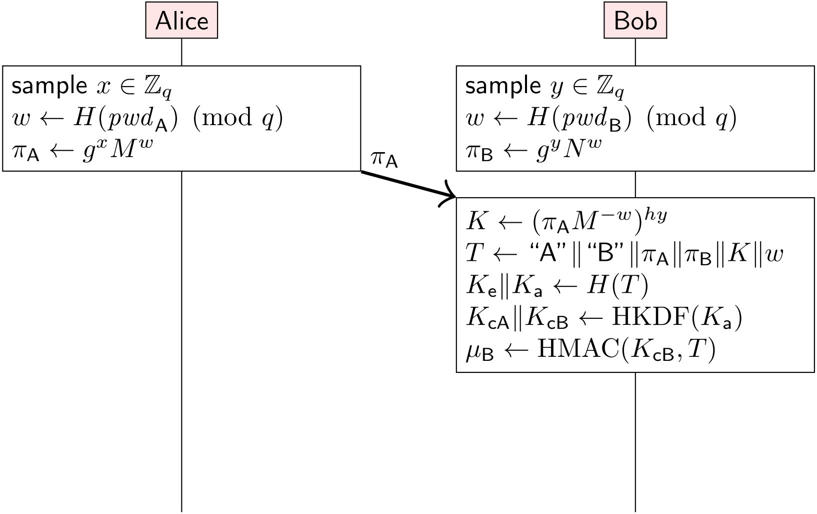
sample $x \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{A}}) \pmod{q}$
$\pi_{\mathsf{A}} \leftarrow g^x M^w$

Bob

sample $y \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{B}}) \pmod{q}$
$\pi_{\mathsf{B}} \leftarrow g^y N^w$

The SPAKE2 protocol

Let E be a group of order $|E| = hq$. Let $g \in E$ be a generator of prime order $|g| = q$. Let $M, N \in E$ be elements whose discrete log is unknown.



Alice

sample $x \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{A}}) \pmod q$
$\pi_{\mathsf{A}} \leftarrow g^x M^w$

π_{A}

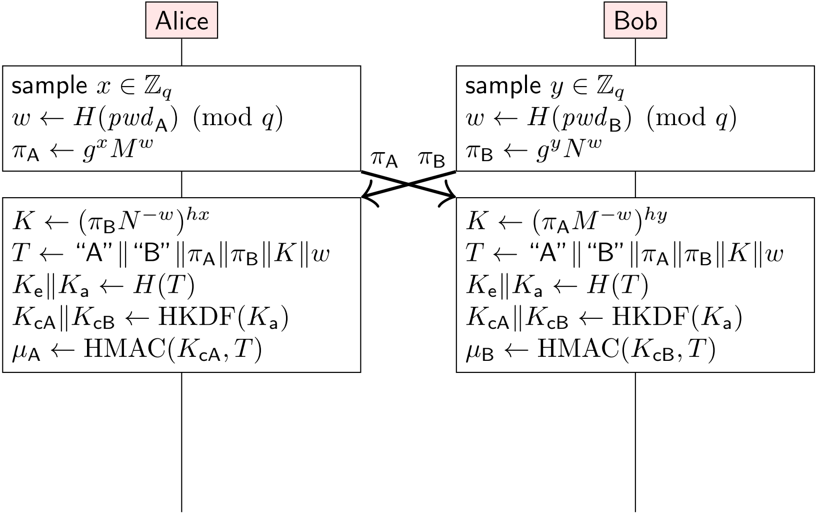
Bob

sample $y \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{B}}) \pmod q$
$\pi_{\mathsf{B}} \leftarrow g^y N^w$

$K \leftarrow (\pi_{\mathsf{A}} M^{-w})^{hy}$
$T \leftarrow \text{“A”} \| \text{“B”} \| \pi_{\mathsf{A}} \| \pi_{\mathsf{B}} \| K \| w$
$K_{\mathsf{e}} \| K_{\mathsf{a}} \leftarrow H(T)$
$K_{\mathsf{cA}} \| K_{\mathsf{cB}} \leftarrow \mathrm{HKDF}(K_{\mathsf{a}})$
$\mu_{\mathsf{B}} \leftarrow \mathrm{HMAC}(K_{\mathsf{cB}}, T)$

The SPAKE2 protocol

Let E be a group of order $|E| = hq$. Let $g \in E$ be a generator of prime order $|g| = q$. Let $M, N \in E$ be elements whose discrete log is unknown.



Alice

sample $x \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{A}}) \pmod q$
$\pi_{\mathsf{A}} \leftarrow g^x M^w$

$K \leftarrow (\pi_{\mathsf{B}} N^{-w})^{hx}$
$T \leftarrow \text{“A”} \| \text{“B”} \| \pi_{\mathsf{A}} \| \pi_{\mathsf{B}} \| K \| w$
$K_{\mathsf{e}} \| K_{\mathsf{a}} \leftarrow H(T)$
$K_{\mathsf{cA}} \| K_{\mathsf{cB}} \leftarrow \mathrm{HKDF}(K_{\mathsf{a}})$
$\mu_{\mathsf{A}} \leftarrow \mathrm{HMAC}(K_{\mathsf{cA}}, T)$

π_{A} π_{B}

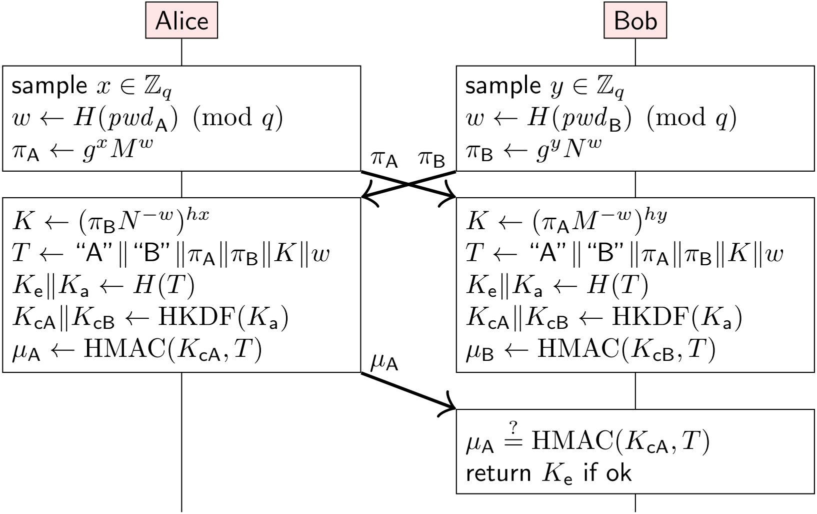
Bob

sample $y \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{B}}) \pmod q$
$\pi_{\mathsf{B}} \leftarrow g^y N^w$

$K \leftarrow (\pi_{\mathsf{A}} M^{-w})^{hy}$
$T \leftarrow \text{“A”} \| \text{“B”} \| \pi_{\mathsf{A}} \| \pi_{\mathsf{B}} \| K \| w$
$K_{\mathsf{e}} \| K_{\mathsf{a}} \leftarrow H(T)$
$K_{\mathsf{cA}} \| K_{\mathsf{cB}} \leftarrow \mathrm{HKDF}(K_{\mathsf{a}})$
$\mu_{\mathsf{B}} \leftarrow \mathrm{HMAC}(K_{\mathsf{cB}}, T)$

The SPAKE2 protocol

Let E be a group of order $|E| = hq$. Let $g \in E$ be a generator of prime order $|g| = q$. Let $M, N \in E$ be elements whose discrete log is unknown.



Alice

sample $x \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{A}}) \pmod q$
$\pi_{\mathsf{A}} \leftarrow g^x M^w$

$K \leftarrow (\pi_{\mathsf{B}} N^{-w})^{hx}$
$T \leftarrow \text{“A”} \| \text{“B”} \| \pi_{\mathsf{A}} \| \pi_{\mathsf{B}} \| K \| w$
$K_{\mathsf{e}} \| K_{\mathsf{a}} \leftarrow H(T)$
$K_{\mathsf{cA}} \| K_{\mathsf{cB}} \leftarrow \mathrm{HKDF}(K_{\mathsf{a}})$
$\mu_{\mathsf{A}} \leftarrow \mathrm{HMAC}(K_{\mathsf{cA}}, T)$

Bob

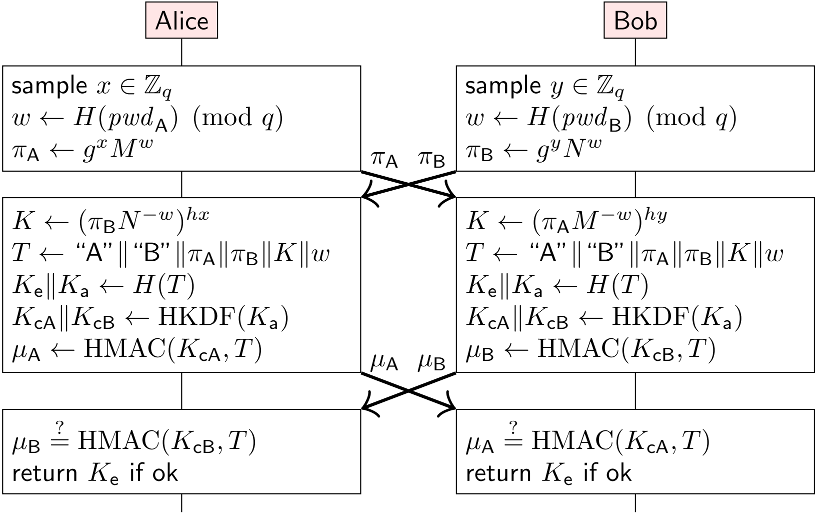
sample $y \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{B}}) \pmod q$
$\pi_{\mathsf{B}} \leftarrow g^y N^w$

$K \leftarrow (\pi_{\mathsf{A}} M^{-w})^{hy}$
$T \leftarrow \text{“A”} \| \text{“B”} \| \pi_{\mathsf{A}} \| \pi_{\mathsf{B}} \| K \| w$
$K_{\mathsf{e}} \| K_{\mathsf{a}} \leftarrow H(T)$
$K_{\mathsf{cA}} \| K_{\mathsf{cB}} \leftarrow \mathrm{HKDF}(K_{\mathsf{a}})$
$\mu_{\mathsf{B}} \leftarrow \mathrm{HMAC}(K_{\mathsf{cB}}, T)$

$\mu_{\mathsf{A}} \overset{?}{=} \mathrm{HMAC}(K_{\mathsf{cA}}, T)$
return K_{e} if ok

The SPAKE2 protocol

Let E be a group of order $|E| = hq$. Let $g \in E$ be a generator of prime order $|g| = q$. Let $M, N \in E$ be elements whose discrete log is unknown.



Alice

sample $x \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{A}}) \pmod q$
$\pi_{\mathsf{A}} \leftarrow g^x M^w$

$K \leftarrow (\pi_{\mathsf{B}} N^{-w})^{hx}$
$T \leftarrow \text{“A”} \| \text{“B”} \| \pi_{\mathsf{A}} \| \pi_{\mathsf{B}} \| K \| w$
$K_{\mathsf{e}} \| K_{\mathsf{a}} \leftarrow H(T)$
$K_{\mathsf{cA}} \| K_{\mathsf{cB}} \leftarrow \mathrm{HKDF}(K_{\mathsf{a}})$
$\mu_{\mathsf{A}} \leftarrow \mathrm{HMAC}(K_{\mathsf{cA}}, T)$

$\mu_{\mathsf{B}} \overset{?}{=} \mathrm{HMAC}(K_{\mathsf{cB}}, T)$
return K_{e} if ok

Bob

sample $y \in \mathbb{Z}_q$
$w \leftarrow H(pwd_{\mathsf{B}}) \pmod q$
$\pi_{\mathsf{B}} \leftarrow g^y N^w$

$K \leftarrow (\pi_{\mathsf{A}} M^{-w})^{hy}$
$T \leftarrow \text{“A”} \| \text{“B”} \| \pi_{\mathsf{A}} \| \pi_{\mathsf{B}} \| K \| w$
$K_{\mathsf{e}} \| K_{\mathsf{a}} \leftarrow H(T)$
$K_{\mathsf{cA}} \| K_{\mathsf{cB}} \leftarrow \mathrm{HKDF}(K_{\mathsf{a}})$
$\mu_{\mathsf{B}} \leftarrow \mathrm{HMAC}(K_{\mathsf{cB}}, T)$

$\mu_{\mathsf{A}} \overset{?}{=} \mathrm{HMAC}(K_{\mathsf{cA}}, T)$
return K_{e} if ok

SPAKE2: Why it works

- $\pi_{\mathsf{A}} = g^x M^w \implies (\pi_{\mathsf{A}} M^{-w})^{hy} = (g^x M^w M^{-w})^{hy} = g^{hxy}$

SPAKE2: Why it works

- $\pi_{\mathsf{A}} = g^x M^w \implies (\pi_{\mathsf{A}} M^{-w})^{hy} = (g^x M^w M^{-w})^{hy} = g^{hxy}$
- π_{A} and π_{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

SPAKE2: Why it works

- $\pi_{\mathsf{A}} = g^x M^w \implies (\pi_{\mathsf{A}} M^{-w})^{hy} = (g^x M^w M^{-w})^{hy} = g^{hxy}$
- π_{A} and π_{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 μ_{A} allows the party that generated π_{B} to verify whether the passwords were equal (similarly with μ_{B} and π_{A}), but not an eavesdropper

SPAKE2: Why it works

- $\pi_{\mathsf{A}} = g^x M^w \implies (\pi_{\mathsf{A}} M^{-w})^{hy} = (g^x M^w M^{-w})^{hy} = g^{hxy}$
- π_{A} and π_{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 μ_{A} allows the party that generated π_{B} to verify whether the passwords were equal (similarly with μ_{B} and π_{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

SPAKE2: Why it works

- $\pi_{\mathsf{A}} = g^x M^w \implies (\pi_{\mathsf{A}} M^{-w})^{hy} = (g^x M^w M^{-w})^{hy} = g^{hxy}$
- π_{A} and π_{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 μ_{A} allows the party that generated π_{B} to verify whether the passwords were equal (similarly with μ_{B} and π_{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

SPAKE2: Trusted setup

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

- Adversary pretends to be Bob, sets $\pi_{\mathsf{B}} = g^y$, receives $\pi_{\mathsf{A}} = g^x M^w$ and μ_{A} from Alice

SPAKE2: Trusted setup

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

- Adversary pretends to be Bob, sets $\pi_{\mathsf{B}} = g^y$, receives $\pi_{\mathsf{A}} = g^x M^w$ and μ_{A} from Alice
- Alice computes $K = (\pi_{\mathsf{B}} N^{-w})^{hx} = (g^y g^{-nw})^{hx} = (g^x)^{(y-nw)h} = (\pi_{\mathsf{A}} M^{-w})^{(y-nw)h}$

SPAKE2: Trusted setup

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

- Adversary pretends to be Bob, sets $\pi_{\mathsf{B}} = g^y$, receives $\pi_{\mathsf{A}} = g^x M^w$ and μ_{A} from Alice
- Alice computes $K = (\pi_{\mathsf{B}} N^{-w})^{hx} = (g^y g^{-nw})^{hx} = (g^x)^{(y-nw)h} = (\pi_{\mathsf{A}} M^{-w})^{(y-nw)h}$
- Adversary knows π_{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 μ_{A} from Alice, pwd guess is correct
- Adversary can now run offline brute-force search

SPAKE2: Trusted setup

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

- Adversary pretends to be Bob, sets $\pi_{\mathsf{B}} = g^y$, receives $\pi_{\mathsf{A}} = g^x M^w$ and μ_{A} from Alice
- Alice computes $K = (\pi_{\mathsf{B}} N^{-w})^{hx} = (g^y g^{-nw})^{hx} = (g^x)^{(y-nw)h} = (\pi_{\mathsf{A}} M^{-w})^{(y-nw)h}$
- Adversary knows π_{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 μ_{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.

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 2025

Software Engineering for Cryptography – Part II

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

University of Cambridge
Part III/MPhil in Advanced Computer Science

Randomness

- Cryptography requires random inputs
    - Key generation
    - Nonce generation
    - Tokens
    - ...

Randomness

- Cryptography requires random inputs
  - Key generation
  - Nonce generation
  - Tokens
  - ...

- Building strong random number generators is not trivial

LAB: build your own PRNG (10 min)

Hidden from the published slides.

LAB: collect (5 min)

Hidden from the published slides.

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
- . . .

Lava lamps

Image from Wikipedia (CC-BY 2.0 Dean Hochman)

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
- . . .

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

Cryptographically secure PRNGs (CSPRNGs)

PRNGs are not suitable for cryptographic applications.

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

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

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

Measuring entropy

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

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.

LAB: compare (10 min)

Hidden from the published slides.

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
- ...

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
- ...

The Linux CSPRNG design

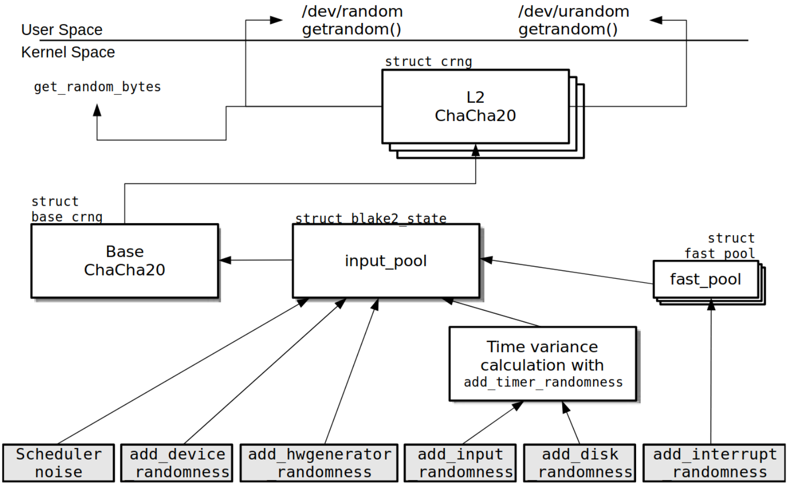


Image from [BSI, 2022].

CSPRNGs in practice

Linux:

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

---

[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`.

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`.

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

- OpenSSL collects entropy from many different sources (`/dev/urandom`, time, . . .)

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

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));
    /* ... */
}
```

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

- Down-stream developers (Debian) saw Valgrind warnings

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

- Down-stream developers (Debian) saw Valgrind warnings
- Remove two lines that adds these uninitialized buffers

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 $|K| = 32\,768$

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 $|K| = 32\,768$

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.

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:

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 $\mathcal{X}$:

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

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 $\mathcal{X}$:

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

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.

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.

Indistinguishability

Two probability ensembles $\mathcal{X} = \{X_n\}_{n\in\mathbb{N}}$ and $\mathcal{Y} = \{Y_n\}_{n\in\mathbb{N}}$ are **computationally indistinguishable** if for every probabilistic polynomial-time distinguisher D there exists a negligible function negl such that:

$$\left|\Pr_{x \overset{\$}{\leftarrow} X_n}[D(x) = 1] - \Pr_{y \overset{\$}{\leftarrow} Y_n}[D(y) = 1]\right| \leq \mathsf{negl}(n)$$

The distinguisher D would output 0 if it thinks its input is sampled from $X \in \mathcal{X}$ and 1 if it thinks its input is sampled from $Y \in \mathcal{Y}$.

Testing and correctness

> *"Testing security is pretty much impossible. It's hard to know if you're ever done."*
>
> – Daniel Rohrer, VP of Software Security at NVIDIA

Approaching correctness

- It is really really hard to convince ourselves that our code is always, always correct and secure

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

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, . . .

Testing strategies

- Bottom-up: testing individual operation thoroughly across their specification range

Testing strategies

- Bottom-up: testing individual operation thoroughly across their specification range
- Top-down: economic testing of overall functionality and compatibility

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

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)

Formal approaches

For serious real-world implementations, it is helpful to ground the *implementation strategy* in a formal approach.

Formal approaches

For serious real-world implementations, it is helpful to ground the *implementation strategy* in a formal approach.

- Derive implementation step-by-step from specification
  - That is most-likely the best approach for your lab reports
  - Scope: a few days

Formal approaches

For serious real-world implementations, it is helpful to ground the *implementation strategy* in a formal approach.

- Derive implementation step-by-step from specification
    - That is most-likely the best approach for your lab reports
    - Scope: a few days

- Prove all implementation steps
    - Requires model of the underlying system (compiler, hardware, ...)
    - Scope: an MPhil thesis

Formal approaches

For serious real-world implementations, it is helpful to ground the *implementation strategy* in a formal approach.

- Derive implementation step-by-step from specification
  - That is most-likely the best approach for your lab reports
  - Scope: a few days
- Prove all implementation steps
  - Requires model of the underlying system (compiler, hardware, ...)
  - Scope: an MPhil thesis
- Derive implementation from formal description & hardware model
  - Requires detailed model of hardware
  - Scope: a PhD thesis or research group

Test-driven development (TDD)

- Popularized by Kent Beck
- Write test first, then write code

Test-driven development (TDD)

- Popularized by Kent Beck
- Write test first, then write code
- Some advantages
    - Understanding of edge cases
    - View point of the callsite
    - Motivates testable APIs (mocking)
    - Psychological: gamification, avoiding write-then-challenge

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

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

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**

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**

Fuzzing

Automated testing of programs using randomized input.

Fuzzing

Automated testing of programs using randomized input.

Initial population:

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

Classic fuzzing loop:

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, . . .)

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, . . .)

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

Fuzzing

- Traditionally focused on C-style bugs
  - but can be applied to logic bugs itself given reference

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
    - ...

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)

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])
```

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

Performance optimizations

- Write a **correct** (naïve) implementation first.

Performance optimizations

- Write a **correct** (naïve) implementation first. Celebrate.

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

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

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)
```

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)
```

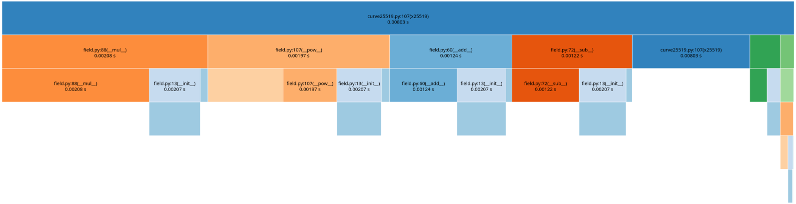
```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001    0.007    0.007 curve25519.py:102(x25519)
  1276    0.001    0.000    0.002    0.000 field.py:77(__mul__)
  1021    0.001    0.000    0.002    0.000 field.py:96(__pow__)
  4596    0.001    0.000    0.002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

Profiling with `cProfile` (Flamegraph)

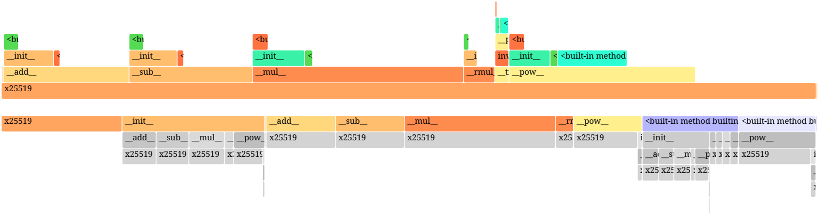
```python
import cProfile, pstats
with cProfile.Profile() as pr:
    x25519(alice_sk, bob_pk)
    pr.disable()
    pr.dump_stats('x25519.prof')
```

Profiling with cProfile (Flamegraph)

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



Profiling with `cProfile` (Flamegraph)



Serialization/marshaling/...

- We already translated the ideal mathematical objects into some representation in our programming language

Serialization/marshaling/...

- We already translated the ideal mathematical objects into some representation in our programming language
- Representing them as byte streams
  - Storage on disk
  - Transport over network
  - ...
- In both cases, there will be two parties: the writer and the reader
  - Could be temporally separate (store now, load later)
  - Could be spatially separate (send over network)

Python's pickle is not great

- A very simple mechanism

Python's pickle is not great

- A very simple mechanism
- Requires both parties to have (roughly) the same class definitions

Python's pickle is not great

- A very simple mechanism
- Requires both parties to have (roughly) the same class definitions
- Allows to customize the serialization/deserialization process
    - Making sure to only store what's needed
    - A wide attack surface

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)
```

Pickle example (2)

```python
import pickle

class Foo: pass

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

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)
```

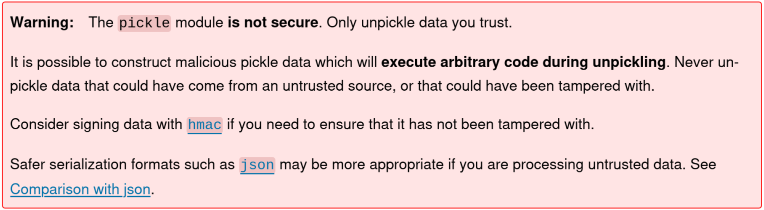
Pickle example (4)

```
$ 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..
```

Pickle example (5)

```
import pickle

class Foo: pass

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

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

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}`

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

Better choices: length-encoding, protobuf, JSON

- Protobuf:
    - Describe types and messages
    - Compiler generates language bindings

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, ...)

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)

Serde example (1)

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

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

Serde example (2)

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

Serde example (3)

```
rmp_serde::to_vec(&msg)?;
// msgp: 81A4546578749 1AC
//       48656C6C6F20776F
//       726C6421
```

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
```

Serde example (5)

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

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

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

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

YAML's country surprise

```
countries-iso:
- SE
- NO
- FI
```

ASN.1

- Abstract Syntax Notation One (ASN.1) is an interface description language
  - Independent of used computer architecture and language

ASN.1

- Abstract Syntax Notation One (ASN.1) is an interface description language
    - Independent of used computer architecture and language
- Used in many specifications such as X.509, LDAP, ...

ASN.1

- Abstract Syntax Notation One (ASN.1) is an interface description language
  - Independent of used computer architecture and language
- Used in many specifications such as X.509, LDAP, ...
- Many features handy for cryptography such as constraints on values and arbitrary-precision integers

ASN.1

- Abstract Syntax Notation One (ASN.1) is an interface description language
  - Independent of used computer architecture and language
- Used in many specifications such as X.509, LDAP, ...
- Many features handy for cryptography such as constraints on values and arbitrary-precision integers
- Supports different encodings
  - Basic Encoding Rules (BER)
  - Distinguished Encoding Rules (DER, subset of BER)
  - XML Encoding Rules

ASN.1 example (1)

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

    OKMessage ::= NULL

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

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)
```

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"}}
```

Popular serialization formats

JSON

- Human readable and flexible
- Few built-in types (no support for, e.g. dates)

Popular serialization formats

JSON	▶ Human readable and flexible ▶ Few built-in types (no support for, e.g. dates)
MessagePack	▶ “Binary JSON encoding” (concise and fast) ▶ Compact encoding and less ambigious encoding

Popular serialization formats

JSON	▶ Human readable and flexible ▶ Few built-in types (no support for, e.g. dates)
MessagePack	▶ “Binary JSON encoding” (concise and fast) ▶ Compact encoding and less ambigious encoding
protobuf	▶ Efficient binary format by Google ▶ Schema required (compiled)

Popular serialization formats

JSON	▶ Human readable and flexible ▶ Few built-in types (no support for, e.g. dates)
MessagePack	▶ “Binary JSON encoding” (concise and fast) ▶ Compact encoding and less ambigious encoding
protobuf	▶ Efficient binary format by Google ▶ Schema required (compiled)
ASN.1	▶ Complex standard for telecom/crypto ▶ Multiple encodings (BER, DER, etc.) ▶ Platform independent

Popular serialization formats

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

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

PSA: Universally Unique Identifier (UUID)

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

- Created to uniquely identify objects (128 bits) and designed to avoid conflicts.

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

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.

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.

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

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

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

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”

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"

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”

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

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

Adversary with compute power (cloud GPU)

Assume:

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

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)

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

Argon2 (RFC 9106)

- Parameters: memory, number of iterations, parallelization

Argon2 (RFC 9106)

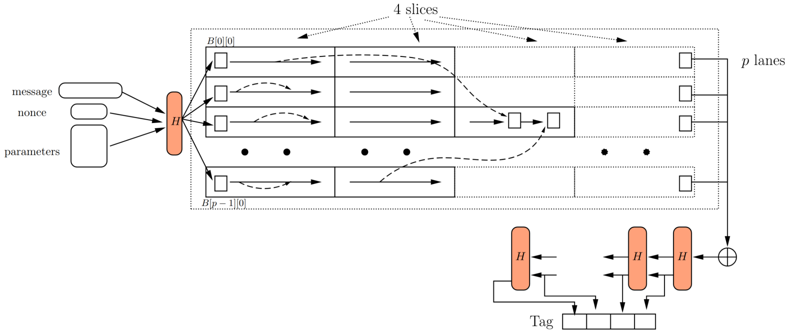
- Parameters: memory, number of iterations, parallelization
- 2+1 version: Argon2d, Argon2i, Argon2id

Argon2 (RFC 9106)

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

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

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

Cambridge HPC

Assume:

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

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:

- $\approx$ 104 billion guesses ($\approx$ 36 bits)

Using generated passphrases

- Adversary makes can guess up to 2^{36} passphrases per day
- We want to protect for at least 1,000 years
- Hence, we need $\log_2(2^{36} \cdot 1000 \cdot 365) \geq 55$ bit
- BIP 39 word list $\rightarrow$ 5 words

"primary boil army core robust"

Forcing strict rate limits

- Using a third-party for evaluation of the hash (e.g. OPAQUE)

Forcing strict rate limits

- Using a third-party for evaluation of the hash (e.g. OPAQUE)
    - Not suitable for local encryption (e.g. full disk encryption)
    - Or where the scheme needs to work offline

Forcing strict rate limits

- Using a third-party for evaluation of the hash (e.g. OPAQUE)
  - Not suitable for local encryption (e.g. full disk encryption)
  - Or where the scheme needs to work offline
- Alternative: use a “built-in third party”
  - Secure Element part of most modern devices (especially smartphones)
  - Resist local attacks

Forcing strict rate limits

- Using a third-party for evaluation of the hash (e.g. OPAQUE)
    - Not suitable for local encryption (e.g. full disk encryption)
    - Or where the scheme needs to work offline
- Alternative: use a “built-in third party”
    - Secure Element part of most modern devices (especially smartphones)
    - Resist local attacks

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

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

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:

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

Approach 1: counting per direction

- Party A counts 0, 2, 4, ... and party B counts 1, 3, 5, ...
- Choose A to be the one with the lexicographical lower DH input

- Simple and collision free
- Difficult to use with $n > 2$ parties and in decentralized settings
- Storage and retry mechanism go into security scope!

The Noise protocol uses 64-bit nonces (to differentiate from random and for compatibility with some ciphers)

Approach 2: nonce-reuse resistant modes

- Using a synthetic initialization vector (SIV) which depends on the nonce and the message
- Example: AES-GCM-SIV (RFC 8452)
- In case of nonce reuse, it is only revealed whether two messages are the same or not.
- Needs two passes over text (no streaming)
- “Collisions” after 2^{32} messages

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:

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

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!

Cryptographic agility

- Idea: allow upgrading the underlying cipher suites
  - Phase out old algorithms
  - Upgrade to new algorithms (e.g. PQC)

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, ...

Case study: JSON Web Token (RFC 7519)

- Given out to clients after authentication
- JOSE Header {`"typ":"JWT", "alg":"HS256"`} + claim
- HS256: HMAC using SHA-256

Case study: JSON Web Token (RFC 7519)

- Given out to clients after authentication
- JOSE Header {`"typ":"JWT", "alg":"HS256"`} + claim
- HS256: HMAC using SHA-256

> *"To support use cases in which the JWT content is secured by a means [. . .] using the "alg" Header Parameter value "none" and with the empty string for its JWS Signature value" – RFC7519*

- In 2020 researchers found that many libraries "correctly" accepted `none` in production systems. . .

Authoritative versioning

- Idea: Each API/protocol endpoint only supports one version
- Advantages:
  - Reduces complexity by having separate endpoints → we can actually delete code!
  - No negotiation required

Authoritative versioning

- Idea: Each API/protocol endpoint only supports one version
- Advantages:
  - Reduces complexity by having separate endpoints → we can actually delete code!
  - No negotiation required
- Migration strategies:
  - Shadow traffic
  - Brown-outs

Authoritative versioning

- Idea: Each API/protocol endpoint only supports one version
- Advantages:
  - Reduces complexity by having separate endpoints → we can actually delete code!
  - No negotiation required
- Migration strategies:
  - Shadow traffic
  - Brown-outs

Opinionated libraries

- Avoid user misconfiguration and wrong usage
  - Limited choice
  - Secure defaults
  - High-level abstractions

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
  - . . .

Pure functions

- All information and state are explicitly passed
    - Input/output parameters
    - Environment: file system, time, ...

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

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!

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

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] }
```

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);
```

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]);
```

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)

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);
```

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

- Microsoft has an isolated production environment for signing final artifacts

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

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

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

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.

Homomorphic Encryption and Private Information Retrieval

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

University of Cambridge
Part III/MPhil in Advanced Computer Science

Post-quantum security

- A post-quantum computer promises efficient solutions to problems underlying classical public-key cryptography using Shor's algorithm
  - Factorize number N in $\mathcal{O}(\mathrm{poly}(\log N))$
  - Solve discrete logarithm in $\mathcal{O}(\mathrm{poly}(\log N))$
  - Adversary can then "quite easily" break encryption techniques (RSA and X25519) relying on these problems

Post-quantum security

- A post-quantum computer promises efficient solutions to problems underlying classical public-key cryptography using Shor's algorithm
    - Factorize number N in $\mathcal{O}(\mathrm{poly}(\log N))$
    - Solve discrete logarithm in $\mathcal{O}(\mathrm{poly}(\log N))$
    - Adversary can then "quite easily" break encryption techniques (RSA and X25519) relying on these problems

- Symmetric algorithms (AES, SHA-3, . . .) and hash functions are believed to be mostly unaffected
    - Grover's algorithm only provides a quadratic speed-up
    - Can be countered by doubling the key size

Post-quantum world

We	**Adversary**	
Classic	Classic	Today’s world

Post-quantum world

We	**Adversary**	
Classic	Classic	Today's world
Quantum	Quantum	Quantum key distribution, . . .
Quantum	Classic	

Post-quantum world

We	**Adversary**	
Classic	Classic	Today's world
Quantum	Quantum	Quantum key distribution, . . .
Quantum	Classic	
Classic	Quantum	**Post-quantum crypto**

Post-quantum world

We	**Adversary**	
Classic	Classic	Today's world
Quantum	Quantum	Quantum key distribution, ...
Quantum	Classic	
Classic	Quantum	**Post-quantum crypto**

Harvest now & decrypt later: an adversary might record interesting data now and await the advent of a practical quantum computer.

Quantum status-quo

$21 = 3 \times 7$

Quantum readiness

- Favouring symmetric schemes
  - Out-of-band key exchanges (e.g. in person, over phone, . . .)
  - One-time-pads
  - . . .

Quantum readiness

- Favouring symmetric schemes
  - Out-of-band key exchanges (e.g. in person, over phone, ...)
  - One-time-pads
  - ...
- Ensure data structures can handle larger key and signature sizes
  - e.g. tree-based log

Quantum readiness

- Favouring symmetric schemes
  - Out-of-band key exchanges (e.g. in person, over phone, ...)
  - One-time-pads
  - ...
- Ensure data structures can handle larger key and signature sizes
  - e.g. tree-based log
- Adopting new algorithms
  - e.g. Kyber KEM, Signal's PQXDH, hash-based signatures
  - Often in a hybrid mode with established techniques

Quantum readiness

- Favouring symmetric schemes
    - Out-of-band key exchanges (e.g. in person, over phone, . . .)
    - One-time-pads
    - . . .
- Ensure data structures can handle larger key and signature sizes
    - e.g. tree-based log
- Adopting new algorithms
    - e.g. Kyber KEM, Signal's PQXDH, hash-based signatures
    - Often in a hybrid mode with established techniques
- Allowing key rotation also for long-term storage

Learning *without* errors (1)

Let's consider the following (overdetermined) equation system over $\mathbb{Z}$:

$$\begin{aligned} 1x + 2y + 3z &= 16 \\ 2x + 3y + 4z &= 27 \\ 3x + 4y + 5z &= 35 \\ 4x + 5y + 6z &= 43 \end{aligned}$$

Learning *without* errors (2)

We can also write it using Matrix-Vector notation and use Gaussian elimination to solve it:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{pmatrix} \times s = \begin{pmatrix} 16 \\ 27 \\ 35 \\ 43 \end{pmatrix}$$

$$\rightarrow s = (1, 3, 4)^\top$$

where $A \in \mathbb{Z}^{4 \times 3}$ and $s \in \mathbb{Z}^3$

Learning *without* errors (3)

The same works perfectly fine in $\mathbb{Z}_q$ (the ring of integers modulo q). For example $q = 7$:

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{pmatrix} \times s = \begin{pmatrix} 2 \\ 6 \\ 0 \\ 1 \end{pmatrix}$$

$$\rightarrow s = (1, 3, 4)^\top$$

where $A \in \mathbb{Z}_7^{4\times 3}$ and $s \in \mathbb{Z}_7^3$

Learning with errors (LWE)

However, adding even a little bit of noise $e \in \chi^4$, i.e. a vector with four independent samples from the random distribution χ, turns this into a tricky problem for which there is no simple solution s.

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \\ 4 & 5 & 6 \end{pmatrix} \times s = \begin{pmatrix} 2 \\ 6 \\ 0 \\ 1 \end{pmatrix} + \begin{pmatrix} +0.2 \\ -0.2 \\ +0.0 \\ +0.2 \end{pmatrix} = \begin{pmatrix} 2.2 \\ 5.8 \\ 0.0 \\ 1.2 \end{pmatrix}$$

Learning with errors (LWE)

Let $\mathbb{Z}_q$ be the ring of integers modulo q. Let n and m be positive integers that we choose depending on our security parameter. Let χ be an error distribution over $\mathbb{Z}_q$ with values $[-B, B]$, where B is much smaller than q.

The decisional **Learning With Errors** (LWE) problem is to distinguish between the following two distributions:

- $(A, A \cdot s + e)$ where $A \in \mathbb{Z}_q^{m \times n}$, $s \in \mathbb{Z}_q^n$, and $e \in \chi^m$

Learning with errors (LWE)

Let $\mathbb{Z}_q$ be the ring of integers modulo q. Let n and m be positive integers that we choose depending on our security parameter. Let χ be an error distribution over $\mathbb{Z}_q$ with values $[-B, B]$, where B is much smaller than q.

The decisional **Learning With Errors** (LWE) problem is to distinguish between the following two distributions:

- $(A, A \cdot s + e)$ where $A \in \mathbb{Z}_q^{m \times n}$, $s \in \mathbb{Z}_q^n$, and $e \in \chi^m$
- (A, U) where $A \in \mathbb{Z}_q^{m \times n}$ and $U \in \mathbb{Z}_q^m$ uniformly random

Notation: we use $c = A \cdot s + e$ to denote the "LWE sample".

LWE: symmetric encryption (one-bit message)

Let's consider the message space $\mathcal{M} = \{0, 1\}$, i.e. a single bit message and therefore we set $m = 1$. We want to build a symmetric encryption scheme (Gen, Enc, Dec) to encrypt the message $b \in \mathcal{M}$.

- Let $s \in \mathbb{Z}_q^n$ be the secret key
- Let $A \in \mathbb{Z}_q^{n \times 1}$ be the public matrix
- Let $e \in \chi^1$ be the error vector with $B < \lfloor q/4 \rfloor$

We define encryption and decryption as:

$$Enc_s(b) = (A, A \cdot s + e + b \cdot \lfloor q/2 \rfloor)$$

$$Dec_s(c) = \begin{cases} 1 & \text{if } \lfloor q/4 \rfloor \leq c - A \cdot s < 3 \cdot \lfloor q/4 \rfloor \\ 0 & \text{otherwise} \end{cases}$$

LWE: symmetric encryption

We can extend this scheme to encrypt multi-bit messages by using a matrix A with m columns, where m is the bit length of the message.

- Let $s \in \mathbb{Z}_q^n$ be the secret key
- Let $A \in \mathbb{Z}_q^{n \times m}$ be the public matrix
- Let $e \in \chi^m$ be the error vector with $B < \lfloor q/4 \rfloor$

We define encryption and decryption as:

$$Enc_s(b) = (A, A \cdot s + e + b \cdot \lfloor q/2 \rfloor)$$

$$Dec_s(c)_i = \begin{cases} 1 & \text{if } \lfloor q/4 \rfloor \leq c_i - (A \cdot s)_i < 3 \cdot \lfloor q/4 \rfloor \\ 0 & \text{otherwise} \end{cases}$$

Homomorphic encryption

A public-key encryption scheme (Gen, Enc, Dec) is homomorphic if for all (pk, sk), it is possible to define groups $\mathcal{M}$, $\mathcal{C}$ (depending on pk only), with group operators $\oplus$ and $\otimes$ respectively, such that:

- The message space is $\mathcal{M}$ and all ciphertexts output by Enc_{pk} are elements of $\mathcal{C}$
- For any $m_1, m_2 \in \mathcal{M}$ and $c_1 \leftarrow \mathsf{Enc}_{pk}(m_1)$, $c_2 \leftarrow \mathsf{Enc}_{pk}(m_2)$ it holds that:

$$\mathsf{Dec}_{sk}(c_1 \otimes c_2) = m_1 \oplus m_2$$

- The distribution of all ciphertexts obtained by applying $\otimes$ to any ciphertexts $\mathsf{Enc}_{pk}(m_1)$, $\mathsf{Enc}_{pk}(m_2)$ is identical to the distribution of $\mathsf{Enc}_{pk}(m_1 + m_2)$

Symmetric LWE is homomorphic

LWE is homomorphic under addition, i.e.

$$Dec(s, Enc(m_1) + Enc(m_2)) = m_1 \oplus m_2$$

Symmetric LWE is homomorphic

LWE is homomorphic under addition, i.e.

$$Dec(s, Enc(m_1) + Enc(m_2)) = m_1 \oplus m_2$$

We can convince ourselves of this by looking at the definition of encryption:

$$Enc(m_1) = (A, A \cdot s + e_1 + m_1 \cdot \lfloor q/2 \rfloor)$$
$$Enc(m_2) = (A, A \cdot s + e_2 + m_2 \cdot \lfloor q/2 \rfloor)$$

Symmetric LWE is homomorphic

LWE is homomorphic under addition, i.e.

$$Dec(s, Enc(m_1) + Enc(m_2)) = m_1 \oplus m_2$$

We can convince ourselves of this by looking at the definition of encryption:

$$Enc(m_1) = (A, A \cdot s + e_1 + m_1 \cdot \lfloor q/2 \rfloor)$$
$$Enc(m_2) = (A, A \cdot s + e_2 + m_2 \cdot \lfloor q/2 \rfloor)$$

$$\begin{aligned} &Enc(m_1) + Enc(m_2) \qquad\qquad\qquad\qquad = \\ &(A + A,\ (A + A) \cdot s + (e_1 + e_2) + (m_1 + m_2) \cdot \lfloor q/2 \rfloor) \end{aligned}$$

Symmetric LWE is homomorphic

LWE is homomorphic under addition, i.e.

$$Dec(s, Enc(m_1) + Enc(m_2)) = m_1 \oplus m_2$$

We can convince ourselves of this by looking at the definition of encryption:

$$Enc(m_1) = (A, A \cdot s + e_1 + m_1 \cdot \lfloor q/2 \rfloor)$$
$$Enc(m_2) = (A, A \cdot s + e_2 + m_2 \cdot \lfloor q/2 \rfloor)$$

$$Enc(m_1) + Enc(m_2) =$$
$$(A + A,\ (A + A) \cdot s + (e_1 + e_2) + (m_1 + m_2) \cdot \lfloor q/2 \rfloor)$$

Note: the noise in the ciphertext grows (doubles) with each operation: $e_{new} = e_1 + e_2$

Private information retrieval (PIR)

We want to build a protocol for private information retrieval (PIR).

Problem:

- Server has a database $D \in \{0, 1\}^N$ with N one-bit entries
- Client wants to retrieve the item at index $i \in [N]$

Private information retrieval (PIR)

We want to build a protocol for private information retrieval (PIR).

Problem:

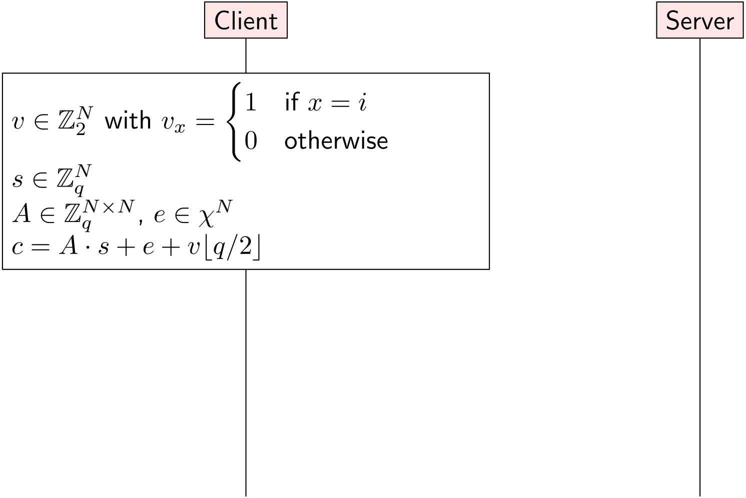
- Server has a database $D \in \{0,1\}^N$ with N one-bit entries
- Client wants to retrieve the item at index $i \in [N]$

Requirements:

- Correctness: Client should learn D_i
- Privacy: Server does not learn which i was requested
- Minimal communication: They must not exchange more than N bits

PIR using LWE (naïve)

Alice wants to retrieve the i-th entry from a database $D \in \{0,1\}^N$ where $i \in [N]$.



$$v \in \mathbb{Z}_2^N \text{ with } v_x = \begin{cases} 1 & \text{if } x = i \\ 0 & \text{otherwise} \end{cases}$$

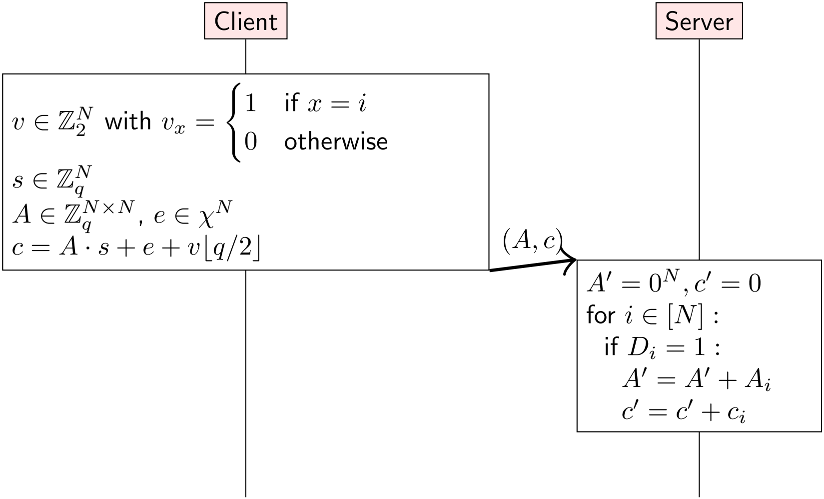
$s \in \mathbb{Z}_q^N$

$A \in \mathbb{Z}_q^{N \times N}$, $e \in \chi^N$

$c = A \cdot s + e + v\lfloor q/2 \rfloor$

PIR using LWE (naïve)

Alice wants to retrieve the i-th entry from a database $D \in \{0,1\}^N$ where $i \in [N]$.



Client

$$v \in \mathbb{Z}_2^N \text{ with } v_x = \begin{cases} 1 & \text{if } x = i \\ 0 & \text{otherwise} \end{cases}$$

$s \in \mathbb{Z}_q^N$

$A \in \mathbb{Z}_q^{N \times N}$, $e \in \chi^N$

$c = A \cdot s + e + v\lfloor q/2 \rfloor$

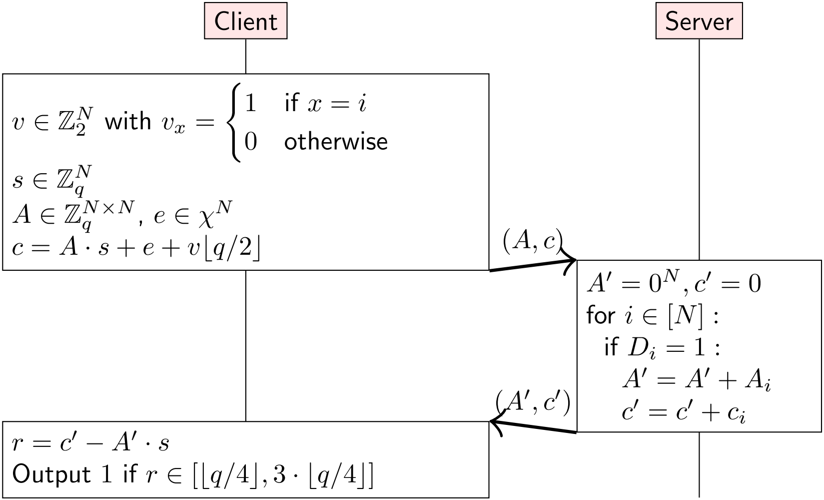
(A, c)

Server

```
$A' = 0^N, c' = 0$
for $i \in [N]$ :
  if $D_i = 1$ :
    $A' = A' + A_i$
    $c' = c' + c_i$
```

PIR using LWE (naïve)

Alice wants to retrieve the i-th entry from a database $D \in \{0,1\}^N$ where $i \in [N]$.



Client

Server

$$v \in \mathbb{Z}_2^N \text{ with } v_x = \begin{cases} 1 & \text{if } x = i \\ 0 & \text{otherwise} \end{cases}$$

$s \in \mathbb{Z}_q^N$

$A \in \mathbb{Z}_q^{N \times N}$, $e \in \chi^N$

$c = A \cdot s + e + v \lfloor q/2 \rfloor$

(A, c)

```
$A' = 0^N, c' = 0$
for $i \in [N]$ :
  if $D_i = 1$ :
    $A' = A' + A_i$
    $c' = c' + c_i$
```

(A', c')

$r = c' - A' \cdot s$

Output 1 if $r \in [\lfloor q/4 \rfloor, 3 \cdot \lfloor q/4 \rfloor]$

PIR using LWE (naïve)

- Privacy: the server only ever sees the LWE sample $(A, c) \in \mathbb{Z}_q^{N \times N} \times \mathbb{Z}_q^N$. Reduction to the LWE assumption shows that it does not learn i

PIR using LWE (naïve)

- Privacy: the server only ever sees the LWE sample $(A, c) \in \mathbb{Z}_q^{N \times N} \times \mathbb{Z}_q^N$. Reduction to the LWE assumption shows that it does not learn i
- Correctness:
    - We add c_i where $D_i = 1$
    - Only one of the c_i encrypts 1, hence we only “add” $\lfloor q/2 \rfloor$ in that case
    - The result c' contains the noise $N \cdot e$

PIR using LWE (naïve)

- Privacy: the server only ever sees the LWE sample $(A, c) \in \mathbb{Z}_q^{N \times N} \times \mathbb{Z}_q^N$. Reduction to the LWE assumption shows that it does not learn i
- Correctness:
  - We add c_i where $D_i = 1$
  - Only one of the c_i encrypts 1, hence we only “add” $\lfloor q/2 \rfloor$ in that case
  - The result c' contains the noise $N \cdot e$
- Minimal communication:
  - $|A| = N^2 \log_2 q$ bits
  - $|c| = N \log_2 q$ bits
  - Let’s improve this!

PIR using LWE (square-root)

Alice wants to retrieve the (i, j)-th entry from a database $D \in \{0,1\}^{\sqrt{N} \times \sqrt{N}}$ where $i, j \in [\sqrt{N}]$.

Client

Server

$v \in \mathbb{Z}_2^{\sqrt{N}}$ with $v_x = \begin{cases} 1 & \text{if } x = j \\ 0 & \text{otherwise} \end{cases}$

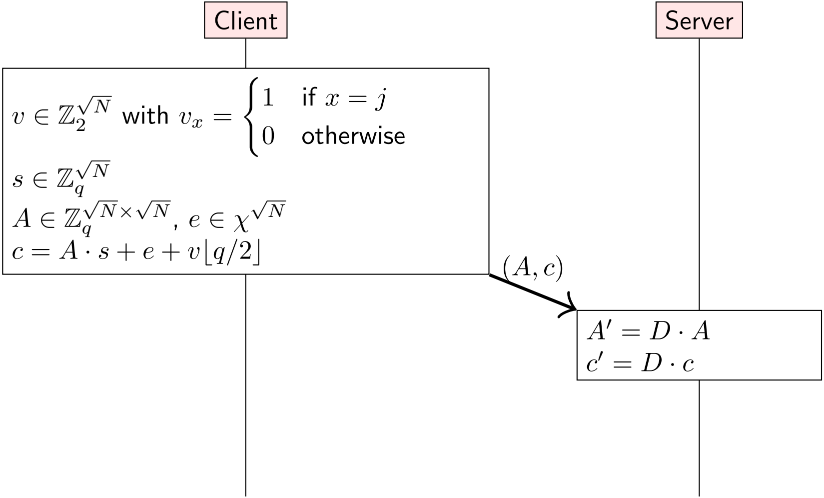
$s \in \mathbb{Z}_q^{\sqrt{N}}$

$A \in \mathbb{Z}_q^{\sqrt{N} \times \sqrt{N}}$, $e \in \chi^{\sqrt{N}}$

$c = A \cdot s + e + v\lfloor q/2 \rfloor$

PIR using LWE (square-root)

Alice wants to retrieve the (i, j)-th entry from a database $D \in \{0,1\}^{\sqrt{N} \times \sqrt{N}}$ where $i, j \in [\sqrt{N}]$.



Client

Server

$$v \in \mathbb{Z}_2^{\sqrt{N}} \text{ with } v_x = \begin{cases} 1 & \text{if } x = j \\ 0 & \text{otherwise} \end{cases}$$

$s \in \mathbb{Z}_q^{\sqrt{N}}$

$A \in \mathbb{Z}_q^{\sqrt{N} \times \sqrt{N}}$, $e \in \chi^{\sqrt{N}}$

$c = A \cdot s + e + v\lfloor q/2 \rfloor$

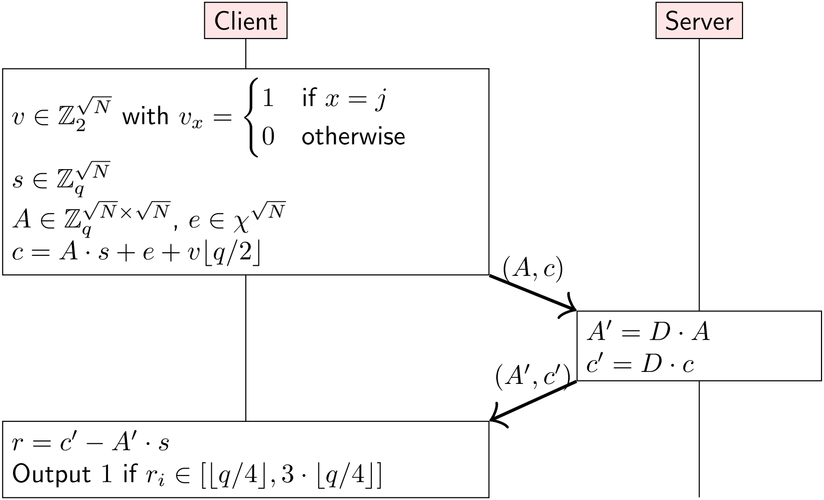
(A, c)

$A' = D \cdot A$

$c' = D \cdot c$

PIR using LWE (square-root)

Alice wants to retrieve the (i,j)-th entry from a database $D \in \{0,1\}^{\sqrt{N}\times\sqrt{N}}$ where $i,j \in [\sqrt{N}]$.



Client

Server

$v \in \mathbb{Z}_2^{\sqrt{N}}$ with $v_x = \begin{cases} 1 & \text{if } x = j \\ 0 & \text{otherwise} \end{cases}$

$s \in \mathbb{Z}_q^{\sqrt{N}}$

$A \in \mathbb{Z}_q^{\sqrt{N}\times\sqrt{N}}$, $e \in \chi^{\sqrt{N}}$

$c = A \cdot s + e + v\lfloor q/2 \rfloor$

(A, c)

$A' = D \cdot A$

$c' = D \cdot c$

(A', c')

$r = c' - A' \cdot s$

Output 1 if $r_i \in [\lfloor q/4 \rfloor, 3 \cdot \lfloor q/4 \rfloor]$

PIR using LWE (square-root)

- Privacy: the server only sees the blinded LWE sample (A', c')

PIR using LWE (square-root)

- Privacy: the server only sees the blinded LWE sample (A', c')
- Correctness:
  - The unit vector v effectively "selects" the i-th column
  - The response vector c' has one entry per entry of the i-th column of D
  - Each element of the response c' contains noise $\sqrt{N} \cdot e$

PIR using LWE (square-root)

- Privacy: the server only sees the blinded LWE sample (A', c')
- Correctness:
  - The unit vector v effectively “selects” the i-th column
  - The response vector c' has one entry per entry of the i-th column of D
  - Each element of the response c' contains noise $\sqrt{N} \cdot e$
- Minimal communication:
  - $|A| = \sqrt{N}^2 \log_2 q = N \log_2 q$ bits
  - $|c| = \sqrt{N} \log_2 q$ bits
  - Better, but not great yet. . .

PIR using LWE: improvements

We can make a few observations that allow us to improve the scheme:

- Since A is already transmitted in the clear, we can use it for multiple queries without having to re-transmit it.
- Similarly, the server can pre-compute $A' = D \cdot A$.
- And in turn A' can be distributed to clients ahead of time (barring any updates to D)

PIR using LWE (optimized square-root)

Alice wants to retrieve the (i, j)-th entry from a database $D \in \{0,1\}^{\sqrt{N} \times \sqrt{N}}$ where $i, j \in [\sqrt{N}]$. Both have pre-computed and shared A and $A' = D \cdot A$.

Client

Server

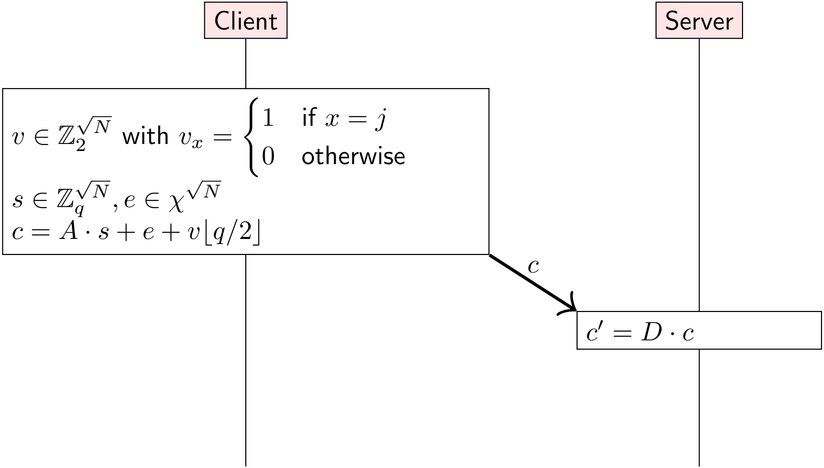
$v \in \mathbb{Z}_2^{\sqrt{N}}$ with $v_x = \begin{cases} 1 & \text{if } x = j \\ 0 & \text{otherwise} \end{cases}$

$s \in \mathbb{Z}_q^{\sqrt{N}}, e \in \chi^{\sqrt{N}}$

$c = A \cdot s + e + v\lfloor q/2 \rfloor$

PIR using LWE (optimized square-root)

Alice wants to retrieve the (i, j)-th entry from a database $D \in \{0,1\}^{\sqrt{N}\times\sqrt{N}}$ where $i, j \in [\sqrt{N}]$. Both have pre-computed and shared A and $A' = D \cdot A$.



Client

Server

$$v \in \mathbb{Z}_2^{\sqrt{N}} \text{ with } v_x = \begin{cases} 1 & \text{if } x = j \\ 0 & \text{otherwise} \end{cases}$$

$$s \in \mathbb{Z}_q^{\sqrt{N}}, e \in \chi^{\sqrt{N}}$$

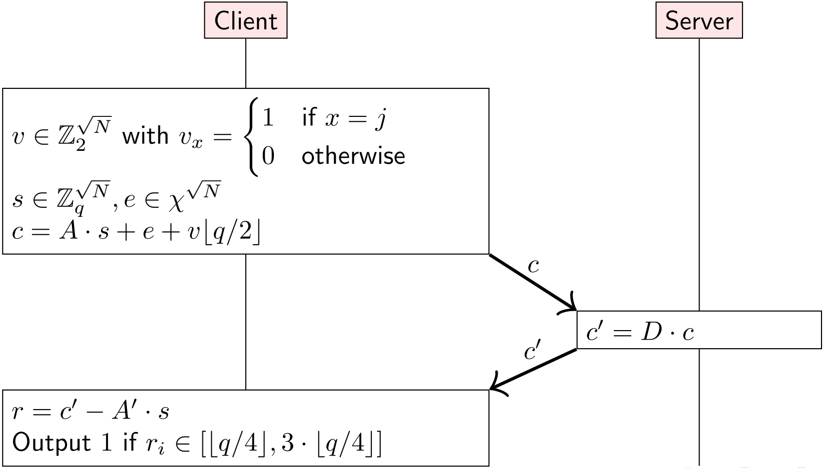
$$c = A \cdot s + e + v\lfloor q/2 \rfloor$$

$$\xrightarrow{c}$$

$$c' = D \cdot c$$

PIR using LWE (optimized square-root)

Alice wants to retrieve the (i, j)-th entry from a database $D \in \{0,1\}^{\sqrt{N} \times \sqrt{N}}$ where $i, j \in [\sqrt{N}]$. Both have pre-computed and shared A and $A' = D \cdot A$.



Client

Server

$$v \in \mathbb{Z}_2^{\sqrt{N}} \text{ with } v_x = \begin{cases} 1 & \text{if } x = j \\ 0 & \text{otherwise} \end{cases}$$

$s \in \mathbb{Z}_q^{\sqrt{N}}, e \in \chi^{\sqrt{N}}$

$c = A \cdot s + e + v\lfloor q/2 \rfloor$

c

$c' = D \cdot c$

c'

$r = c' - A' \cdot s$

Output 1 if $r_i \in [\lfloor q/4 \rfloor, 3 \cdot \lfloor q/4 \rfloor]$

Assignment 3

Implement a PIR scheme using symmetric LWE encryption.

- Users should be able to query 1-bit fields from a database with more than 1,000 entries
- Simulate the network by exchanging serialized messages
- Note: to our knowledge, there are no relevant standards/specifications. You will have to make your own decisions.
- Deadline: 24 Mar 2025

LWE Public-key Encryption (1)

We recall our example from the previous slides and represent it as an augmented $m \times (n+1)$ matrix P:

$$P = (A \mid A \cdot s + e) = \left(\begin{array}{ccc|c} 1 & 2 & 3 & 2.2 \\ 2 & 3 & 4 & 5.8 \\ 3 & 4 & 5 & 0.0 \\ 4 & 5 & 6 & 1.2 \end{array} \right)$$

LWE Public-key Encryption (2)

We sample a random vector $r \in \{0,1\}^m$ and compute $r \cdot P$.

$$r \cdot P = \begin{pmatrix} 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 2.2 \\ 2 & 3 & 4 & 5.8 \\ 3 & 4 & 5 & 0.0 \\ 4 & 5 & 6 & 1.2 \end{pmatrix} = \begin{pmatrix} 6 & 8 & 10 & 7 \end{pmatrix}$$

LWE Public-key Encryption (2)

We sample a random vector $r \in \{0,1\}^m$ and compute $r \cdot P$.

$$r \cdot P = \begin{pmatrix} 0 & 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 2.2 \\ 2 & 3 & 4 & 5.8 \\ 3 & 4 & 5 & 0.0 \\ 4 & 5 & 6 & 1.2 \end{pmatrix} = \begin{pmatrix} 6 & 8 & 10 & 7 \end{pmatrix}$$

With s we can distinguish the ciphertext c from a random vector:

$$(r \cdot P) \cdot \begin{pmatrix} s_0 \\ \vdots \\ s_{n-1} \\ -1 \end{pmatrix} = \begin{pmatrix} 6 & 8 & 10 & 7 \end{pmatrix} \begin{pmatrix} 1 \\ 3 \\ 4 \\ -1 \end{pmatrix} = 63 \equiv 0 \pmod{q}$$

LWE Public-key Encryption (3)

Let $x \in \{0,1\}$ and $r \in \{0,1\}^m$. We can then define the encryption of x as:

$$c = r \cdot P + (0, \ldots, 0, x \cdot \lfloor q/2 \rfloor)$$

The recipient can then use their secret key s to decrypt the ciphertext by computing:

$$\mathrm{Dec}(P, s, c) = \begin{cases} 1 & \text{if } \lfloor q/4 \rfloor \leq c - (P \cdot \hat{s}) < 3 \cdot \lfloor q/4 \rfloor \\ 0 & \text{otherwise} \end{cases}$$

where $\hat{s} = (s^T \mid -1)^T$ i.e. the column vector s augmented with -1 in the last position.

LWE Public-key Encryption Parameters

Let λ be our security parameter. Then let $n = \lambda$ and $m = \Theta(n \cdot \log q)$. Let χ be a discrete Gaussian distribution bounded to $[-B, B]$ such that $B \cdot m < q/4$. Then the scheme from the previous slide is a semantically secure public-key encryption scheme.

LWE Public-key Encryption Parameters

Let λ be our security parameter. Then let $n = \lambda$ and $m = \Theta(n \cdot \log q)$. Let χ be a discrete Gaussian distribution bounded to $[-B, B]$ such that $B \cdot m < q/4$. Then the scheme from the previous slide is a semantically secure public-key encryption scheme.

- A typical choice for q is 2^{16}

LWE Public-key Encryption Parameters

Let λ be our security parameter. Then let $n = \lambda$ and $m = \Theta(n \cdot \log q)$. Let χ be a discrete Gaussian distribution bounded to $[-B, B]$ such that $B \cdot m < q/4$. Then the scheme from the previous slide is a semantically secure public-key encryption scheme.

- A typical choice for q is 2^{16}
- For 128-bit security, we choose $n = 2^7$

LWE Public-key Encryption Parameters

Let λ be our security parameter. Then let $n = \lambda$ and $m = \Theta(n \cdot \log q)$. Let χ be a discrete Gaussian distribution bounded to $[-B, B]$ such that $B \cdot m < q/4$. Then the scheme from the previous slide is a semantically secure public-key encryption scheme.

- A typical choice for q is 2^{16}
- For 128-bit security, we choose $n = 2^7$
- With $m = 2n \cdot \log_2 q$ we get $m = 2^{11}$

LWE Public-key Encryption Parameters

Let λ be our security parameter. Then let $n = \lambda$ and $m = \Theta(n \cdot \log q)$. Let χ be a discrete Gaussian distribution bounded to $[-B, B]$ such that $B \cdot m < q/4$. Then the scheme from the previous slide is a semantically secure public-key encryption scheme.

- A typical choice for q is 2^{16}
- For 128-bit security, we choose $n = 2^7$
- With $m = 2n \cdot \log_2 q$ we get $m = 2^{11}$
- Remains to choose B such that $B \cdot m < q/4$

LWE Public-key Encryption Parameters

Let λ be our security parameter. Then let $n = \lambda$ and $m = \Theta(n \cdot \log q)$. Let χ be a discrete Gaussian distribution bounded to $[-B, B]$ such that $B \cdot m < q/4$. Then the scheme from the previous slide is a semantically secure public-key encryption scheme.

- A typical choice for q is 2^{16}
- For 128-bit security, we choose $n = 2^7$
- With $m = 2n \cdot \log_2 q$ we get $m = 2^{11}$
- Remains to choose B such that $B \cdot m < q/4$
  - Higher B gives better security but the noise growth is higher
  - Lower B gives less security and we might need to increase n

Improving LWE Public-key Encryption

Problem: The resulting public key P is large: $n \cdot m$ elements in $\mathbb{Z}_q$. The ciphertexts c has n elements.

- Derive the matrix A pseudorandomly from a short seed.
  - Generate the entries using $H(\text{seed} \parallel i \parallel j)$.

Improving LWE Public-key Encryption

Problem: The resulting public key P is large: $n \cdot m$ elements in $\mathbb{Z}_q$. The ciphertexts c has n elements.

- Derive the matrix A pseudorandomly from a short seed.
  - Generate the entries using $H(\text{seed} \parallel i \parallel j)$.
- Pack more bits into each ciphertext by using $\mathcal{M} = \mathbb{Z}_p$ with $p > 2$
  - Encode $m \in \mathcal{M}$ as $m \cdot (q/p)$
  - During decryption, find nearest multiple of (q/p)

Improving LWE Public-key Encryption

Problem: The resulting public key P is large: $n \cdot m$ elements in $\mathbb{Z}_q$. The ciphertexts c has n elements.

- Derive the matrix A pseudorandomly from a short seed.
    - Generate the entries using $H(\text{seed} \parallel i \parallel j)$.
- Pack more bits into each ciphertext by using $\mathcal{M} = \mathbb{Z}_p$ with $p > 2$
    - Encode $m \in \mathcal{M}$ as $m \cdot (q/p)$
    - During decryption, find nearest multiple of (q/p)
- Use the LWE over rings assumption (RLWE)
    - Reduces key and ciphertext sizes
    - Also reduces required computation
    - Used in practical schemes