Lecture 08 – Key Management/TLS

Ryan Cunningham
University of Illinois
ECE 422/CS 461 – Fall 2017

Security News

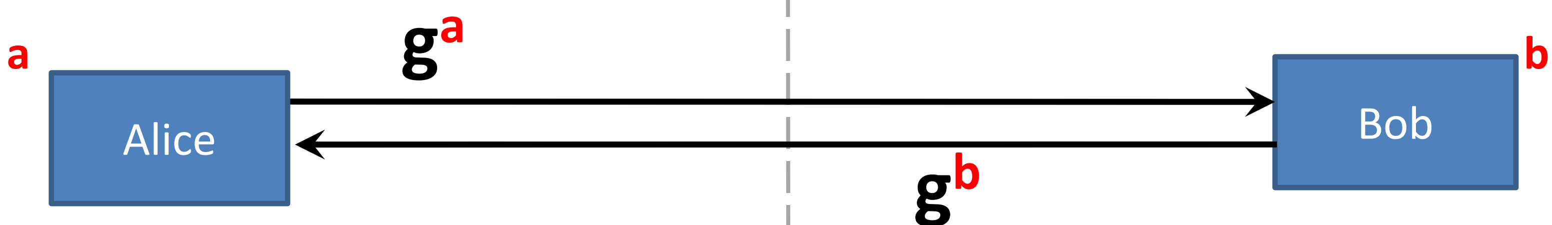
- Unpatched Apache Struts vulnerability caused Equifax breach (patch available since March)
- Microsoft patched .NET zero day
- Several D-Link routers pwned sideways

Diffie-Hellman protocol

1. Alice and Bob agree on public parameters (maybe in standards doc)

2.

Alice	**Bob**
Generates random secret exponent **a**.	Generates random secret value **b**.



3.

Computes **x**	Computes **x’**
$= (g^b)^a$	$= (g^a)^b$
$= g^{ba}$	$= g^{ab}$

(Notice that **x** == **x’**)

Can use **k** := hash(**x**) as a shared key.

Man-in-the-middle (MITM) attack

Alice does D-H exchange, *really with Mallory*, ends up with g^{au}

Bob does D-H exchange, *really with Mallory*, ends up with g^{bv}

Alice and Bob each think they are talking with the other, but really Mallory is between them and knows both secrets

Bottom line: D-H gives you secure connection, but you don't know who's on the other end!

Public Key Encryption

- ***Key generation:*** Bob generates a keypair public key, $\mathbf{k_{pub}}$ and private key, $\mathbf{k_{priv}}$

- ***Encrypt:*** Anyone can encrypt the message M, resulting in ciphertext C = **Enc**($\mathbf{k_{pub}}$, M)

- ***Decrypt:*** Only Bob has the private key needed to decrypt the ciphertext: M=**Dec**($\mathbf{k_{priv}}$, C)

- ***Security***: Infeasible to guess M or $\mathbf{k_{priv}}$, even knowing $\mathbf{k_{pub}}$ and seeing ciphertexts

Public Key Digital Signature

- Key generation: Bob generates a keypair public key, $\mathbf{k}_{\mathbf{pub}}$ and private key, $\mathbf{k}_{\mathbf{priv}}$

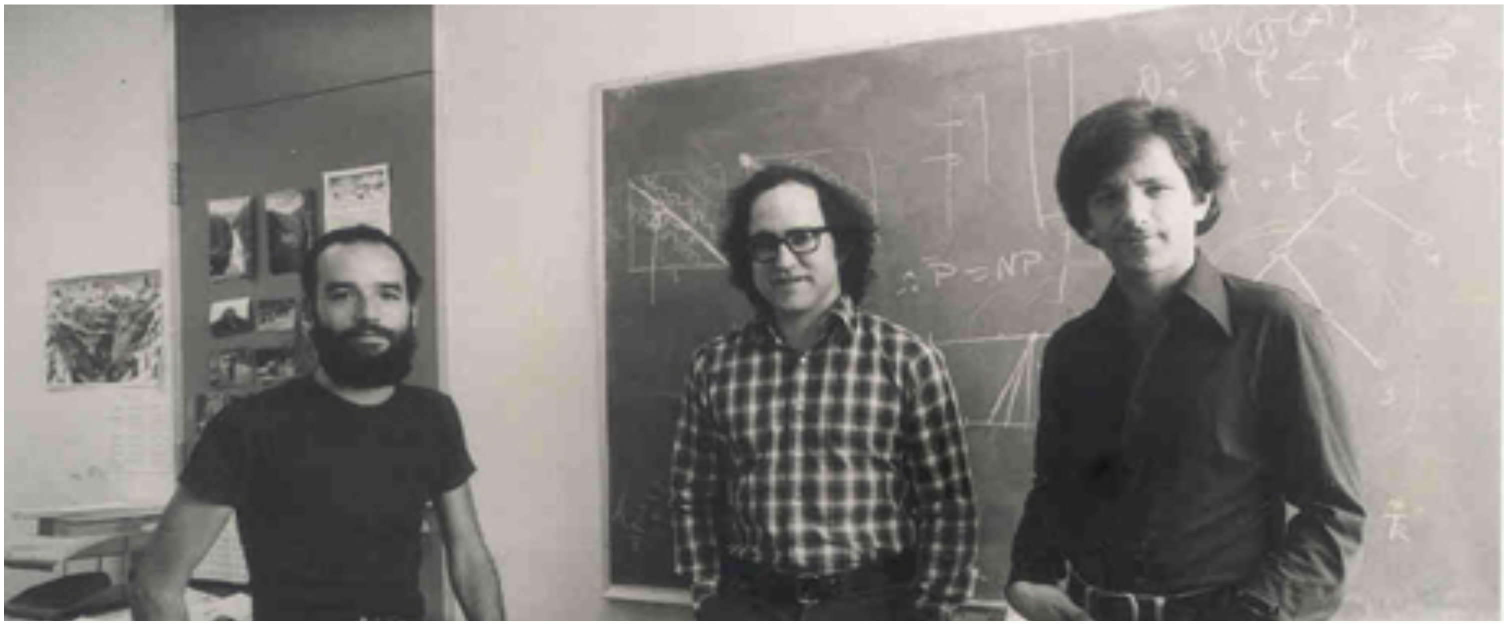
- Bob can sign a message M, resulting in signature S =Sign($\mathbf{k}_{\mathbf{priv}}$, M)

- Anyone who knows $\mathbf{k}_{\mathbf{pub}}$ can check the signature: Verify($\mathbf{k}_{\mathbf{pub}}$, M, S) =? 1

- "Unforgeable": Computationally infeasible to guess S or $\mathbf{k}_{\mathbf{priv}}$, even knowing $\mathbf{k}_{\mathbf{pub}}$ and seeing signatures on other messages

A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

R.L. Rivest, A. Shamir, and L. Adleman*



Best known, most common public-key algorithm: **RSA**

Rivest, Shamir, and Adleman 1978

(earlier by Clifford Cocks of British intelligence, in secret)

How RSA works

Key generation:

1. Pick large (say, 1024 bits) random primes **p** and **q**
2. Compute **N** := **pq** (RSA uses multiplication mod **N**)
3. Pick **e** to be relatively prime to $\Phi(N)=(\mathbf{p}-1)(\mathbf{q}-1)$
4. Find **d** so that **ed** mod (**p**-1)(**q**-1) = 1
5. Finally:

   Public key is (**e**,**N**)

   Private key is (**d**,**N**)

To sign: S = ***Sign***(**x**) $= \mathbf{x}^{\mathbf{d}} \bmod \mathbf{N}$

To verify: ***Verif***(S) $= S^{\mathbf{e}} \bmod \mathbf{N}$

Check ***Verif***(S) =? M

Why RSA works

"Completeness" theorem:

For all $0 < \mathbf{x} < \mathbf{N}$, we can show that ***Verif***(***Sign***($\mathbf{x}$)) = $\mathbf{x}$

Proof:

Verif(***Sign***($\mathbf{x}$)) $= (\mathbf{x}^{\mathbf{d}} \bmod \mathbf{pq})^{\mathbf{e}} \bmod \mathbf{pq}$

$= \mathbf{x}^{\mathbf{ed}} \bmod \mathbf{pq}$

$= \mathbf{x}^{\mathbf{a}(\mathbf{p}-1)(\mathbf{q}-1)+1} \bmod \mathbf{pq}$ for some $\mathbf{a}$

(because $\mathbf{ed} \bmod (\mathbf{p}-1)(\mathbf{q}-1) = 1$)

$= (\mathbf{x}^{(\mathbf{p}-1)(\mathbf{q}-1)})^{\mathbf{a}}\mathbf{x} \bmod \mathbf{pq}$

$= (\mathbf{x}^{(\mathbf{p}-1)(\mathbf{q}-1)} \bmod \mathbf{pq})^{\mathbf{a}}\mathbf{x} \bmod \mathbf{pq}$

$= 1^{\mathbf{a}}\mathbf{x} \bmod \mathbf{pq}$

(because of the fact that if $\mathbf{p,q}$ are prime, then for all $0 < \mathbf{x} < \mathbf{N}$, $\mathbf{x}^{(\mathbf{p}-1)(\mathbf{q}-1)} \bmod \mathbf{pq} = 1$) Fermat's little theorem

$= \mathbf{x}$

Subtle fact: RSA can be used for either confidentiality or integrity

RSA for confidentiality:

Encrypt with public key, Decrypt with private key

Public key is **(e,N)**

Private key is **(d,N)**

To encrypt: $E(\mathbf{x}) = \mathbf{x}^{e} \bmod \mathbf{N}$

To decrypt: $D(\mathbf{x}) = \mathbf{x}^{d} \bmod \mathbf{N}$

RSA for integrity:

Encrypt (“sign”) with private key

Decrypt (“verify”) with public key

RSA drawback: Performance

Factor of 1000 or more slower than AES.

Dominated by exponentiation – cost goes up (roughly) as cube of key size.

Message must be shorter than **N**.

Use in practice:

Hybrid Encryption (similar to key exchange):

Use RSA to encrypt a random key **k < N**, then use AES

Signing*:*

Compute **v** := hash(**m**), use RSA to sign the hash

Should always use crypto libraries to get details right

Key Management

The hard part of crypto: **Key-management**

Principles:

0. Always remember, key management is the hard part!

1. Each key should have only one purpose
   (in general, no guarantees when keys reused elsewhere)

2. Vulnerability of a key increases:
   a. The more you use it.
   b. The more places you store it.
   c. The longer you have it.

3. Keep your keys far from the attacker.

4. Protect yourself against compromise of old keys.

   Goal: **forward secrecy** — learning old key shouldn't help adversary learn new key.

   [How can we get this?]

Building a secure channel

What if you want confidentiality and integrity at the same time?

Encrypt, then MAC
not the other way around

Use separate keys for confidentiality and integrity.

- Need two shared keys,
  - but only have one?
  - That's what PRGs are for!
- If there's a reverse (Bob to Alice) channel, use separate keys for that too

Issue: How big should keys be?

Want prob. of guessing to be infinitesimal… but watch out for Moore's law – safe size gets 1 bit larger every 18 months

128 bits usually safe for ciphers/PRGs

Need larger values for MACs/PRFs due to **birthday attack**

Often trouble if adversary can find <u>any two messages</u> with same MAC

Attack: Generate random values, look for coincidence.

Requires $O(2^{|k|/2})$ time, $O(2^{|k|/2})$ space.

For 128-bit output, takes 2^{64} steps: doable!

Upshot: Want output of MACs/PRFs to be twice as big as cipher keys e.g. use HMAC-SHA256 alongside AES-128

Key Type *Move the cursor over a type for description*	Cryptoperiod Originator Usage Period (OUP)	Recipient Usage Period
Private Signature Key	1-3 years	-
Public Signature Key	Several years (depends on key size)	
Symmetric Authentication Key	<= 2 years	<= OUP + 3 years
Private Authentication Key	1-2 years	
Public Authentication Key	1-2 years	
Symmetric Data Encryption Key	<= 2 years	<= OUP + 3 years
Symmetric Key Wrapping Key	<= 2 years	<= OUP + 3 years
Symmetric RBG keys	Determined by design	-
Symmetric Master Key	About 1 year	-
Private Key Transport Key	<= 2 years (1)	
Public Key Transport Key	1-2 years	
Symmetric Key Agreement Key	1-2 years (2)	
Private Static Key Agreement Key	1-2 years (3)	
Public Static Key Agreement Key	1-2 years	
Private Ephemeral Key Agreement Key	One key agreement transaction	
Public Ephemeral Key Agreement Key	One key agreement transaction	
Symmetric Authorization Key	<= 2 years	
Private Authorization Key	<= 2 years	
Public Authorization Key	<= 2 years	

https://www.keylength.com/en/4/

Date	Minimum of Strength	Symmetric Algorithms	Factoring Modulus	Discrete Logarithm Key	Discrete Logarithm Group	Elliptic Curve	Hash (A)	Hash (B)
(Legacy)	80	2TDEA*	1024	160	1024	160	SHA-1**	
2016 - 2030	112	3TDEA	2048	224	2048	224	SHA-224 SHA-512/224 SHA3-224	
2016 - 2030 & beyond	128	AES-128	3072	256	3072	256	SHA-256 SHA-512/256 SHA3-256	SHA-1
2016 - 2030 & beyond	192	AES-192	7680	384	7680	384	SHA-384 SHA3-384	SHA-224 SHA-512/224
2016 - 2030 & beyond	256	AES-256	15360	512	15360	512	SHA-512 SHA3-512	SHA-256 SHA-512/256 SHA-384 SHA-512 SHA3-512

https://www.keylength.com/en/4/

Attacks against Crypto

1. Brute force: trying all possible private keys
2. Mathematical attacks: factoring
3. Timing attacks: using the running time of decryption
4. Hardware-based fault attack: induce faults in hardware
5. Chosen ciphertext attack
6. Architectural changes

Security on the web

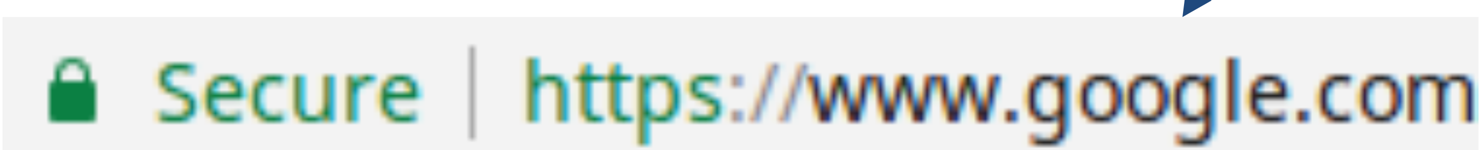
- We don’t want a network adversary to **modify our pages** or **eavesdrop**

Integrity

Confidentiality

- Defense: HTTPS

Authenticity



Threat model

- Controls infrastructure (routers, DNS, wireless access points)
- Passive attacker: only eavesdrops
- Active attacker: eavesdrops, injects, blocks, and modifies packets
- What examples?
  - Internet Cafe, hotel, ECE/Siebel, fake web site
- Does not protect against:
  - Intruder on server
  - Spyware on client
  - SQL injection, XSS, CSRF

Certificates

- Make use of trusted **"Certificate Authorities" (CA)**

- "This public key with SHA-256 hash (XXX) belongs to the site (name, e.g., Amazon.com)"
  - **Digitally signed** by a certificate authority

- Your browsers (e.g., Firefox, Chrome) trust a specific set of CAs as root CAs
  - Shipped with the public keys of the root CAs
  - Why do we need more than 1?

Certificates

How does Alice (web browser) obtain Bob's public key?

[Think of like a notary]

Browser (Alice)
(Knows $\mathbf{CA_{Pub}}$)

Server (Bob)

Certificate Authority (CA)
(Keeps $\mathbf{CA_{Priv}}$ Secret)

1. Choose ($\mathbf{Bob_{Priv}}$,$\mathbf{Bob_{Pub}}$)
$\mathbf{Bob_{Pub}}$ and evidence he is "Bob"

2. Checks evidence
Signs certificate with $\mathbf{CA_{Priv}}$
"Bob's key is $\mathbf{Bob_{Pub}}$
Signed, CA"

3. Keeps certificate on file

---

1. Initiates a connection

2. Sends cert to Alice
"Bob's key is $\mathbf{Bob_{Pub}}$ Signed, CA"

3. Verifies CA's signature with $\mathbf{CA_{Pub}}$ and creates a secure channel using $\mathbf{Bob_{Pub}}$

How the CA verifies your identity

- Typically ‘DV’ (**domain** validated)
  - Proves you are in control of DNS registration
  - Just an email based challenge to the address in the domain registration records
    - Or some default email address, admin@domain.com
    - Minimally secure [Why?]
  - Alternately a web-based challenge
    - Include challenge response in a <meta> tag
- Can also get ‘EV’ certs (extended validation)
- Cert has an expiration date

  (e.g., one year ahead) [Why?]

How to invalidate certificates?

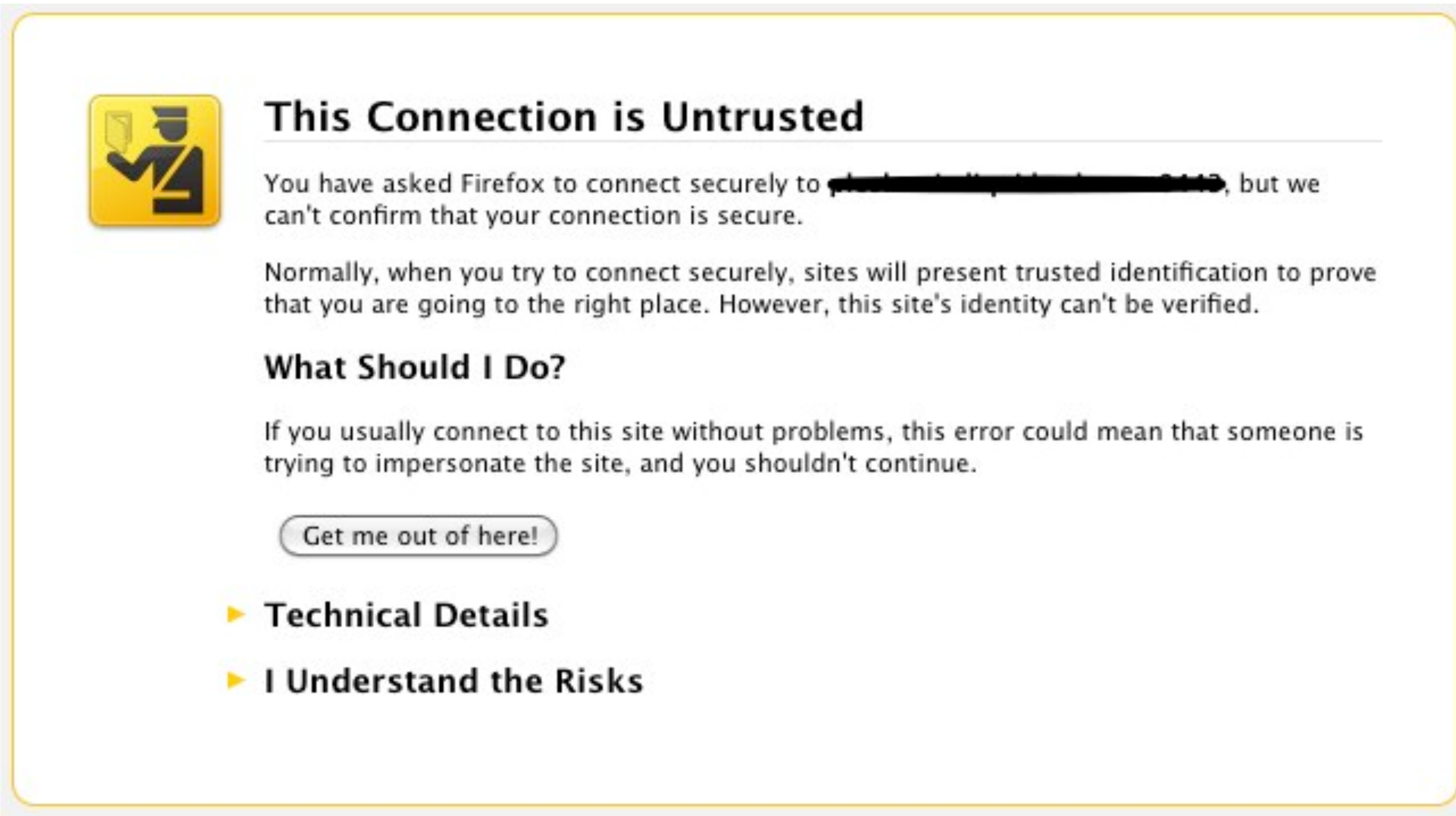
- Expiration date of certs
- Certificate revocation
- What happens if a CA’s secret key is leaked?
  - Can we trust the old certs from that CA?
- Interesting fact:
  - Google has instrumented Chrome such that when it observes a certificate for Google.com that it doesn’t recognize, it panics…. (has happened several times)

Self-signed Certificates

- Issuer signs their own certificate
  - A loop in the owner and signer
- Avoid CA fees, useful for testing
  - You can add yourself as a CA to your own browser
- Browsers display warnings that users have to override
- Protects only against passive attacker "optimistic encryption"

Example: https://www.pcwebshop.co.uk/


Not secure
https://pcwebshop.co.uk


This Connection is Untrusted
You have asked Firefox to connect securely to [redacted], but we can't confirm that your connection is secure.
Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.
What Should I Do?
If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.
Get me out of here!
Technical Details
I Understand the Risks

How do we translate?

Cryptographic Primitives

Symmetric Encryption

RSA

HMAC

Certificate

Public Key

RC4

Diffie-Hellman

DSA

ECDSA

Asymmetric Encryption

How do we translate?

Cryptographic Primitives

Symmetric Encryption

RSA

HMAC

Certificate

Public Key

RC4

Diffie-Hellman

DSA

ECDSA

Asymmetric Encryption

Objectives

Message Integrity

Confidentiality

Authentication

How do we translate?

Cryptographic Primitives

Symmetric Encryption

RSA

HMAC

Certificate

Public Key

AES

Diffie-Hellman

DSA

ECDSA

Asymmetric Encryption

[**Typical HTTPS Connection**]

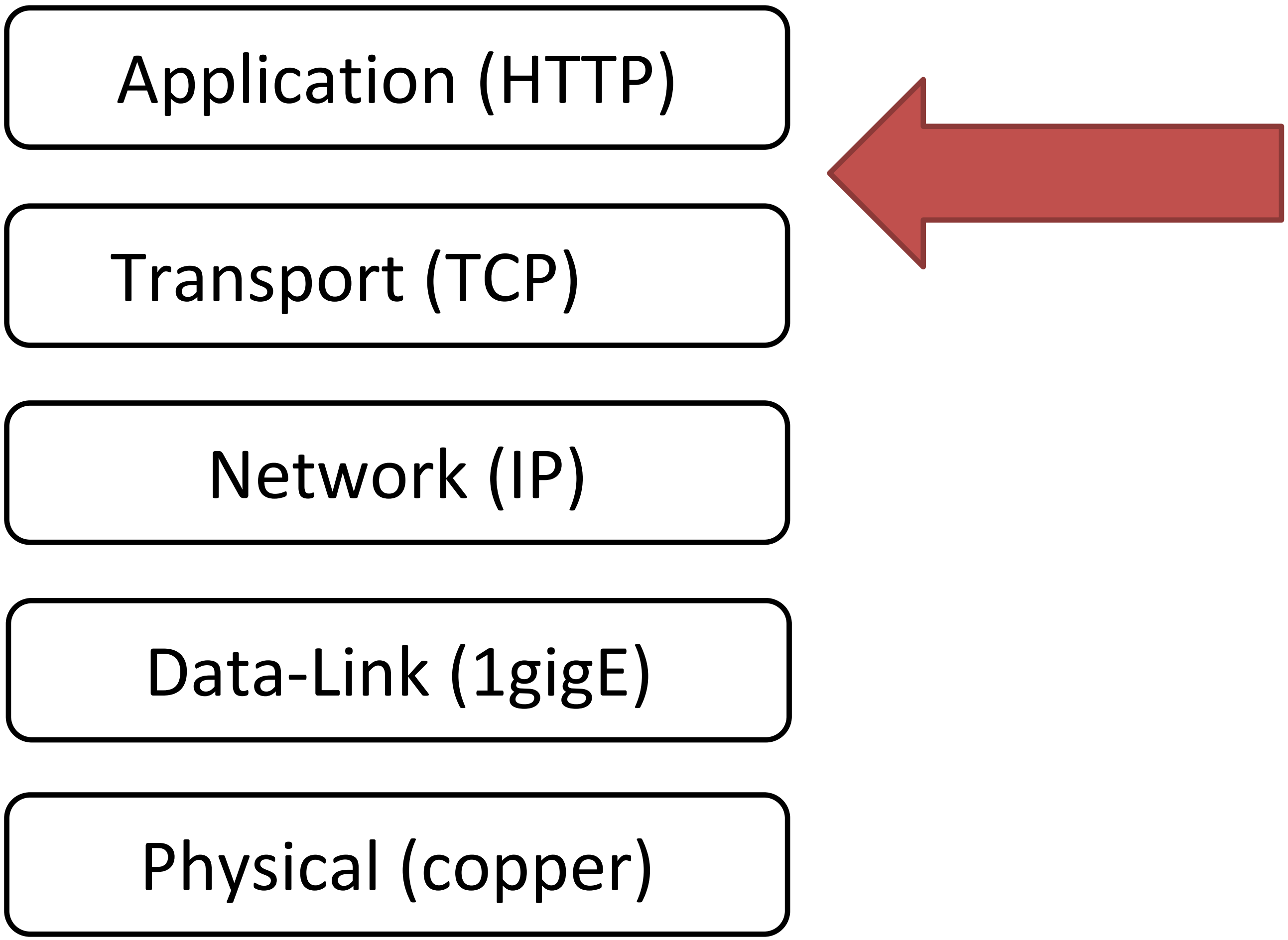
Case Study: SSL/TLS

- Arguably the most important (and widely used) cryptographic protocol on the Internet

- Almost all encrypted protocols (minus SSH) use SSL/TLS for transport encryption

- HTTPS, POP3, IMAP, SMTP, FTP, NNTP, XMPP (Jabber), OpenVPN, SIP (VoIP), …

Where does TLS live?

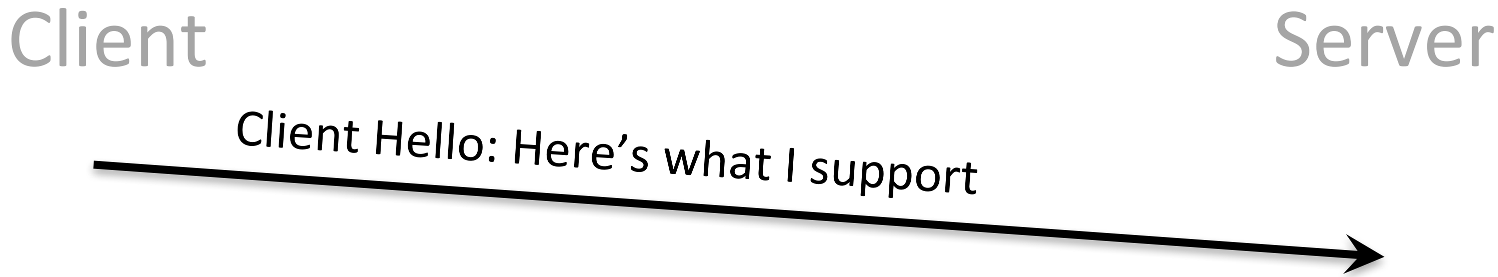


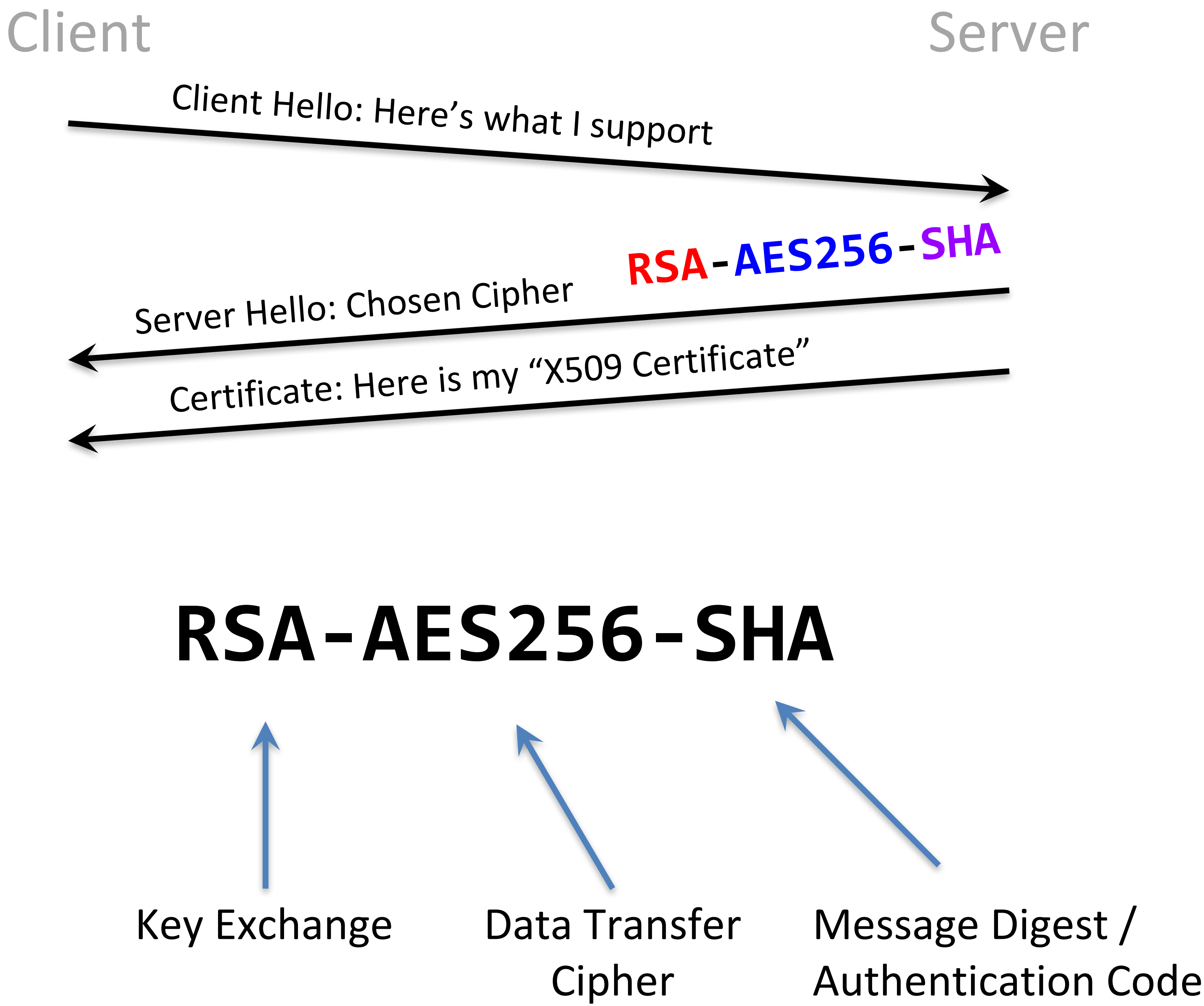
Goals

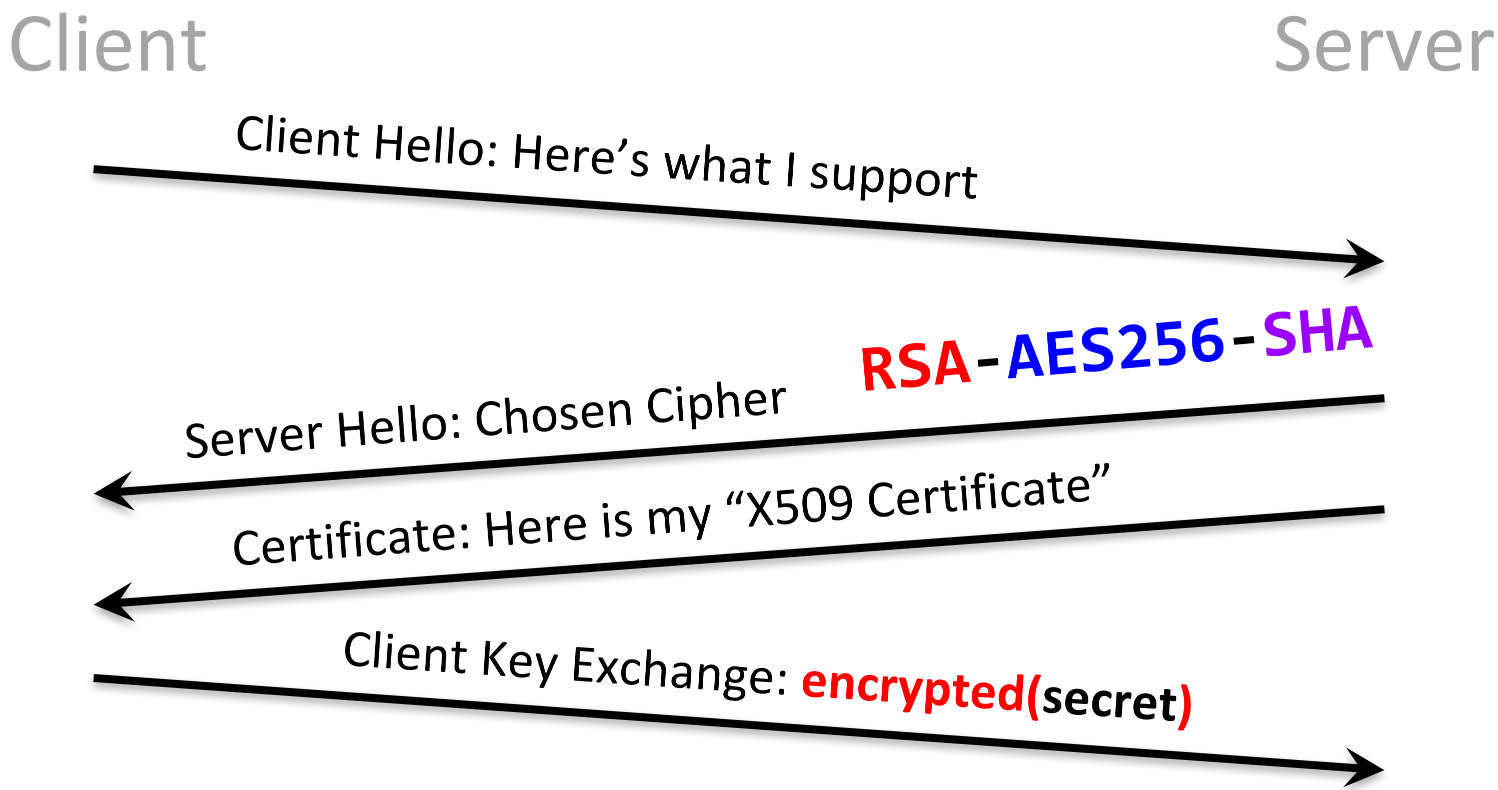
- ✓ Confidentiality (Symmetric Crypto)
- ✓ Message Integrity (HMACs)
- ✓ Authentication (Public Key Crypto)

Client Server

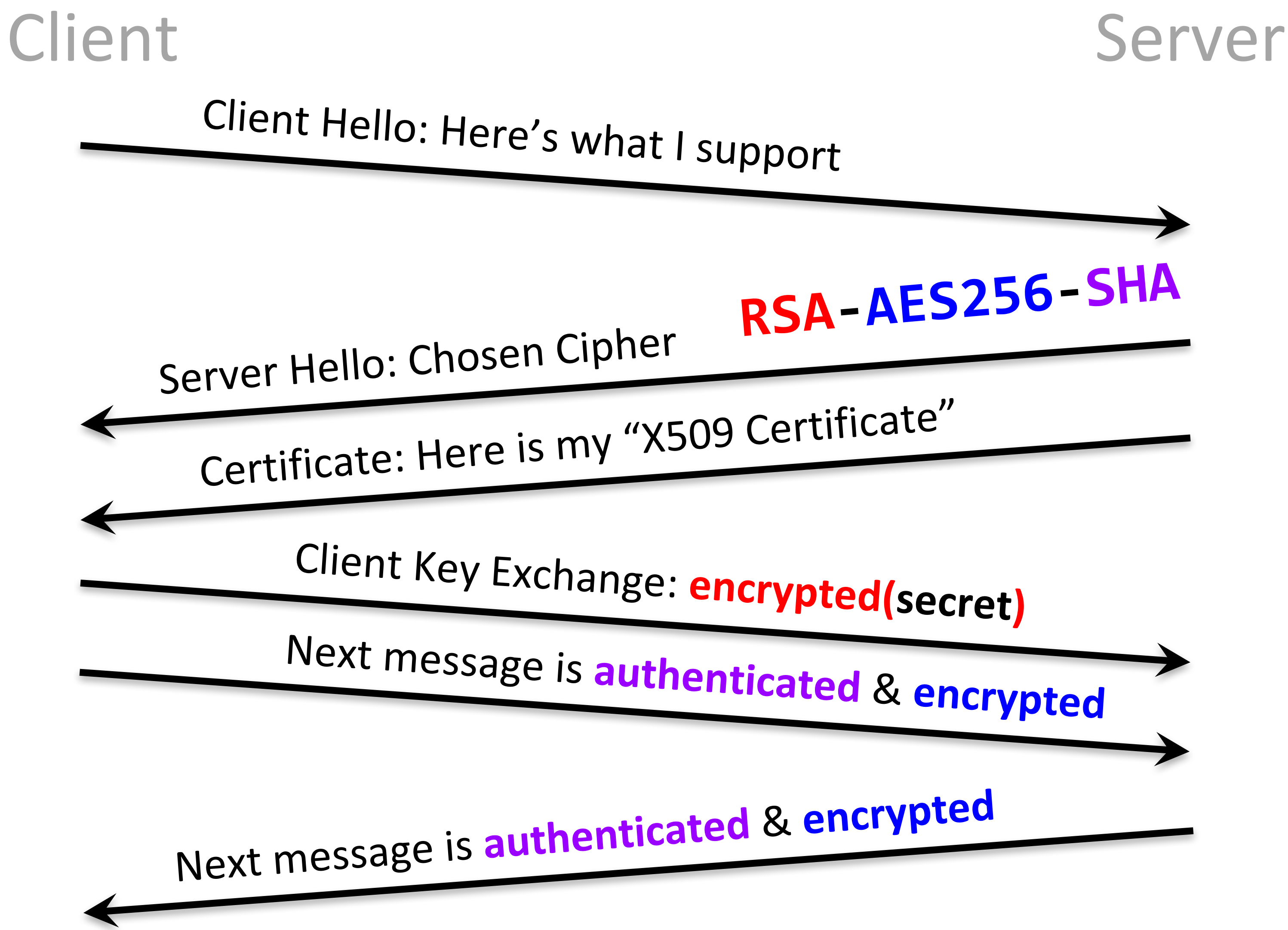
“the handshake”


Client
Server
Client Hello: Here's what I support


Client
Server
Client Hello: Here's what I support
RSA-AES256-SHA
Server Hello: Chosen Cipher
Certificate: Here is my "X509 Certificate"
RSA-AES256-SHA
Key Exchange
Data Transfer Cipher
Message Digest / Authentication Code



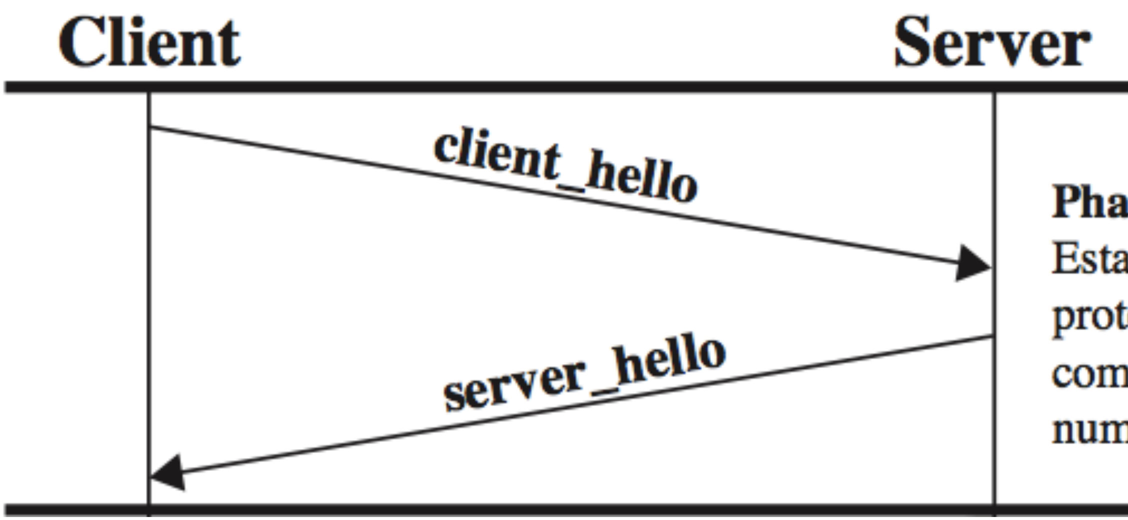
Encrypted using Server's public key
(The same public key included in the Cert)
This means: only the server can decrypt the secret! (Avoids MitM)



Shared **secret** is encrypted using Server's RSA public key
(The same RSA public key included in the Cert)
This means: only the server can decrypt the secret! (Avoids MitM)

TLS Handshake

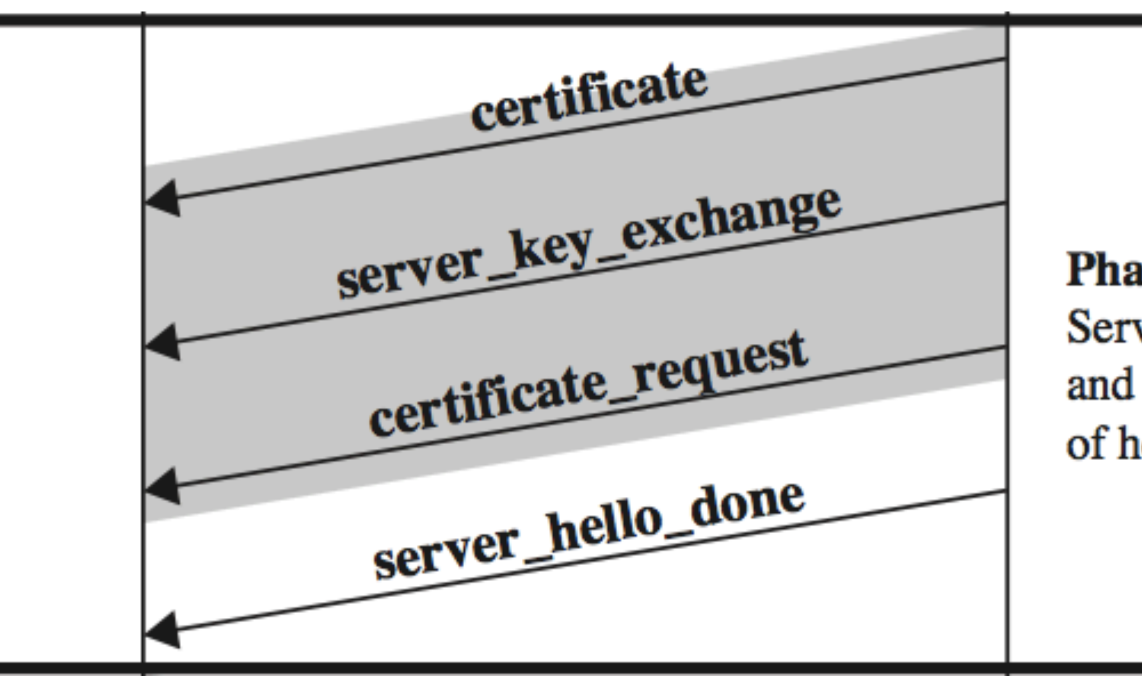
- Phase 1: establish capabilities
  - Which version of TLS?
  - What our session ID?
  - What is our cipher suite?
  - Are we compressing data?



Phase 1
Establish security capabilities, including protocol version, session ID, cipher suite, compression method, and initial random numbers.

TLS Handshake

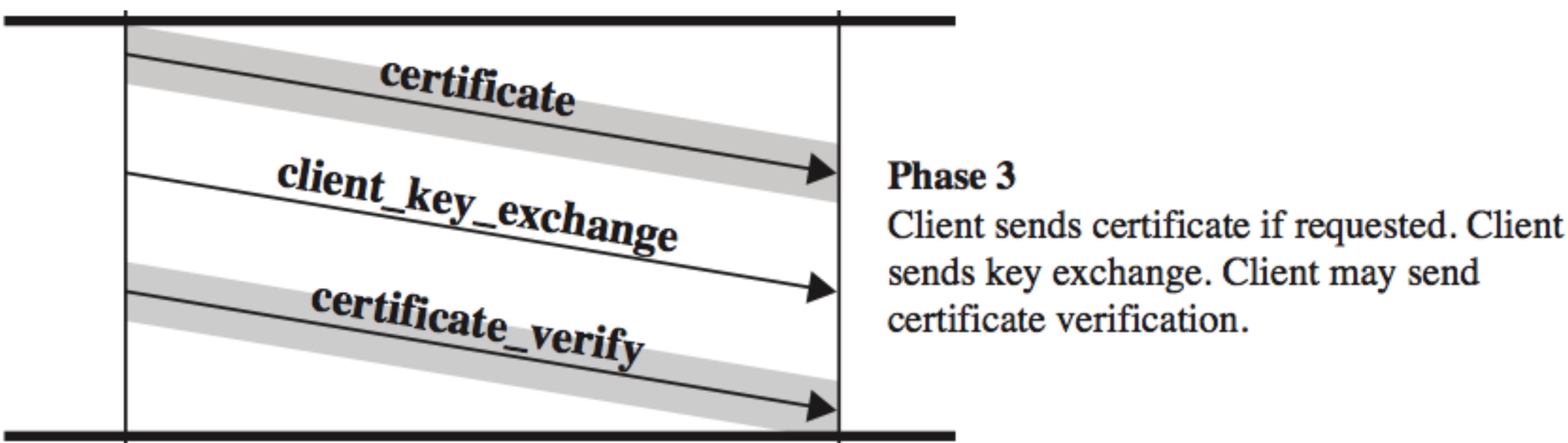
- Phase 2: Server Authentication
  - Server sends certificate



Phase 2
Server may send certificate, key exchange, and request certificate. Server signals end of hello message phase.

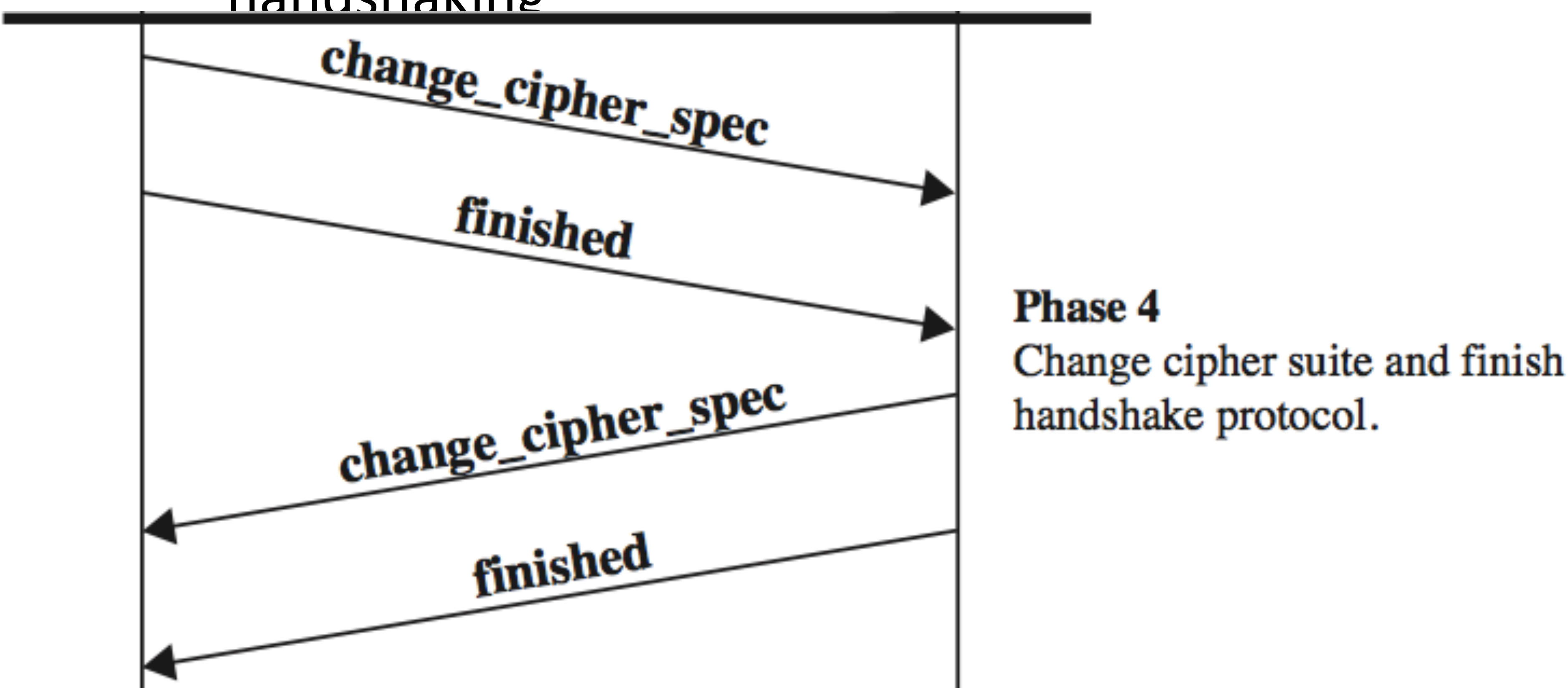
TLS Handshake

- Phase 3: Client Authentication
  - –Client sends certificate (maybe)
  - –Client exchanges key
  - –Client sends verification of server cert



TLS Handshake

- Phase 4: Switch to Secure Connection
  - Change to agreed upon cipher suite and stop handshaking



Cipher Suites

```
The following CipherSuite definitions require that the server provide
an RSA certificate that can be used for key exchange.  The server may
request any signature-capable certificate in the certificate request
message.

   CipherSuite TLS_RSA_WITH_NULL_MD5                 = { 0x00,0x01 };
   CipherSuite TLS_RSA_WITH_NULL_SHA                 = { 0x00,0x02 };
   CipherSuite TLS_RSA_WITH_NULL_SHA256              = { 0x00,0x3B };
   CipherSuite TLS_RSA_WITH_RC4_128_MD5              = { 0x00,0x04 };
   CipherSuite TLS_RSA_WITH_RC4_128_SHA              = { 0x00,0x05 };
   CipherSuite TLS_RSA_WITH_3DES_EDE_CBC_SHA         = { 0x00,0x0A };
   CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA          = { 0x00,0x2F };
   CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA          = { 0x00,0x35 };
   CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA256       = { 0x00,0x3C };
   CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA256       = { 0x00,0x3D };
```

HTTPS key exchange

At the end of the exchange, a secret is used to generate 4 keys (2 for MAC, 2 for encryption)

1. RSA key exchange
   - Use RSA for encryption to achieve confidentiality
   - Use RSA for signature to achieve authentication
2. Ephemeral Diffie Hellman (EDH)
   - For *forward secrecy* guarantees
3. Fixed Diffie Hellman
   - For packet inspection within the server's network