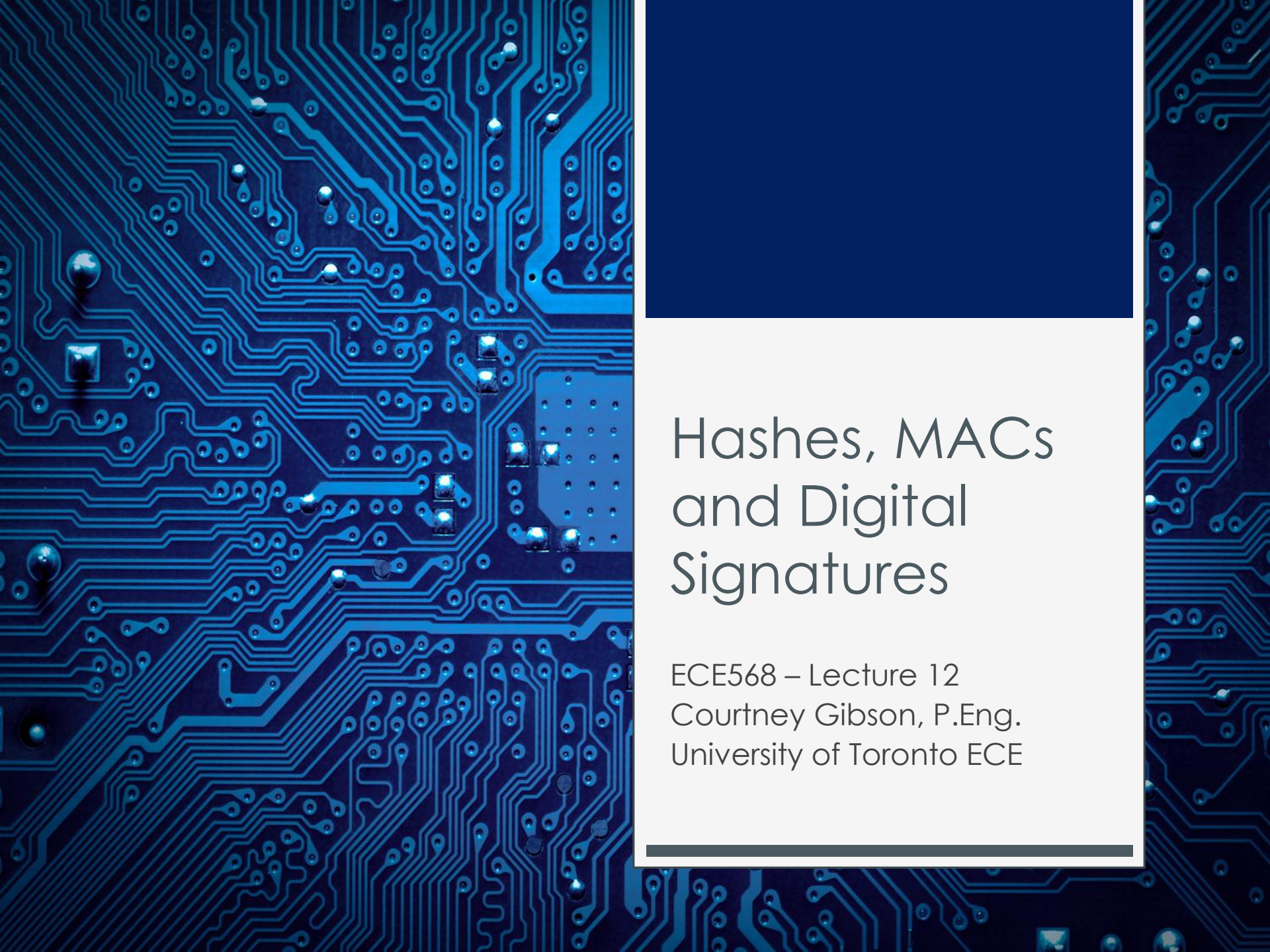


Hashes, MACs and Digital Signatures

ECE568 – Lecture 12

Courtney Gibson, P.Eng.
University of Toronto ECE

Outline

Integrity and Authentication

Cryptographic Hashes

Modification Detection Codes (MDCs)

Message Authentication Codes (MACs)

Digital Signatures

- X509 Certificates

Integrity/Authentication

Does **encryption** provide integrity and authentication?

- On decrypting an encrypted message, is a receiver assured that the message has not been tampered with (integrity) and arrived from the intended source (authentication)?
- **This is a common misconception, it is NOT true!**

Integrity/Authentication

Even if the encryption key is not known, an attacker can:

- Insert random data at the start/end of the message
- Replace the entire message with random data
- Replay previous messages
- Reorder blocks if using ECB mode
- Flip bits with stream ciphers

If the data decrypts to something sensible, the receiver does not know it was tampered with

Cryptographic Hashes

Hashes

Hashes are a fundamental cryptographic tool for providing integrity and authentication

- Hashes are used as part of:
  - Modification Detection Codes (MDC) to provide integrity
  - Message Authentication Codes (MAC) to provide integrity and authentication
  - Digital signatures to provide integrity, authentication and non-repudiation
- A **hash function** converts a large input into a smaller (typically fixed size) output, **H(m) = h**
  - Input **m** is called the data **preimage**
  - Output **h** is called **hash value** or **message digest**
  - **H()** is a lossy compression function

Cryptographic Hashes

A cryptographic hash function has three desirable properties:

- **Preimage Resistance:** Given a hash value, it is hard to find a preimage that will yield the hash value (*i.e.*, it is hard to reverse the hash function)
  - Given a hash value **h** and function **H**, it is hard to find **m** such that **H(m) = h**
- **2nd Preimage Resistance:** Given a preimage, it is hard to find another preimage that hashes to the same hash value
  - Given **m**, it is hard to find an **m'**, such that **H(m) = H(m')**
- **Collision Resistance:** It should be hard to find collisions (*i.e.*, two pre-image values that coincidentally hash to the same hash value)
  - Hard to find **any** two **m** and **m'** such that **H(m) = H(m')**

Example

Using the SHA1 algorithm:

```
$ echo "Cryptographic hash values are like fingerprints" | shasum
d05b4ffc0677f1c5811dae6d7b914c2b60578d48
$ echo "cryptographic hash values are like fingerprints" | shasum
62e18bbb87c8e894dc3c73cc62ae6006d73bbe06
```

shasum takes any input and produces a fixed-length hash value. Note that very small changes in the preimage (the first character) produce a very different hash value.

Hash Length

Assuming a hash function has the three required properties, it can be treated as an **ideal hash**. The security of an ideal hash depends entirely on the length of the hash value.

- If the length of the hash is **n** bits, then:
  - **2^{nd} Preimage Resistance**
    - The expected number of guesses to find another pre-image that hashes to a given hash value is 2^{n-1}
  - **Collision resistance**
    - The expected number of tries to find any two pre-images that hash to the same value is $2^{n \div 2}$
    - This is often called a **birthday attack**

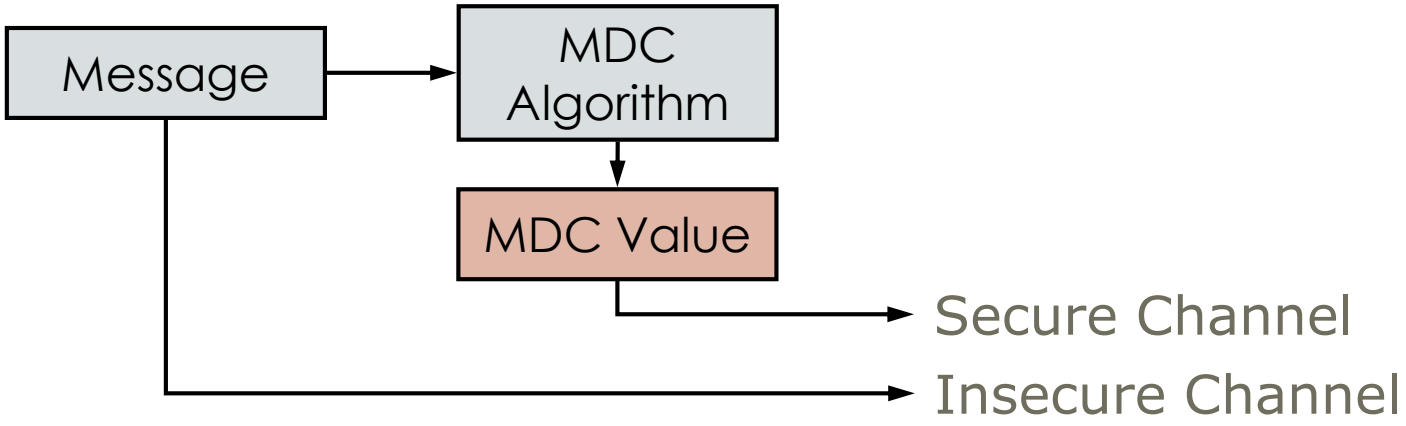
MDC

Modification Detection Codes

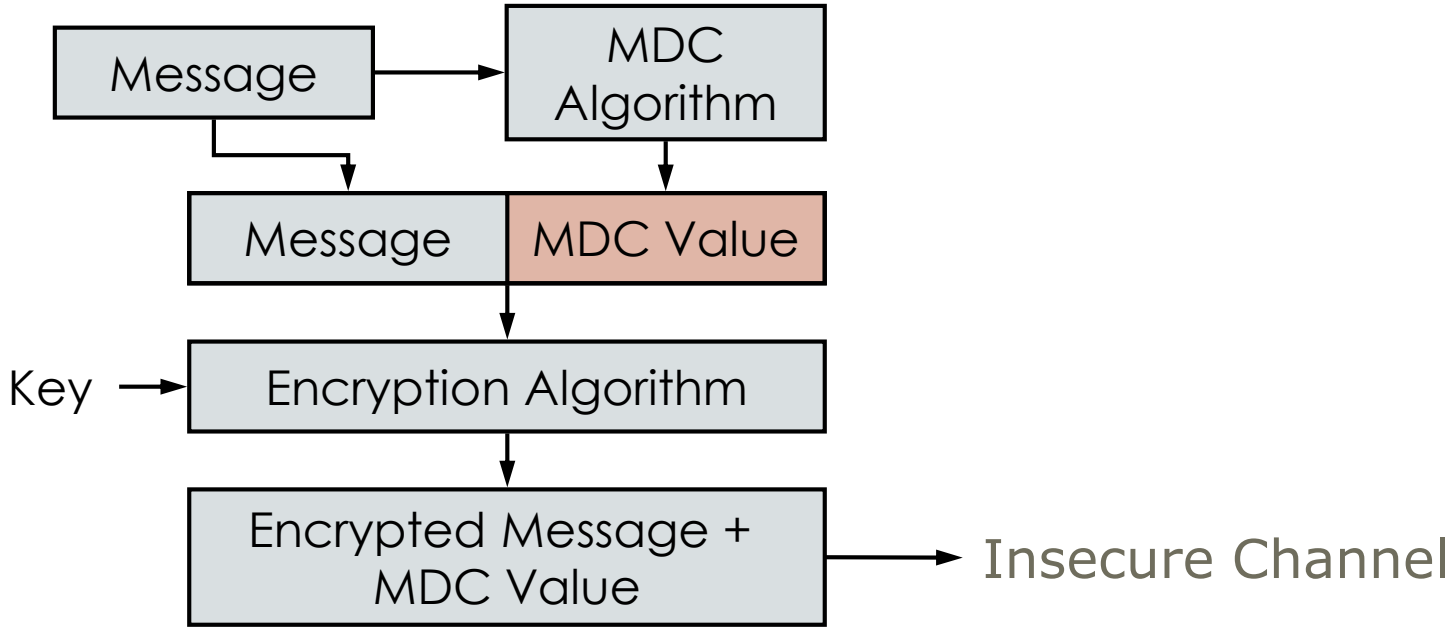
Modification Detection Code

- Modification Detection Code (MDC) use hashes to provide **integrity**
  - *e.g.*, file downloads
  - Also called Message Integrity Code (MIC)
- Taking a hash of a message and sending the hash and the message separately allows the receiver to detect if the message has been modified in transit

Using MDC Securely



- MDC with a secure channel provides integrity
- Remember that the purpose of the MDC is to allow the receiver to verify the integrity of the message
  - It does not protect message confidentiality
  - If confidentiality is required, the message should be encrypted separately



- MDC with encryption provides confidentiality, integrity and authentication
- After decryption, the receiver can verify the source and the integrity of the message by checking that the MDC value matches the message

Modification Detection Codes

The two commonly used MDC are MD5 and SHA1:

- **MD5**
  - Created by Ron Rivest at RSA
  - Produces a 128-bit hash value from an arbitrarily large input
  - MD5 has been broken
- **SHA1**
  - Created by NIST with the help of the NSA
  - Produces a 160-bit hash
  - SHA1 has demonstrated weaknesses
- **SHA256 (or SHA2)**
  - A successor of SHA1
  - Produces a 256-bit hash
  - SHA256 is considered strong, but is not widely used because it is not in many standards, but that is changing

Description of SHA1 and MD5

Both are Iterated functions, much like AES and DES

- **SHA1**:
  - Hashes 512 bits of a message (a block) at a time
  - Each block is passed through 4 rounds of operations
  - Each round uses 20 operations to update a 160-bit state
  - After a block is processed, SHA1 outputs its 160-bit state, which is then used as an input for hashing the next block
- **MD5** is similar, 4 rounds, 16 operations per round

MAC

Message Authentication Codes

Message Authentication Code

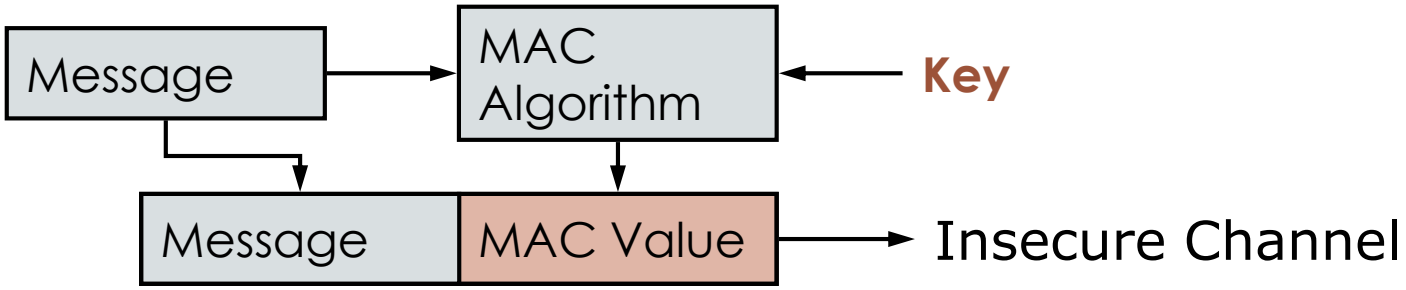
A Message Authentication Code (MAC) uses a hash to provide **integrity** and **authentication**

- A MAC is constructed as **h = H(k, M)**, where **k** is secret key and **M** is the message
- The hash function can also take a key as a separate input, and the output becomes dependent on both the key and the message

Receiver knows that whoever generated the MAC must also know the key, thus authenticating the message source

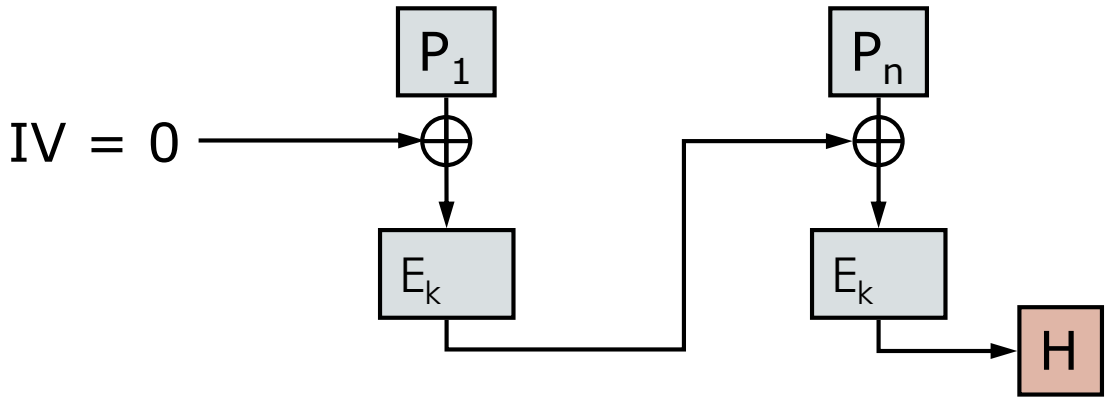
Using MACs Securely

- MACs can be used to provide integrity and authentication (without confidentiality)
- The receiver can use the shared key to verify that the MAC matches the message, thus authenticating the source of the message as well as the contents
- Note that the message is not encrypted so its confidentiality is not protected



MAC using Symmetric Ciphers

- Often constructed from symmetric ciphers
- A common construction is CBC-MAC
  - The structure is similar to CBC encryption for block ciphers except a single hash value is produced at the end
  - Hash size is the same as block size of block cipher
  - MAC key must be different from encryption key (why?)



MAC using Hashes

A MAC can also be constructed by concatenating the secret key with the message and using a hash, which creates a **keyed-Hash MAC** or **HMAC**

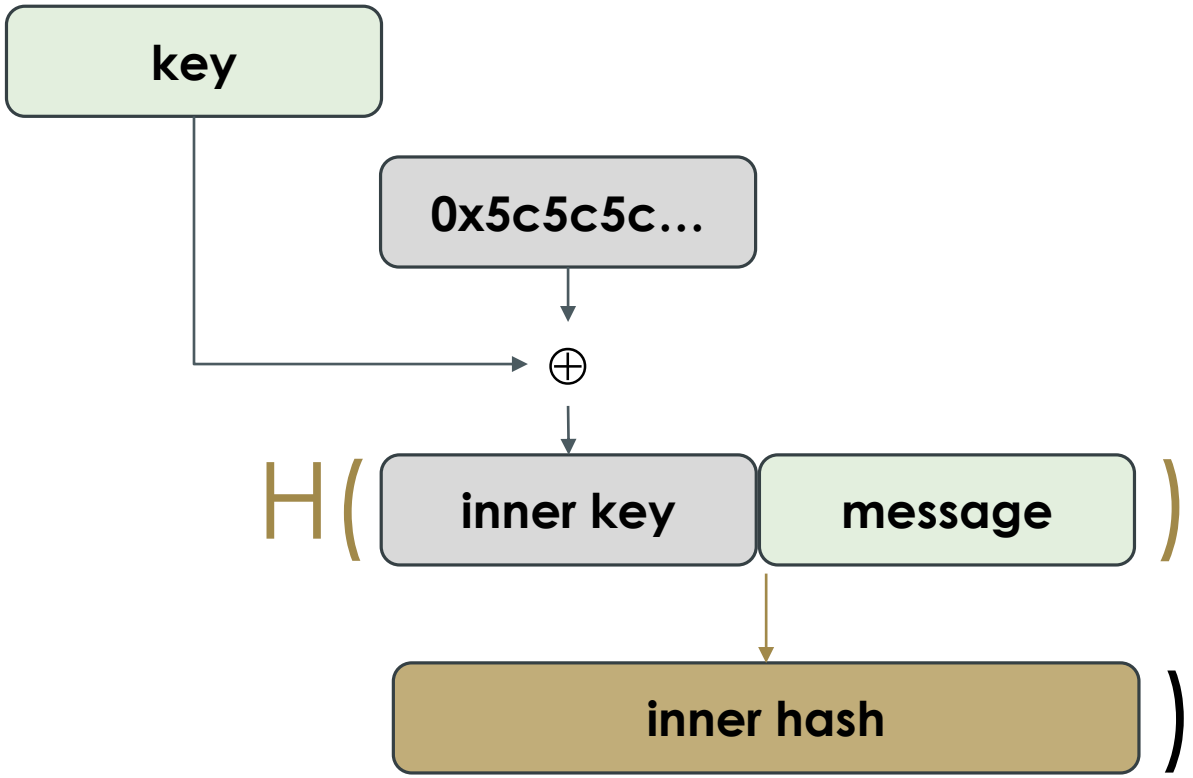
key

H(**message**)

HMAC = H( )

MAC using Hashes

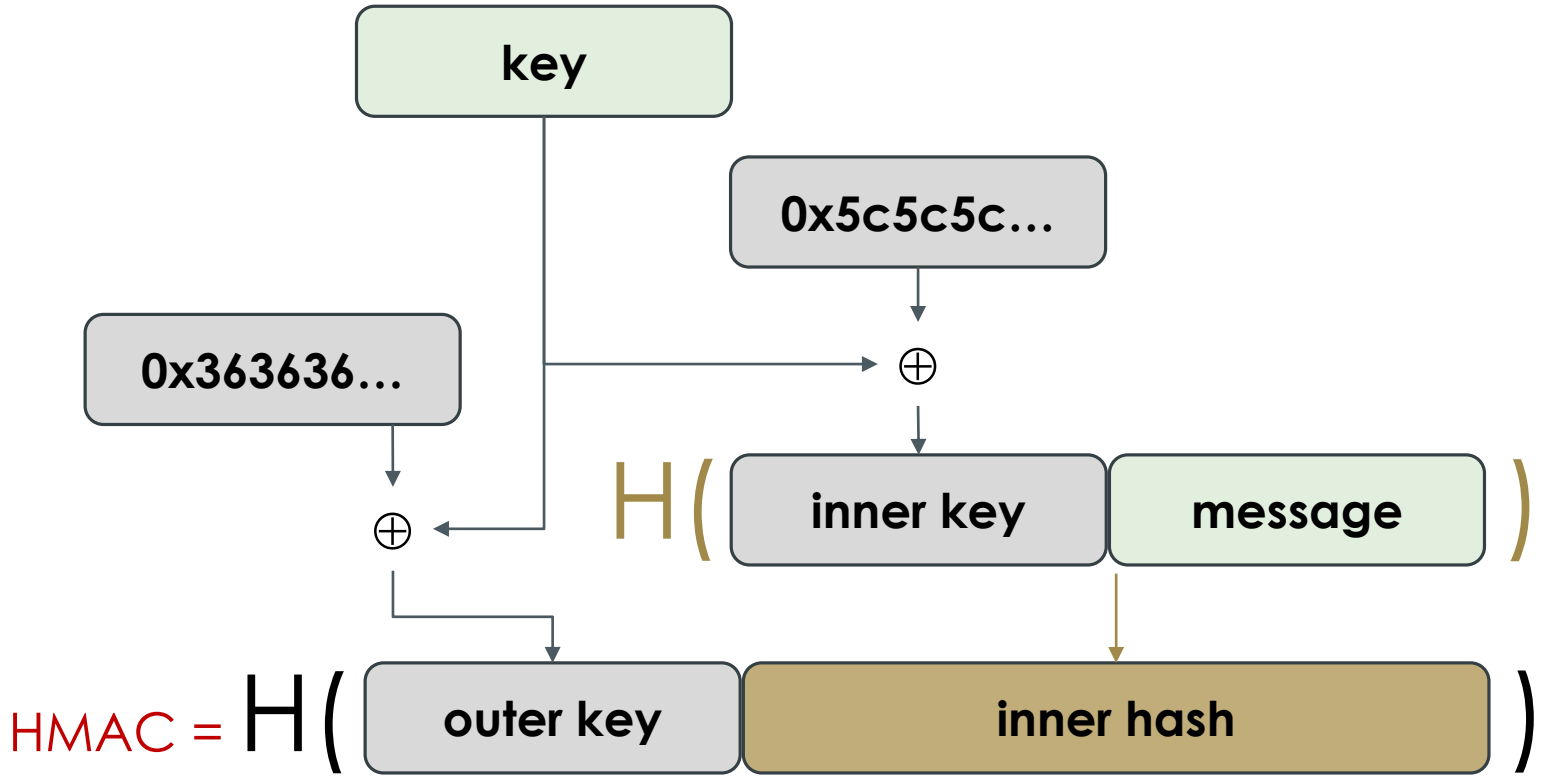
A MAC can also be constructed by concatenating the secret key with the message and using a hash, which creates a **keyed-Hash MAC** or **HMAC**



HMAC = H(

MAC using Hashes

A MAC can also be constructed by concatenating the secret key with the message and using a hash, which creates a **keyed-Hash MAC** or **HMAC**



MAC using Hashes

HMAC = H[(K ⊕ opad) + H((K ⊕ ipad) + M)]

- "+" denotes string concatenation, "⊕" denotes logical XOR
- **M** is the arbitrary-length message
- Assume hash block size = **n** bits (e.g., 512 bits for SHA1)
- **K** is the key, padded with 0's on right side to **n** bits
- **opad** = 0x3636... (or `00110110`) repeated to **n** bits
- **ipad** = 0x5c5c... (or `01011100`) repeated to **n** bits

Effectively:

$$HMAC = H(key_1 + H(key_2 + message))$$

- The inner and outer padding are chosen to minimize number of common bits in key_1 and key_2

HMAC

HMAC applies the hash twice for security

- HMAC = H((K ⊕ opad) + H((K ⊕ ipad) + M))

Simply concatenating the key with the message and hashing (*e.g.*, H(K + M)) is not secure

- Since many MDCs are iterated functions, a single (non-nested) hash may allow an adversary to add arbitrary information at the end of the message and compute a new, forged MAC

HMAC-SHA256 is currently one of the more-prevalent HMAC codes used in production

Hash-Based Data Structures

Block Chain

Hash Tree

Hash-Based Data Structures

It's often useful to verify the integrity of a **set** of things instead of just a single object.

- **Naïve approach:** Concatenate the objects into a long string and compute hash over the string

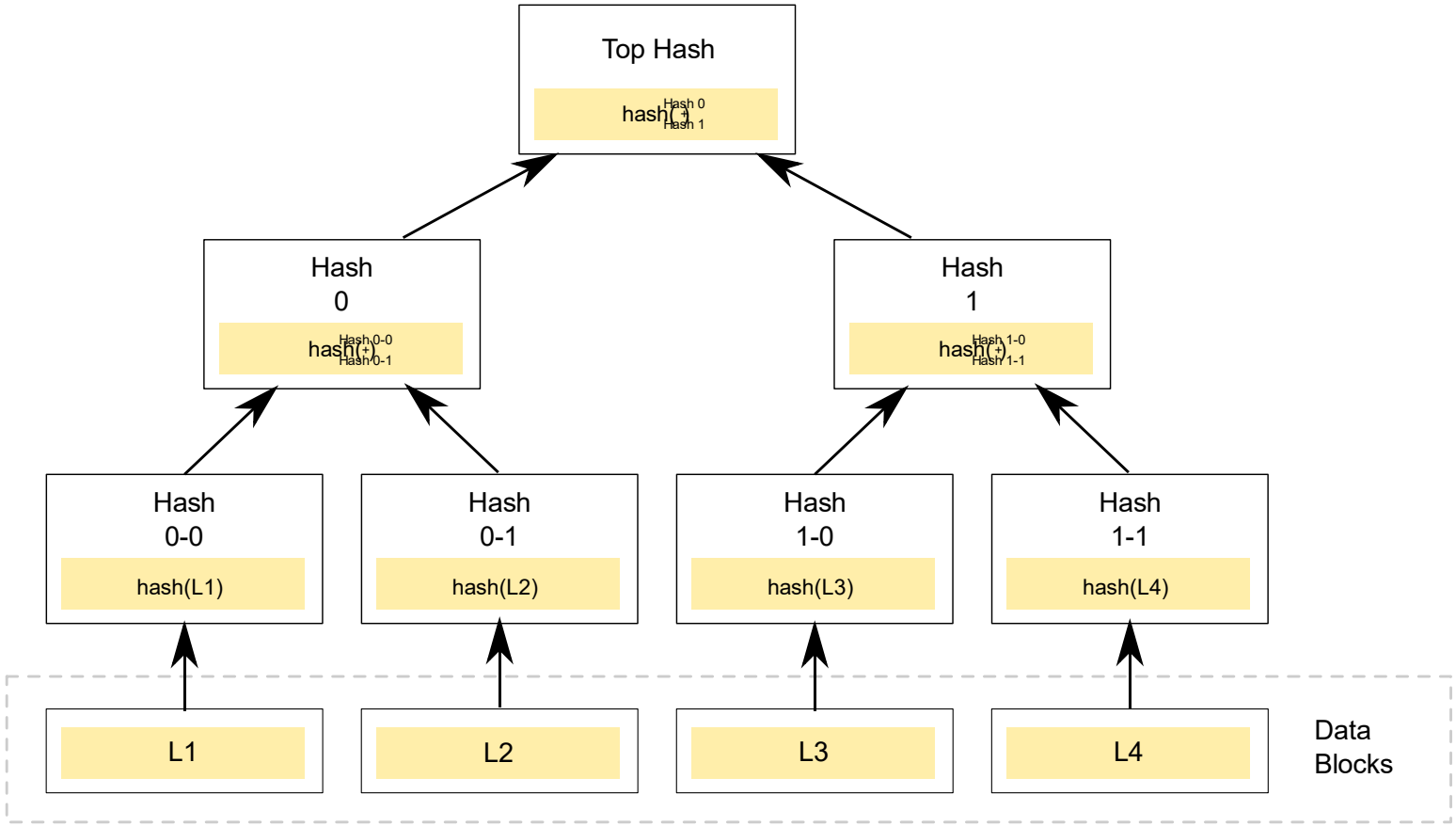
- **What's the problem with this?**

Hash-Based Data Structures

A number of useful data structures exist that can not only hold data, but also ensure that the data hasn't changed.

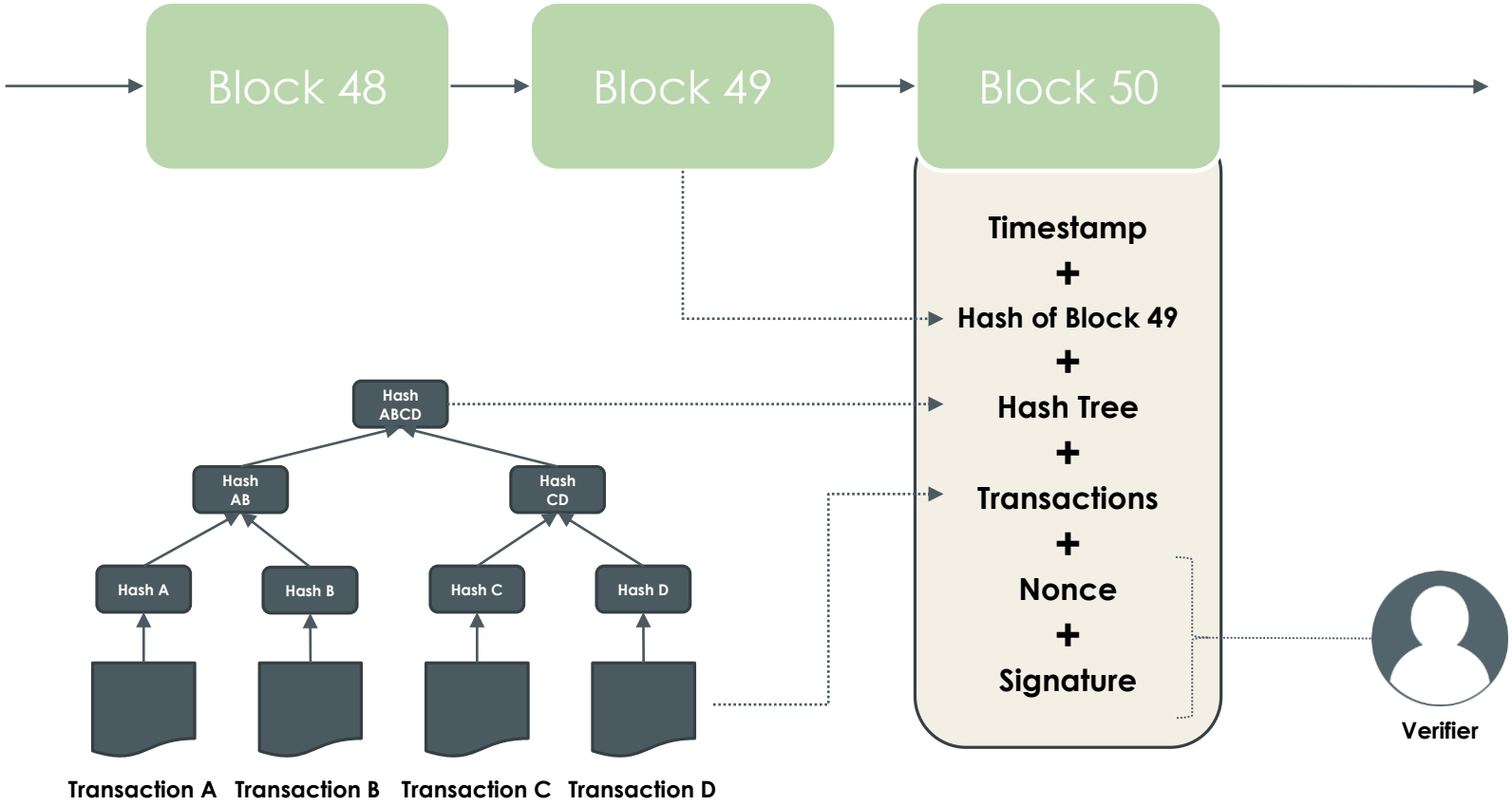
- **Hash Tree:** Provides data integrity, while allowing for easy updates.
- **Block Chain:** Allows for a journal of events to be created, with both integrity and authentication.

Merkle Tree



Source: Wikimedia Commons

Blockchain


Block 48
Block 49
Block 50
Timestamp
+
Hash of Block 49
+
Hash Tree
+
Transactions
+
Nonce
+
Signature
Hash ABCD
Hash AB
Hash CD
Hash A
Hash B
Hash C
Hash D
Transaction A
Transaction B
Transaction C
Transaction D
Verifier

Digital Signatures

X509 Certificates

Certificates

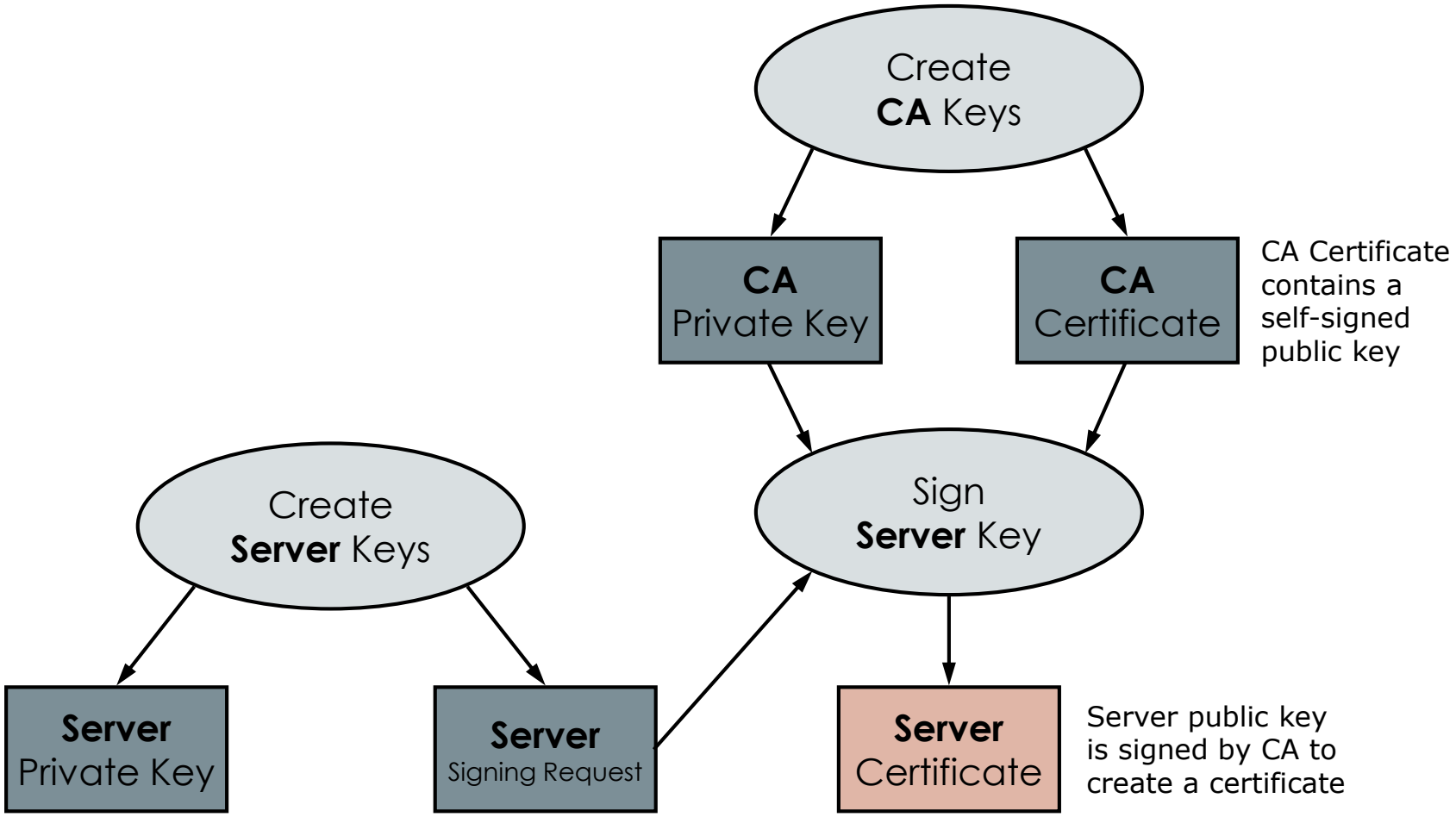
- Recall that a **certificate** is a message **signed** by a trusted entity (*e.g.*, certificate authority)
  - **Example:** a public key certificate contains a public key and an identity: "This public key xxx belong to www.amazon.com", certifying the authenticity of the public key
  - Typically, the CA signs the hash of the message rather than the entire message
- A signed message may be accompanied with a certificate
  - Recipient decrypts certificate using CA's public key to obtain sender's public key
  - Recipient decrypts the signature of a signed message using the sender's public key, generating a value that can be compared with the hash of the message

X.509 Certificate

- A standard format for certificates is X.509
  - A common method for storing X.509 certificates is in PEM files (".pem" extension) with Base64 encoding
    - Base64 produces ASCII output for easy transmission
    - Note that a PEM file can contain private keys or public-key certificates, or both
  - Other formats exist
- A web server using SSL (HTTPS) will send its X.509 certificate to the client, who can then use the certificate to verify the authenticity of the server's public key, and then use it to communicate with the server

Structure of an X.509 Certificate

- An X.509 certificate contains the following:
  - Issuer: information about the certificate authority
  - Subject: information about the bearer
    - The most important part is the **Common Name** (CN) which contains the name of the host being authenticated (*e.g.*, www.amazon.com).
  - Expiry and validity dates
  - Version numbers, etc.
  - Subject public key
  - Certificate signature: digital signature of the first part of the certificate, signed by the Issuer's private key


Create
CA Keys
CA
Private Key
CA
Certificate
CA Certificate
contains a
self-signed
public key
Create
Server Keys
Sign
Server Key
Server
Private Key
Server
Signing Request
Server
Certificate
Server public key
is signed by CA to
create a certificate

Example

Create ourselves a CA (Certificate Authority) certificate

- openssl req -new -x509 -extensions v3_ca -keyout cakey.pem -out cacert.pem -days 3650

Generate a new RSA private key for our web server

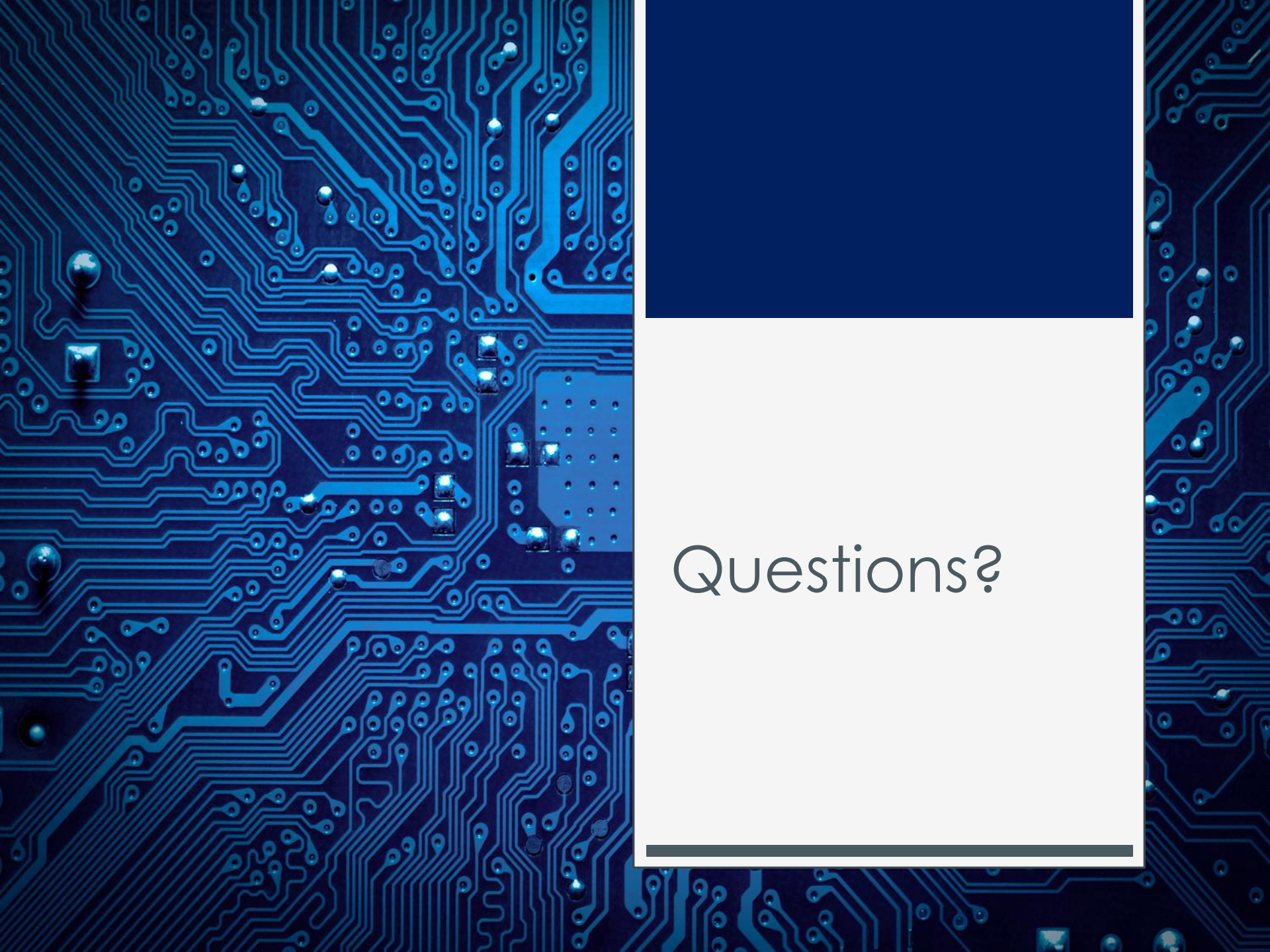
- openssl genrsa -des3 -out server.key 1024

Generate a CSR (Certificate Signing Request) for our server's key

- openssl req -new -key server.key -out server.csr

Sign the CSR with our CA key

- openssl x509 -req -days 365 -in server.csr -CA cacert.pem -CAkey cakey.pem -CAserial serial.txt –CAcreateserial -out server.crt


Questions?