Formal Methods for Analyzing Crypto Protocols: from attacks to proofs

Karthikeyan Bhargavan

INVENTORS FOR THE DIGITAL WORLD

+ many, many others.
(INRIA, Microsoft Research,
LORIA, IMDEA, Univ of Pennsylvania, Univ of Michigan, JHU)

Analyzing Real-World Protocols

Internet protocols (TLS, SSH, IPsec) seemingly implement textbook cryptographic protocols

… yet, not exactly the same protocols

- Modeling gaps between paper proofs and real protocol
- Implementation gaps between protocol and deployment

These gaps lead to many attacks, new questions

- Can we prove the deployed protocol correct?
- Can we show that a theoretical attack can be exploited?
- Important to understand where these gaps come from, so we can close them in new protocol designs

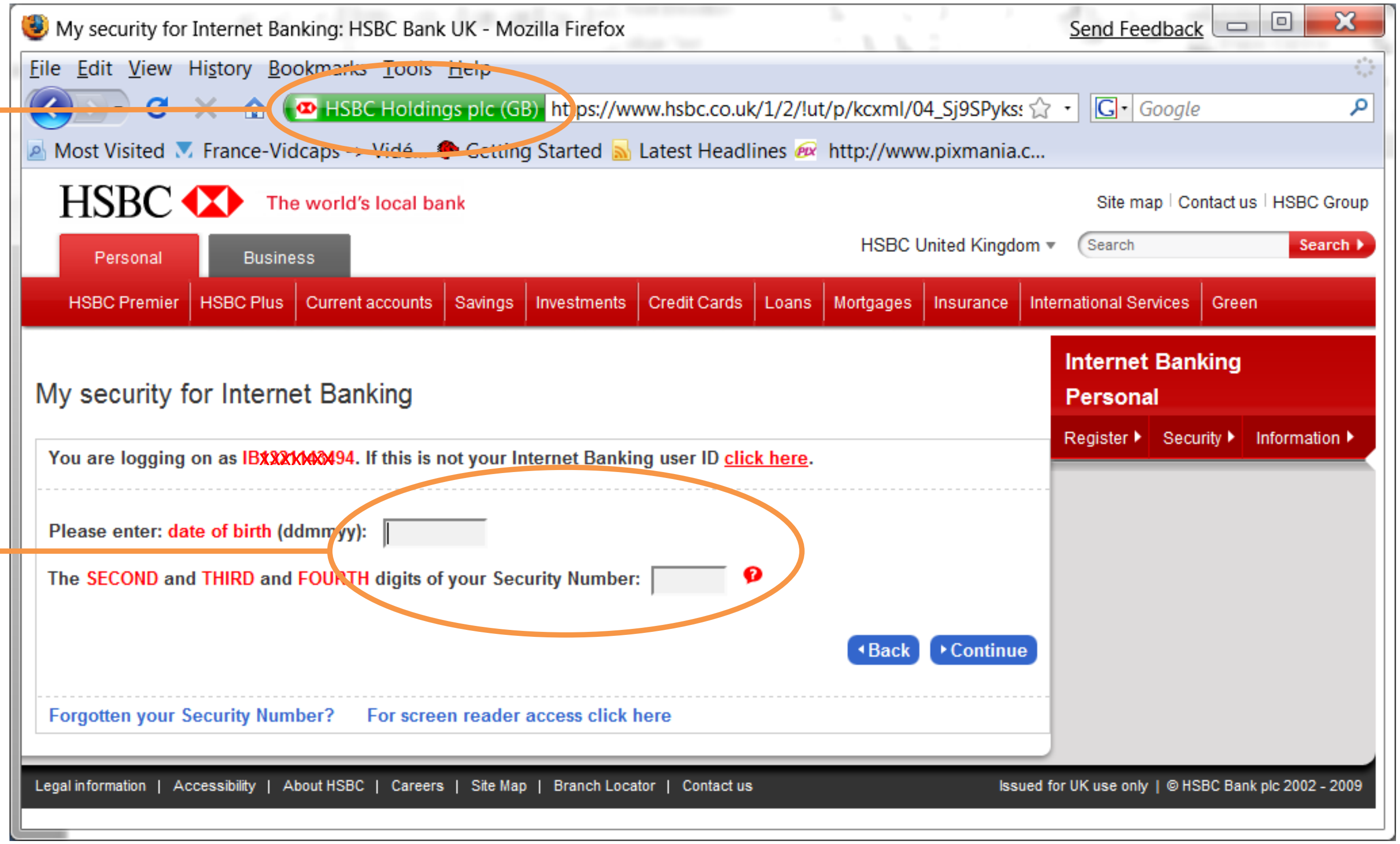
Example: HTTPS for Web Security

Secure connection to bank's website

Nobody other than the bank can read what I type (confidentiality)

My secret login Information

Nobody other than me can access my account page (authentication)



Goal: *Prevent unauthorized access to data even if an unknown attacker controls the network and some other bank clients.*

Secure Channel?
compose a standard AKE with a standard AEAD

Many recent attacks on HTTPS

• BEAST	CBC predictable IVs	[Sep'11]
• CRIME	Compression before Encryption	[Sep'12]
• RC4	Keystream biases	[Mar'13]
• Lucky 13	MAC-Encode-Encrypt CBC	[May'13]
• 3Shake	Insecure resumption	[Apr'14]
• POODLE	SSLv3 MAC-Encode-Encrypt	[Dec'14]
• SMACK	State machine attacks	[Jan'15]
• FREAK	Export-grade 512-bit RSA	[Mar'15]
• LOGJAM	Export-grade 512-bit DH	[May'15]
• SLOTH	RSA-MD5 signatures	[Jan'16]
• DROWN	SSLv2 RSA-PKCS#1v1.5	[Mar'16]

Many recent attacks on HTTPS

High-profile attacks, with Logos!

What’s going on?

How do we prevent this in the future?

Lecture Plan

Part I: *Attacks on Authenticated Key Exchange in TLS*

Part 2: *Finding Protocol Flaws with Symbolic Analysis*

Part 3: *Mechanizing Cryptographic Protocol Proofs*

Part 4: *Towards High-Assurance Crypto Software*

Part I:

Attacks on Authenticated Key Exchange in TLS

Reading Materials

- ***TLS 1.2.*** IETF RFC 5246.
- ***Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS***. IEEE Security and Privacy 2014.
- ***Messy State of the Union: Taming the Composite State Machines of TLS***. IEEE Security and Privacy 2015.
- **Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice.** ACM CCS 2015.
- ***Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH.*** ISOC NDSS 2016.

Transport Layer Security (1994—)

The default secure channel protocol?

HTTPS, 802.1x, VPNs, files, mail, VoIP, …

20 years of attacks and fixes

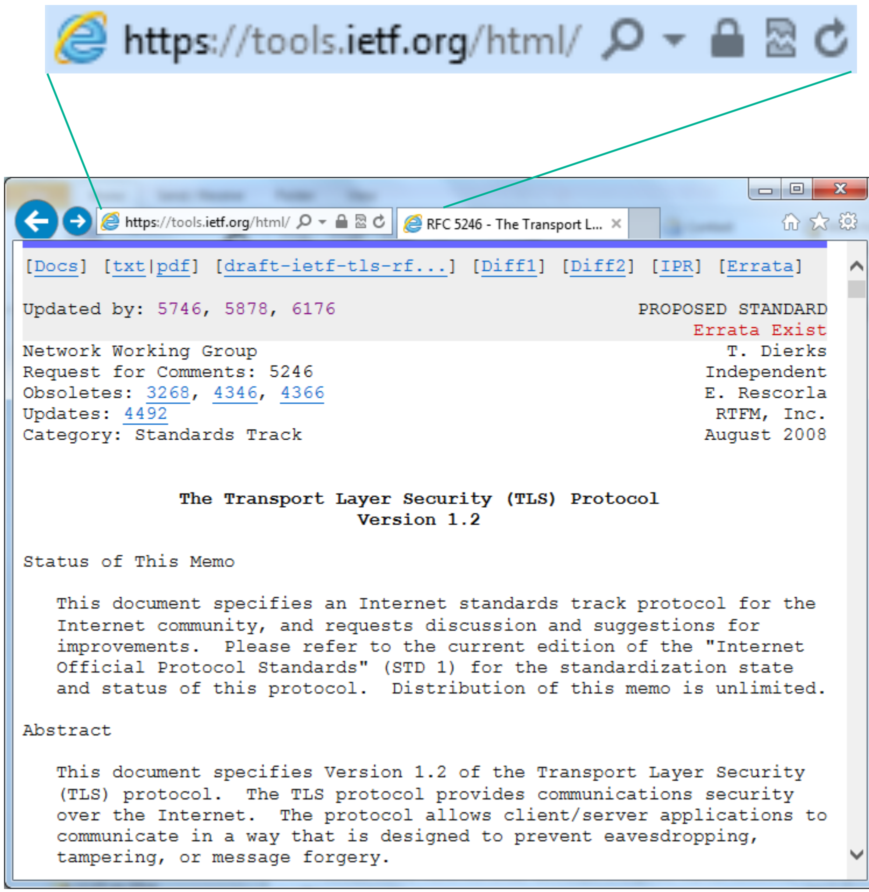
1994 Netscape's Secure Sockets Layer
1996 SSLv3
1999 TLS1.0 (RFC2246)
2006 TLS1.1 (RFC4346)
2008 TLS1.2 (RFC5246)
2018? TLS1.3

Many implementations

OpenSSL, SecureTransport, NSS, SChannel, GnuTLS, JSSE, PolarSSL, …
many bugs, attacks, patches every year

Many security theorems

mostly for simplified models of TLS



[Docs] [txt|pdf] [draft-ietf-tls-rf...] [Diff1] [Diff2] [IPR] [Errata]

Updated by: 5746, 5878, 6176 PROPOSED STANDARD
Errata Exist

Network Working Group T. Dierks
Request for Comments: 5246 Independent
Obsoletes: 3268, 4346, 4366 E. Rescorla
Updates: 4492 RTFM, Inc.
Category: Standards Track August 2008

**The Transport Layer Security (TLS) Protocol
Version 1.2**

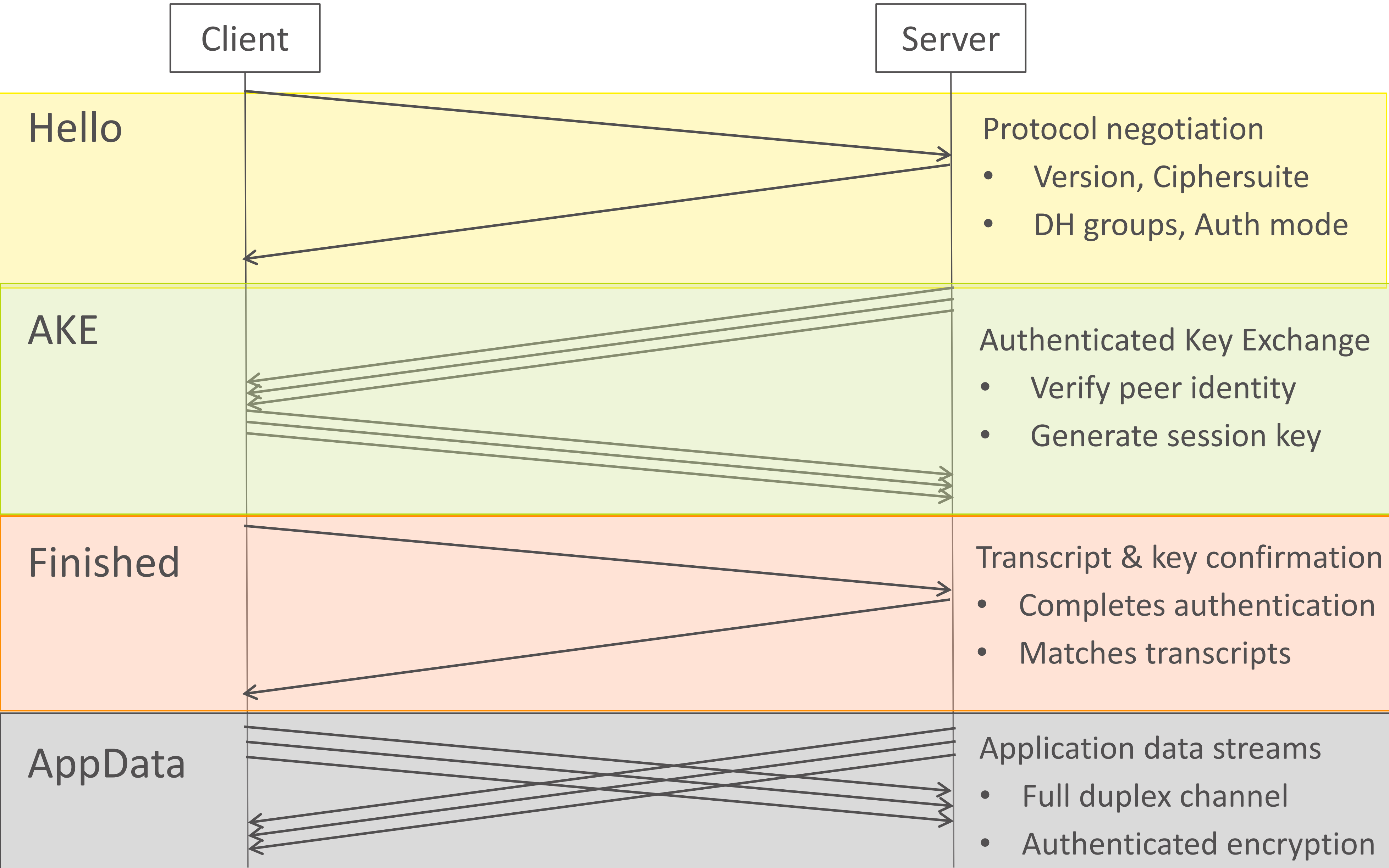
Status of This Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

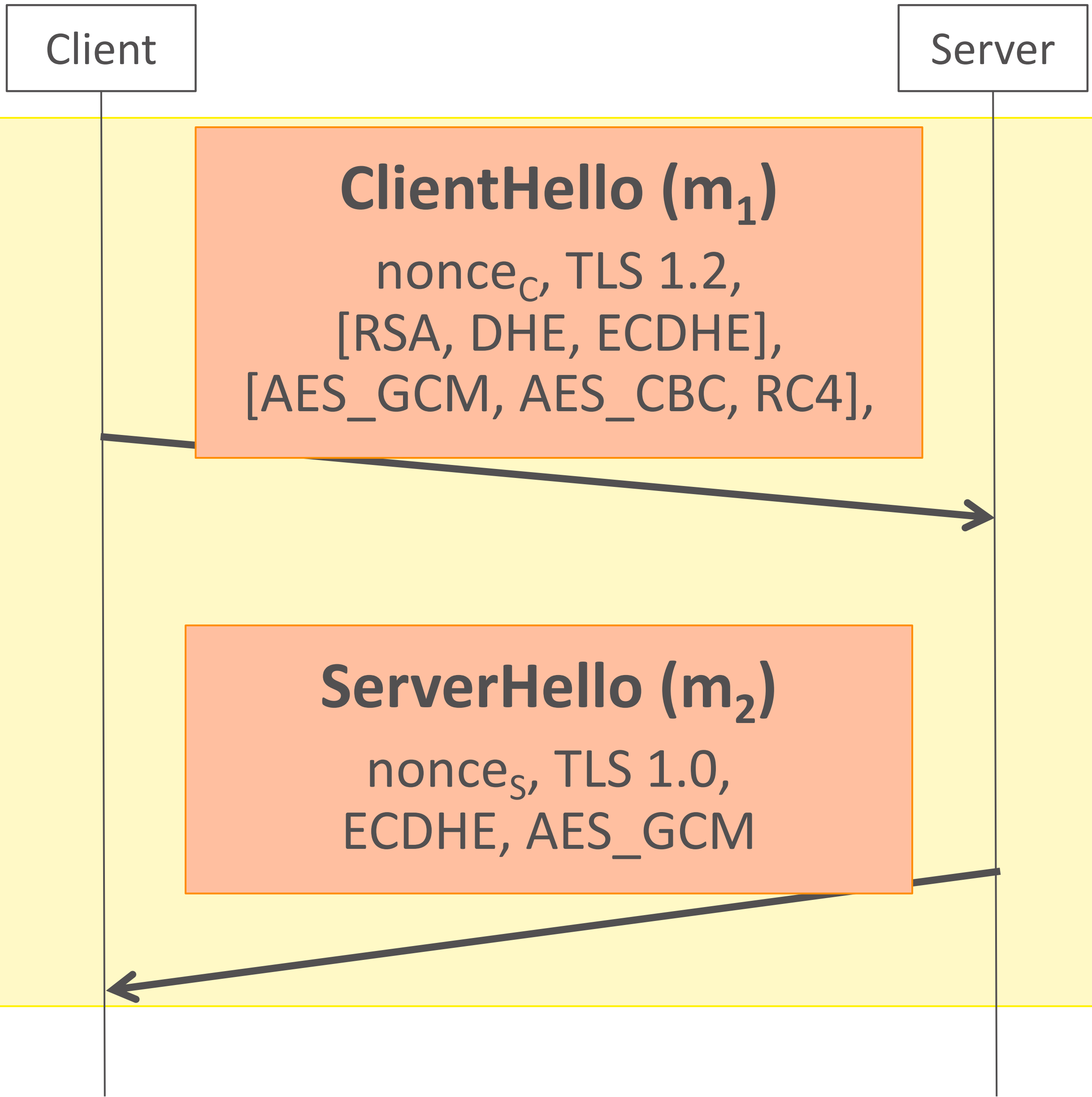
Abstract

This document specifies Version 1.2 of the Transport Layer Security (TLS) protocol. The TLS protocol provides communications security over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

TLS protocol overview



TLS negotiation



The many, many modes of TLS

Protocol versions

- TLS 1.2, TLS 1.1, TLS 1.0, SSLv3, SSLv2

Key exchanges

- ECDHE, FFDHE, RSA, PSK, …

Authentication modes

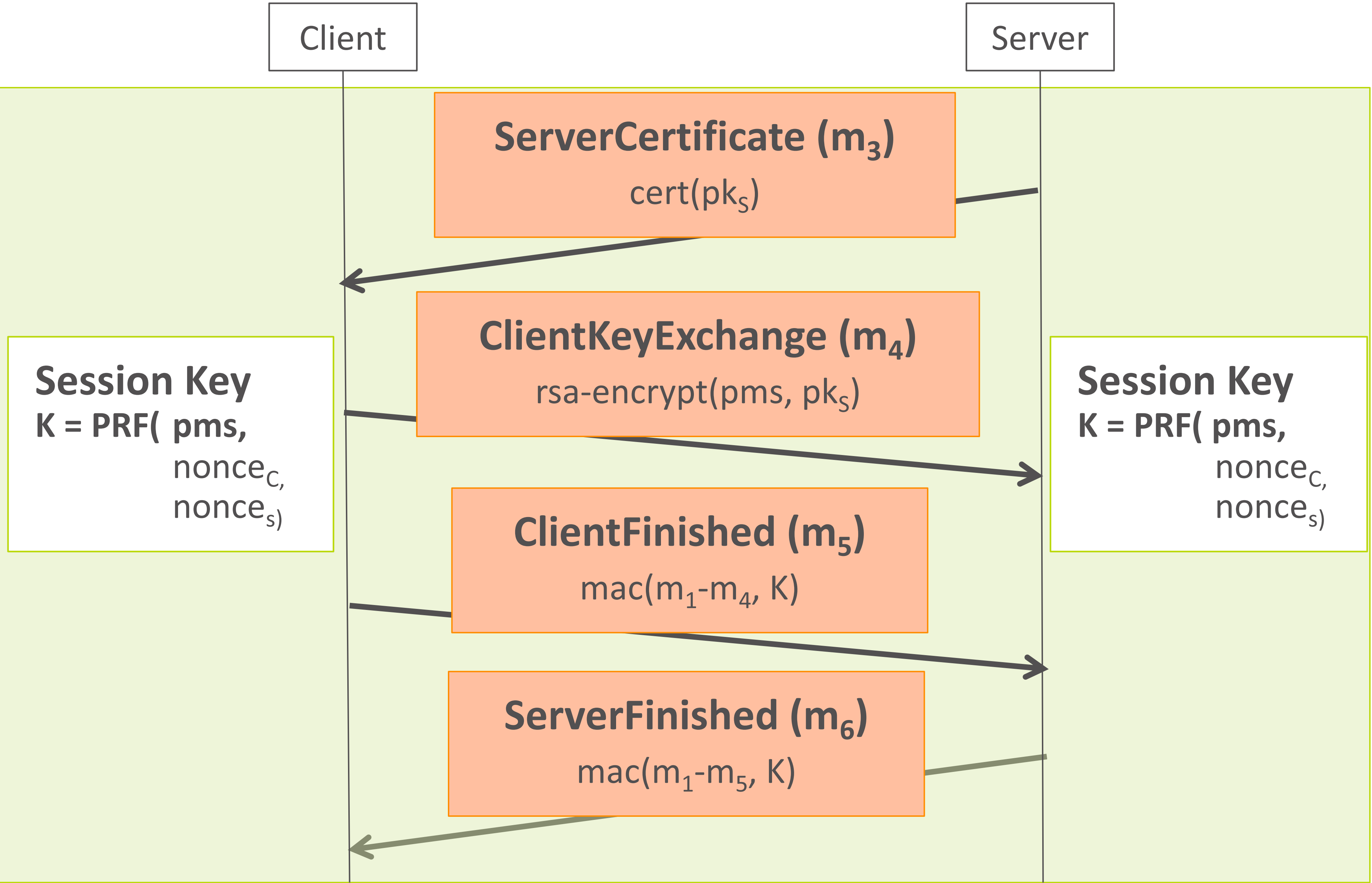
- ECDSA, RSA signatures, PSK,…

Authenticated Encryption Schemes

- AES-GCM, CBC MAC-Encode-Encrypt, RC4,…

100s of possible protocol combinations!

RSA Key Transport



RSA Key Transport

- Client chooses secret pms,
  adds maximum protocol version pv_{max},
  pads according to RSA PKCS#1 v1.5,
  and encrypts with server's public key pk_S

  $$rsa\text{-}pkcs1\text{-}encrypt(pms, pk_S) = [pad \mid pv_{max} \mid pms]^e \bmod pq$$

- Server decrypts, checks pad and protocol version,
  computes session key from pms

Security: In theory, relies on hardness of factoring pq

RSA Key Transport: Attacks and Proofs

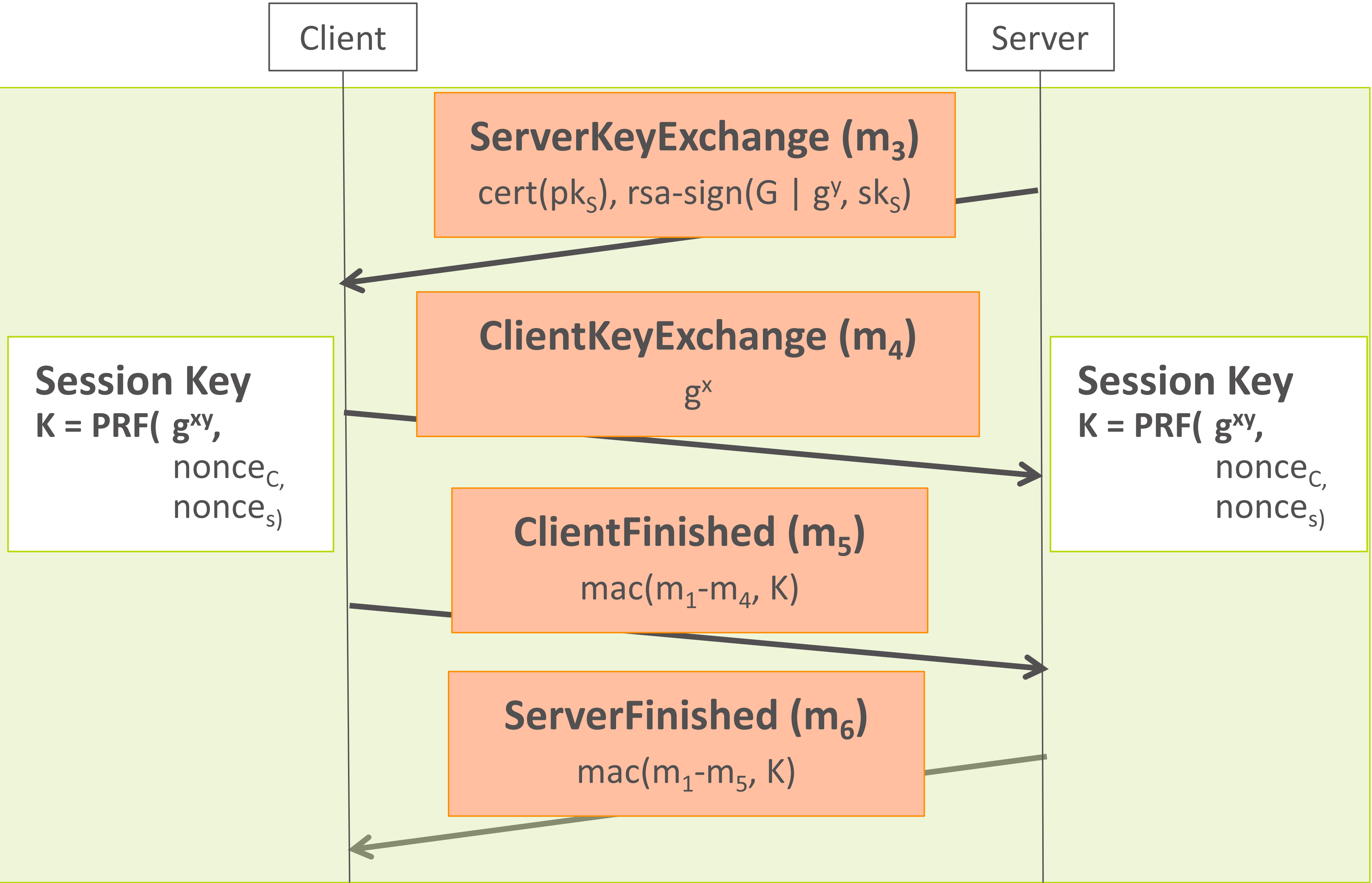
- [1994] Classic protocol, many proofs
- [1998] Chosen Ciphertext attack on PKCS#1
- [2002] Mitigations in TLS and other protocols
- [2013] Proof of TLS assuming mitigation
- [2016] DROWN: downgrade to SSLv2 + Bleichenbacher + software bugs

DROWN: Breaking TLS using SSLv2

Nimrod Aviram[1], Sebastian Schinzel[2], Juraj Somorovsky[3], Nadia Heninger[4], Maik Dankel[2], Jens Steube[5], Luke Valenta[4], David Adrian[6], J. Alex Halderman[6], Viktor Dukhovni[7], Emilia Käsper[8], Shaanan Cohney[4], Susanne Engels[3], Christof Paar[3] and Yuval Shavitt[1]

[1]Department of Electrical Engineering, Tel Aviv University

(EC)DHE Key Exchange



(EC)DHE Key Exchange

- Server chooses group (p,g) and a public value g^y and signs it with its certificate signing key sk_S :

$$\text{rsa-sign}([\text{nonce}_C \mid \text{nonce}_S \mid p \mid g \mid g^y], sk_S)$$

(Can use named elliptic curves instead of $p \mid g$)

- Classic Diffie-Hellman Key Exchange

$$pms = g^{xy} \bmod p$$

Security: In theory, relies on (some) D-H assumption

- Provides forward secrecy, preferred over RSA

(EC)DHE Key Exchange Analysis

- [1994] Classic protocol, many proofs
- [2011] Proof of mutually-authenticated DHE
- [2013] Proof of server-authenticated RSA+DHE
- [2015] Logjam: Downgrade to DHE_EXPORT + discrete logarithm + configuration bugs

Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice

David Adrian[¶] Karthikeyan Bhargavan[*] Zakir Durumeric[¶] Pierrick Gaudry[†] Matthew Green[§]
J. Alex Halderman[¶] Nadia Heninger[‡] Drew Springall[¶] Emmanuel Thomé[†] Luke Valenta[‡]
Benjamin VanderSloot[¶] Eric Wustrow[¶] Santiago Zanella-Béguelin[‖] Paul Zimmermann[†]

[*] INRIA Paris-Rocquencourt [†] INRIA Nancy-Grand Est, CNRS, and Université de Lorraine
[‖] Microsoft Research [‡] University of Pennsylvania [§] Johns Hopkins [¶] University of Michigan

For additional materials and contact information, visit WeakDH.org.

What goes wrong in TLS?

Cryptographic Weaknesses in Legacy Constructions

- Weak hash functions, weak DH groups, short block ciphers, leaky PKCS#11v1.5 padding

Logical Flaws in Protocol

- Cross-Protocol Attacks, Downgrade Attacks, Transcript Synchronization/Collision Attacks

Implementation Bugs in TLS Libraries

- Bugs in crypto library, Buffer overflows in packet parsing, Composition bugs in state machines, Bad configurations

Sometimes, a mix of all of the above!

Recall: the many modes of TLS

Protocol versions

- TLS 1.2, TLS 1.1, TLS 1.0, SSLv3, SSLv2

Key exchanges

- ECDHE, FFDHE, RSA, PSK, …

Authentication modes

- ECDSA, RSA signatures, PSK,…

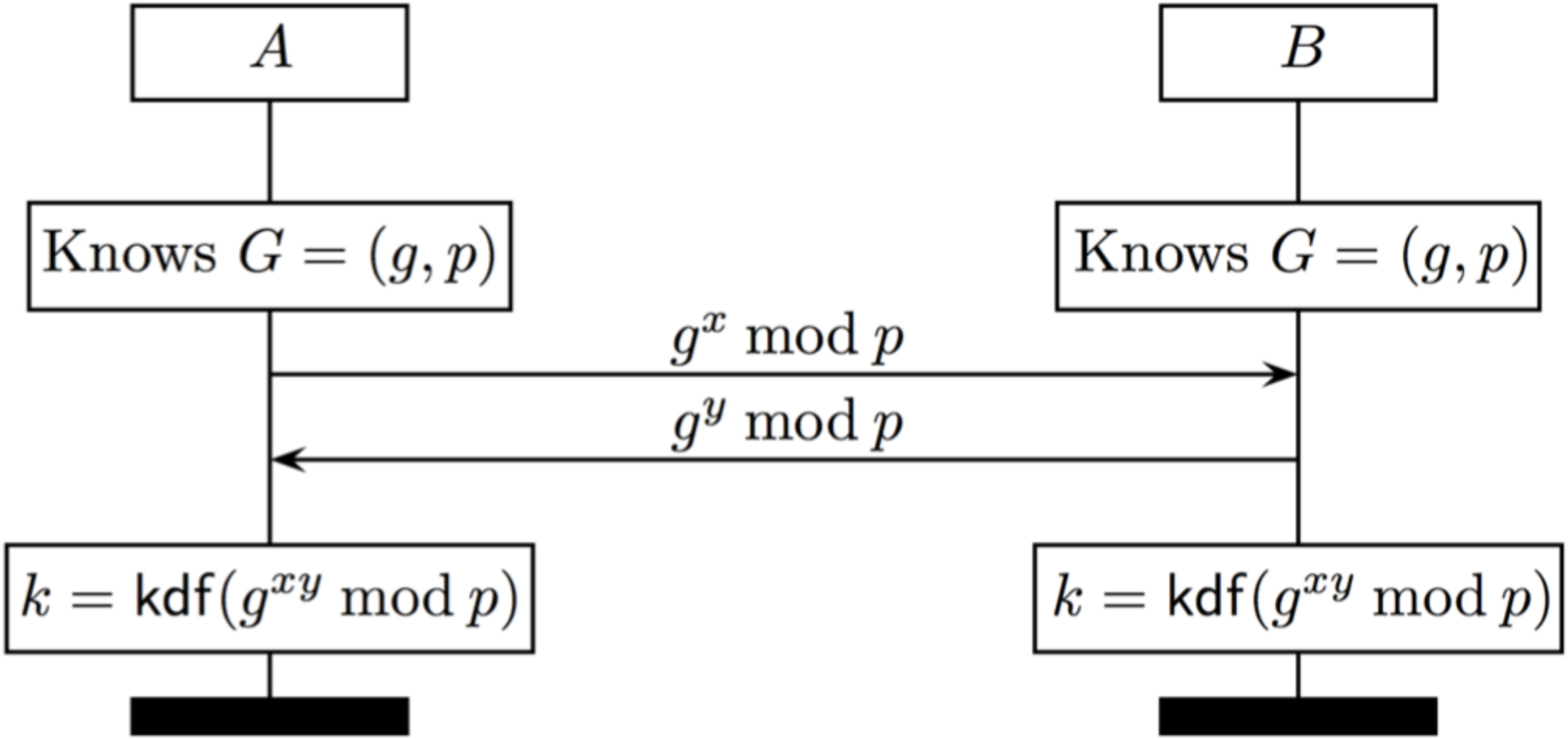
Authenticated Encryption Schemes

- AES-GCM, CBC MAC-Encode-Encrypt, RC4,…

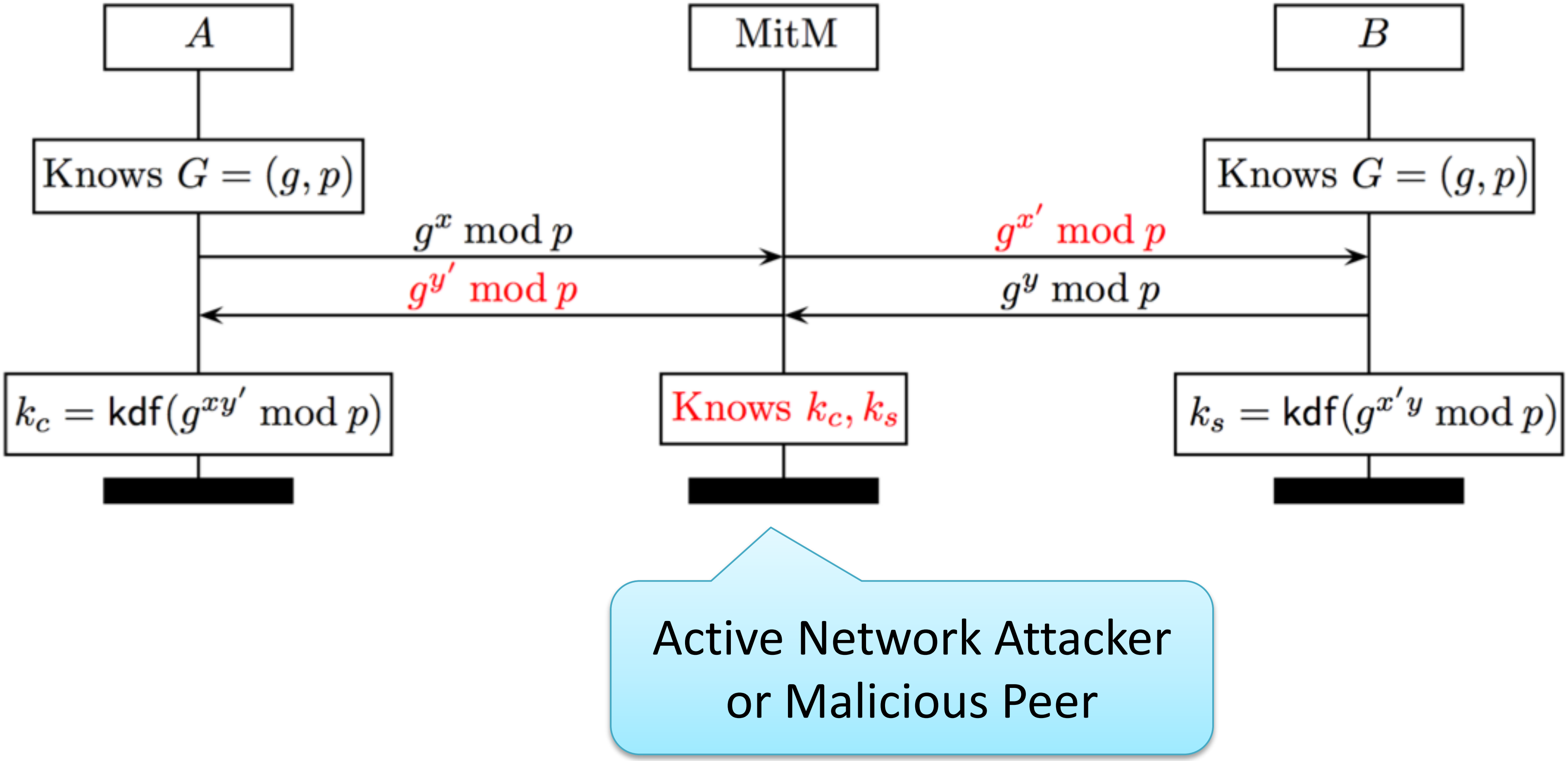
100s of possible protocol combinations!

Exploiting
Crypto Weaknesses:
Weak DH Groups

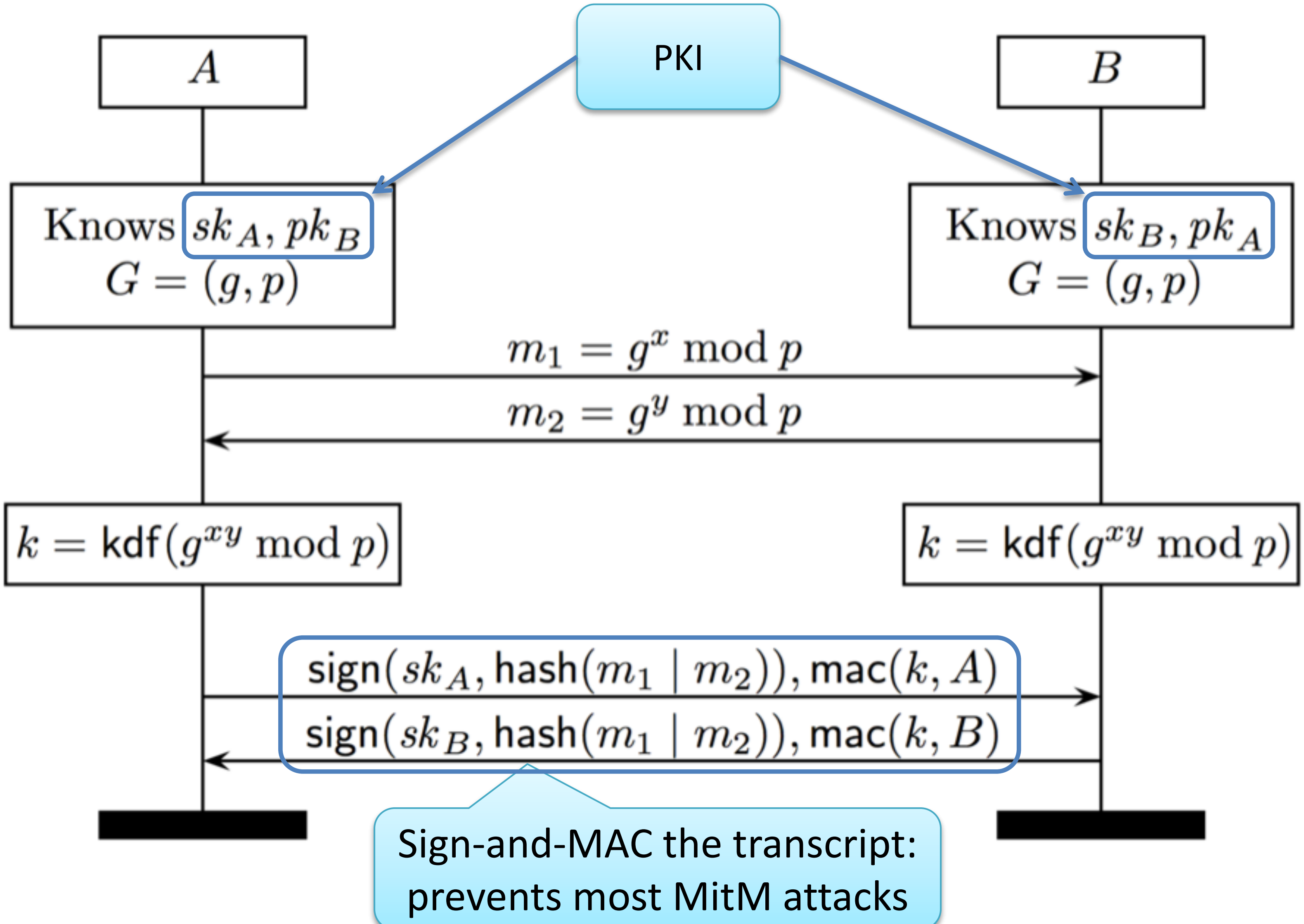
Anonymous Diffie-Hellman (ADH)



Man-in-the-Middle attack on ADH



Authenticated DH (SIGMA)



Weak Diffie-Hellman Groups

Diffie-Hellman shared secret computation

$$k = \mathrm{kdf}(g^{xy} \; mod \; p)$$

Theoretical Security:

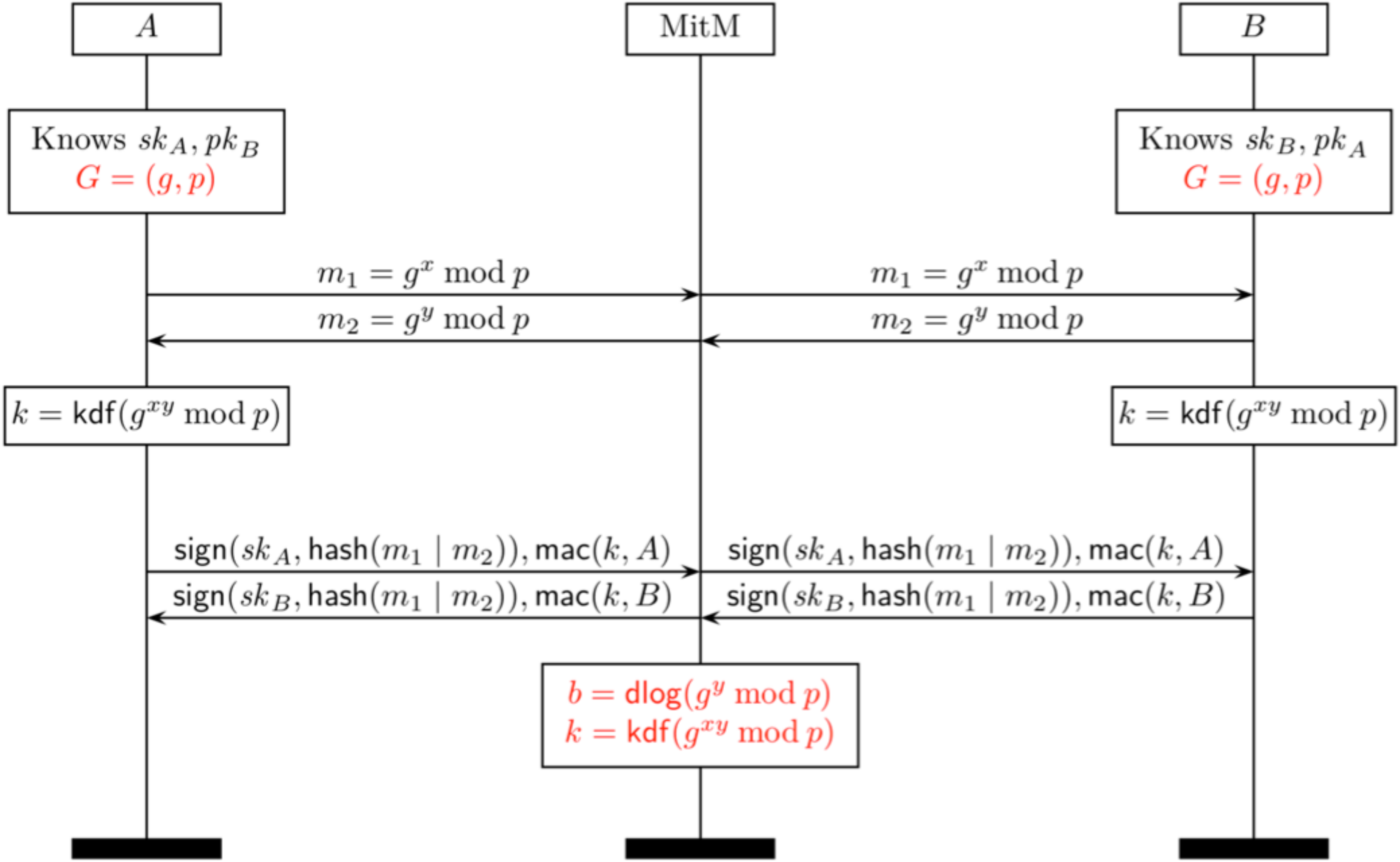
- Relies on some DH assumption (CDH, Gap, PRF-ODF,…)
- Attacker cannot compute k without knowing x or y

Attacks:

- Best known attacks rely on discrete log:

$$\boldsymbol{y = \log(g^{y} \; mod \; p)}$$

Discrete Log Attack on SIGMA



How likely is a discrete log-based attack?

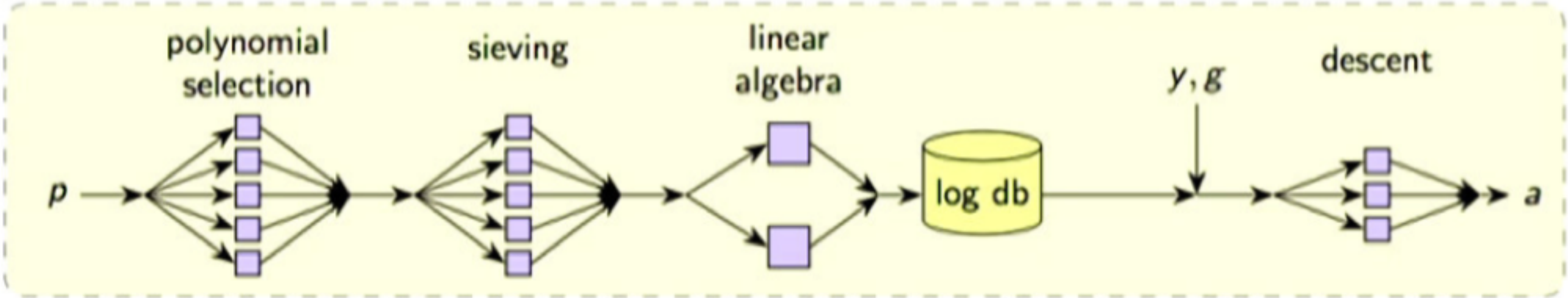
Discrete Log Computation Records

- **[Joux et al. 2005]** **431-bit prime**
- **[Kleinjung et al. 2007]** **530-bit prime**
- **[Bouvier et al. 2014]** **596-bit prime**
- **+** other results for special groups

Best known generic technique:
Number Field Sieve (NFS) and variants

Computing Discrete Logs with NFS

(slide from N. Heninger)



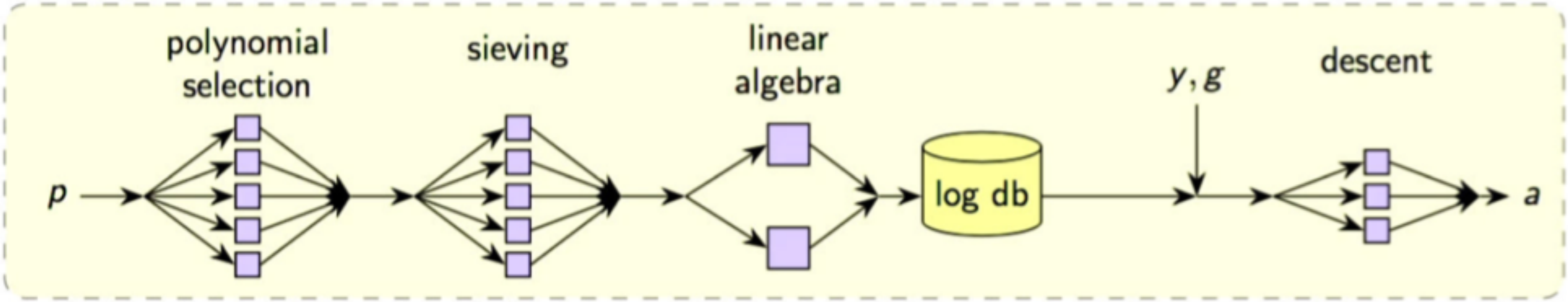
How long does the number field sieve take?

Answer 1:

$$L(1/3, 1.923) = \exp(1.923(\log N)^{1/3}(\log \log N)^{2/3})$$

Computing Discrete Logs with NFS

(slide from N. Heninger)



How long does the number field sieve take?

Answer 2:

512-bit DH: ≈ 10 core-years.

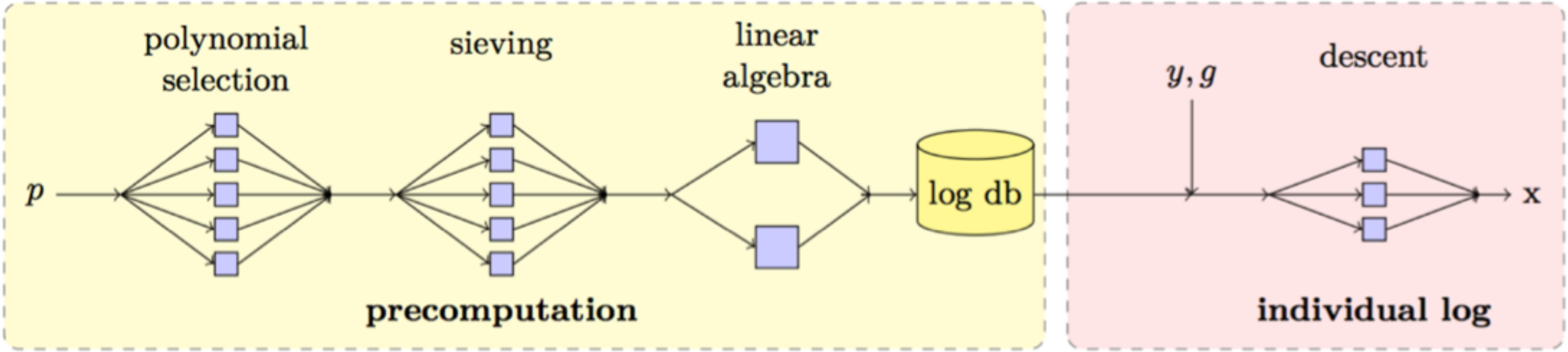
768-bit DH: ≈ 35,000 core-years.

1024-bit DH: ≈ 45,000,000 core-years.

2048-bit DH: Minimum recommended key size today.

Exploiting Pre-computation

(slide from N. Heninger)



	Sieving			Linear Algebra		Descent
	I	log B	core-years	rows	core-years	core-time
RSA-512	14	29	0.5	4.3M	0.33	
DH-512	15	27	2.5	2.1M	7.7	10 mins

Times for cluster computation:

	polysel	sieving	linalg	descent
	2000-3000 cores		288 cores	36 cores
DH-512	3 hours	15 hours	120 hours	70 seconds

TLS-DHE in practice

Internet-wide scan of HTTPS servers using Zmap (2015)

- 14.3M hosts, 24% support DHE
- 70,000 distinct groups *(p,g)*

Small-sized prime groups

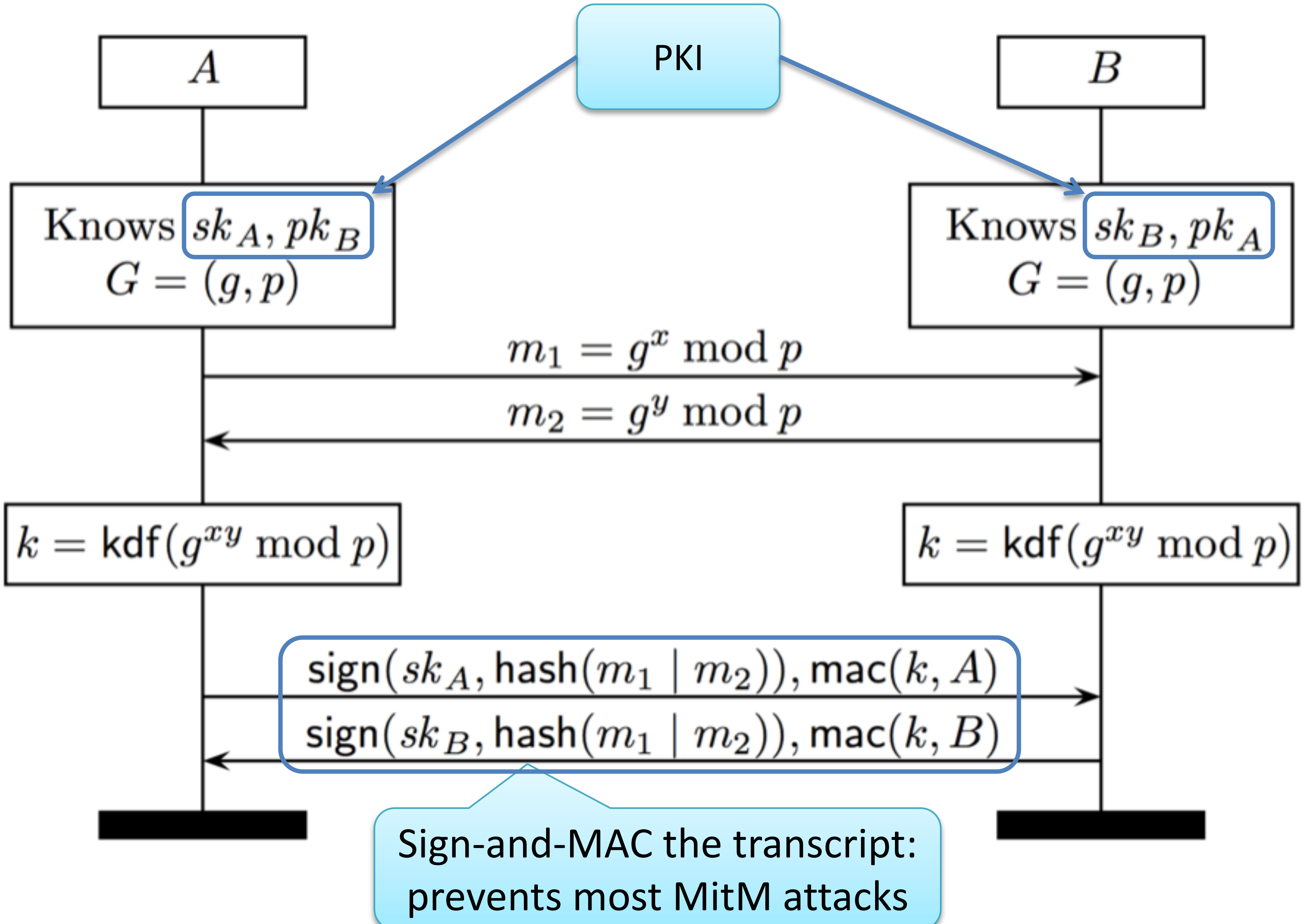
- 84% (2.9M) servers use 1024-bit primes
- 2.6% (90K) servers use 768-bit primes
- 0.0008% (2.6K) servers use 512-bit primes

What percentage of the internet does our TLS-DHE cryptographic proofs apply to?

- Depends on how powerful your adversary is

Exploiting
Crypto Weaknesses:
Weak Hash Functions

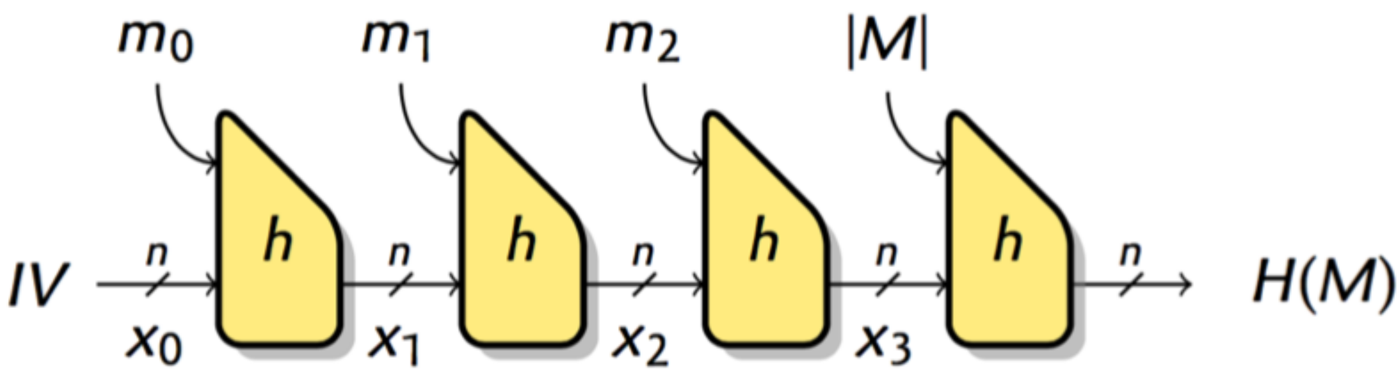
Authenticated DH (SIGMA)



Authentication via Transcript Signatures

- Sign the full transcript
  - **sign**(sk_B, **hash**($m_1 \mid m_2$))
  - *Example*: TLS 1.3, SSH-2, TLS 1.2 client auth
- How weak can the **hash** function be?
  - do we need collision resistance?
  - do we only need 2nd preimage resistance?

Quick Primer on Hash Functions

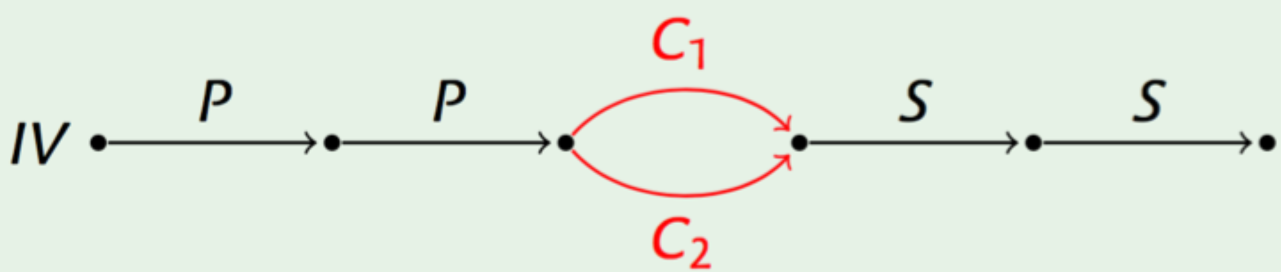


- Hash function: public function $\{0,1\}^* \to \{0,1\}^n$
  - Maps arbitrary-length message to fixed-length hash
- Mekle-Damgård mode: n-bit chain value
  - Process message iteratively
  - Use the message length in the padding (MD strengthening)
- Hash function should behave like a random function
  - Hard to find collisions, preimages
  - Hash can be used as a fingerprint

Hash Function Cryptanalysis

Collision attack

- Find $M_1 \neq M_2$ such that $H(M_1) = H(M_2)$
- Generic attack with complexity $2^{n/2}$ (expected security)
- Shortcut attacks
  - MD5: complexity 2^{16} [Wang & *al.*'05, Stevens & *al.*'09]
  - SHA1: complexity 2^{61} [Wang & *al.*'05, Stevens '13]

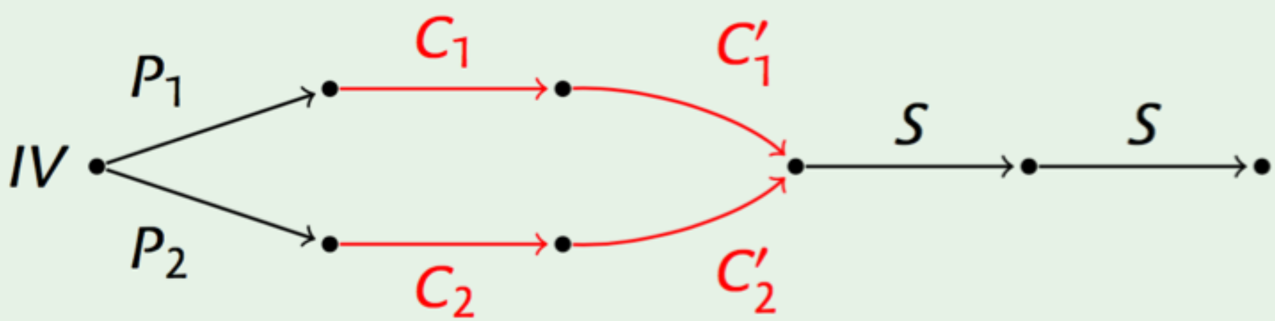


- Arbitrary common prefix/suffix, random collision blocks

Hash Function Cryptanalysis

Chosen-prefix collision attack

- Given P_1, P_2, find $M_1 \neq M_2$ such that $H(P_1 \| M_1) = H(P_2 \| M_2)$
- Generic attack with complexity $2^{n/2}$ (expected security)
- Shortcut attacks
  - MD5: complexity 2^{39} [Stevens & *al.*'09]
  - SHA1: complexity 2^{77} [Stevens '13]



Hash Function Cryptanalysis

2^{nd} preimage attack

- Given M_1, $H(M_1)$, find $M_2 \neq M_1$ s.t. $H(M_1) = H(M_2)$
- Generic attack with complexity 2^n (expected)
  - MD5: complexity 2^{128}
  - SHA1: complexity 2^{160}
  - No practical attacks

- Protocols that rely only on 2^{nd} preimage resistance can safely use even MD5
  - E.g. public key fingerprints in SSH

Hash Function Attack Complexity

- MD5: known attack complexities
  - **MD5** second preimage 2^{128} hashes (infeasible)
  - **MD5** generic collision: 2^{64} hashes (months?)
  - **MD5** chosen-prefix collision: 2^{39} hashes (1 hour)
  - **MD5** common-prefix collision: 2^{16} hashes (seconds)

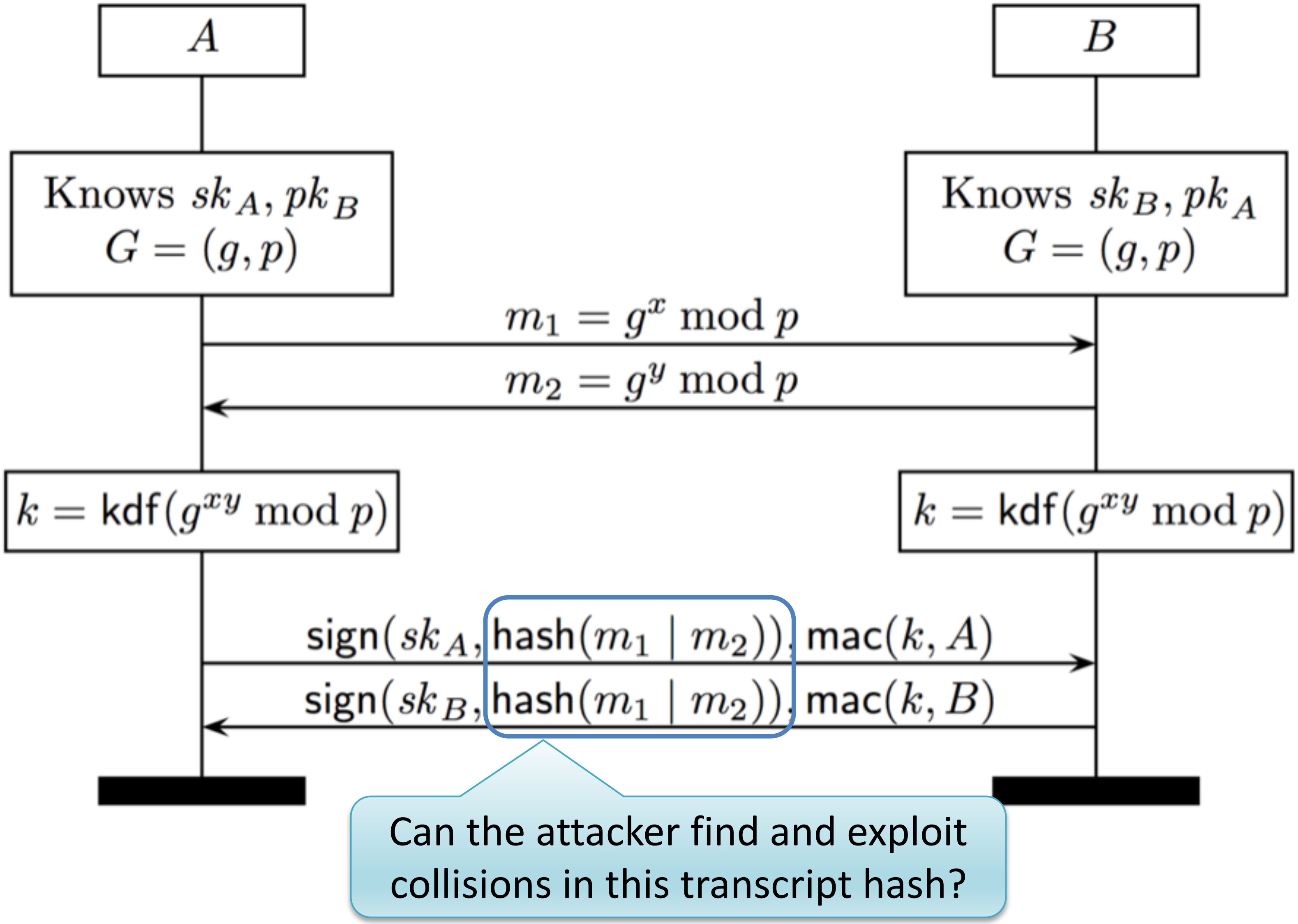
- SHA1: estimated attack complexities
  - **SHA1** second preimage 2^{160} hashes (infeasible)
  - **SHA1** generic collision: 2^{80} hashes (infeasible)
  - **SHA1** chosen-prefix collision: 2^{77} hashes (?)
  - **SHA1** common-prefix collision: 2^{61} hashes (months)

Authentication via Transcript Signatures

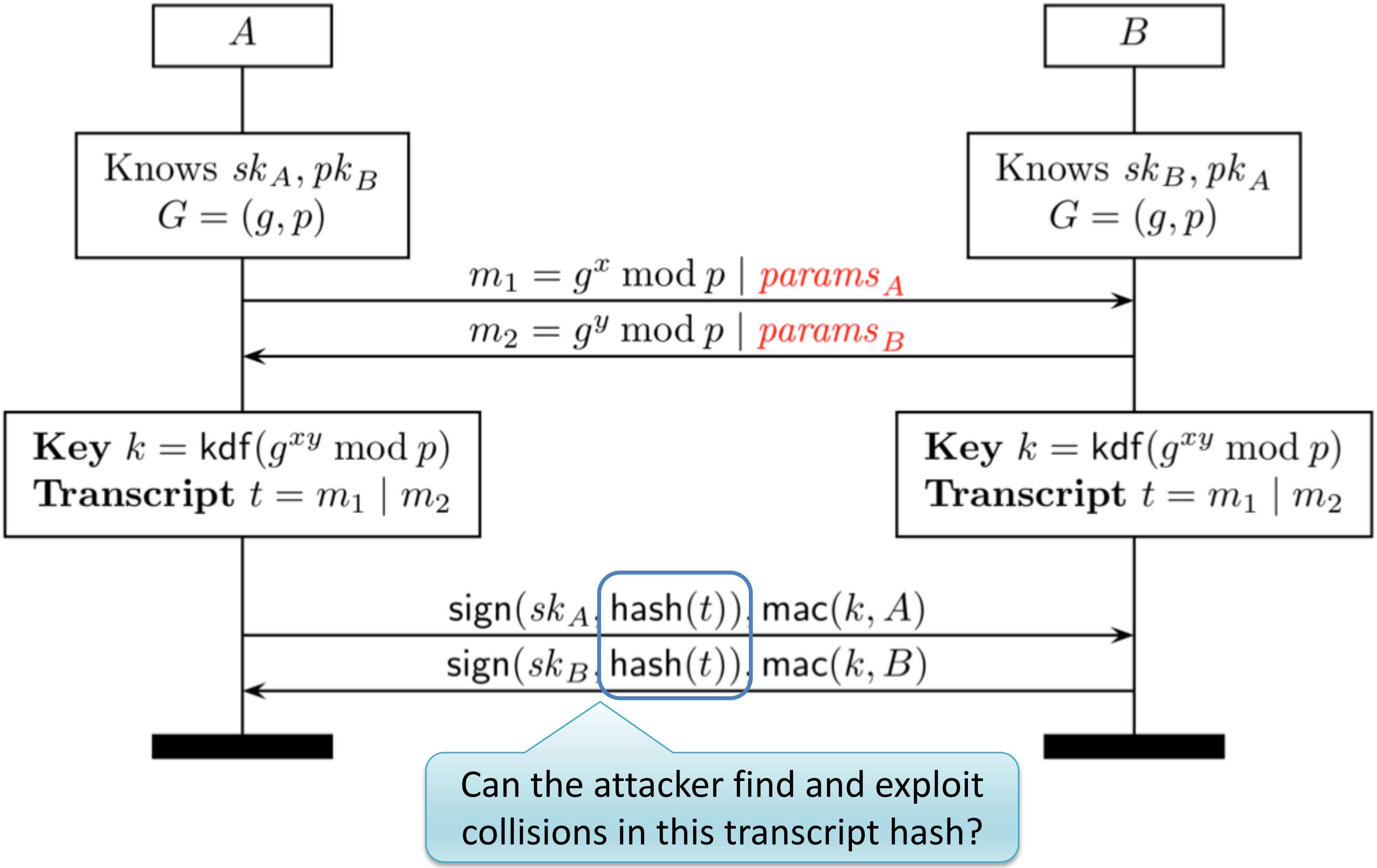
- Sign the full transcript
  - **sign**(sk_B, **hash**($m_1 \mid m_2$))
  - *Example*: TLS 1.3, SSH-2, TLS 1.2 client auth

- How weak can the **hash** function be?
  - do we need collision resistance?
  - do we only need 2nd preimage resistance?
- Is it still safe to use MD5, SHA-1 in TLS, IKE, SSH?
  - *Disagreement*: cryptographers vs. practitioners (see Schneier vs. Hoffman, RFC4270)

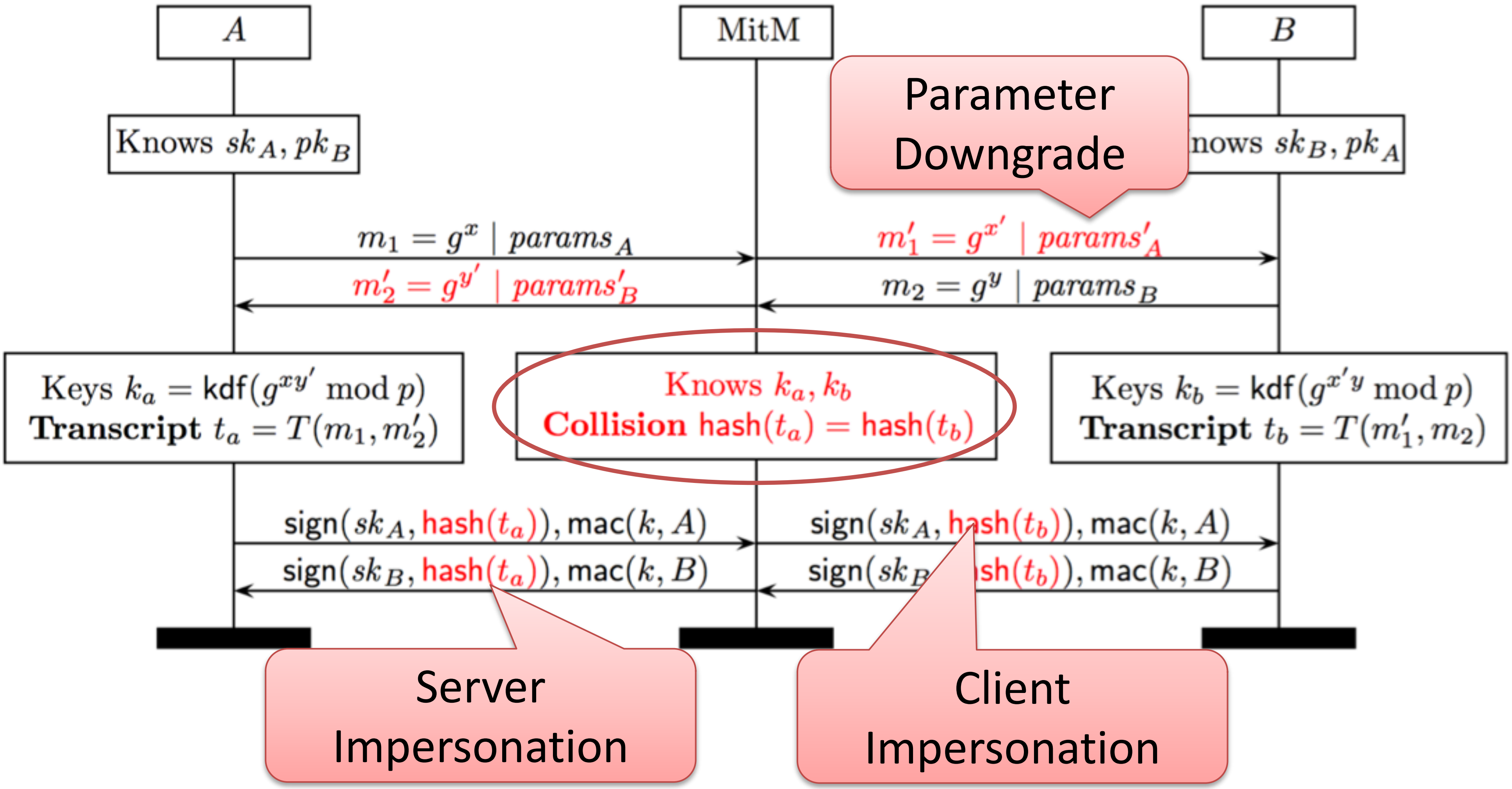
Transcript Collisions on SIGMA



Hash Collisions in SIGMA



SLOTH: Transcript Collision Attacks



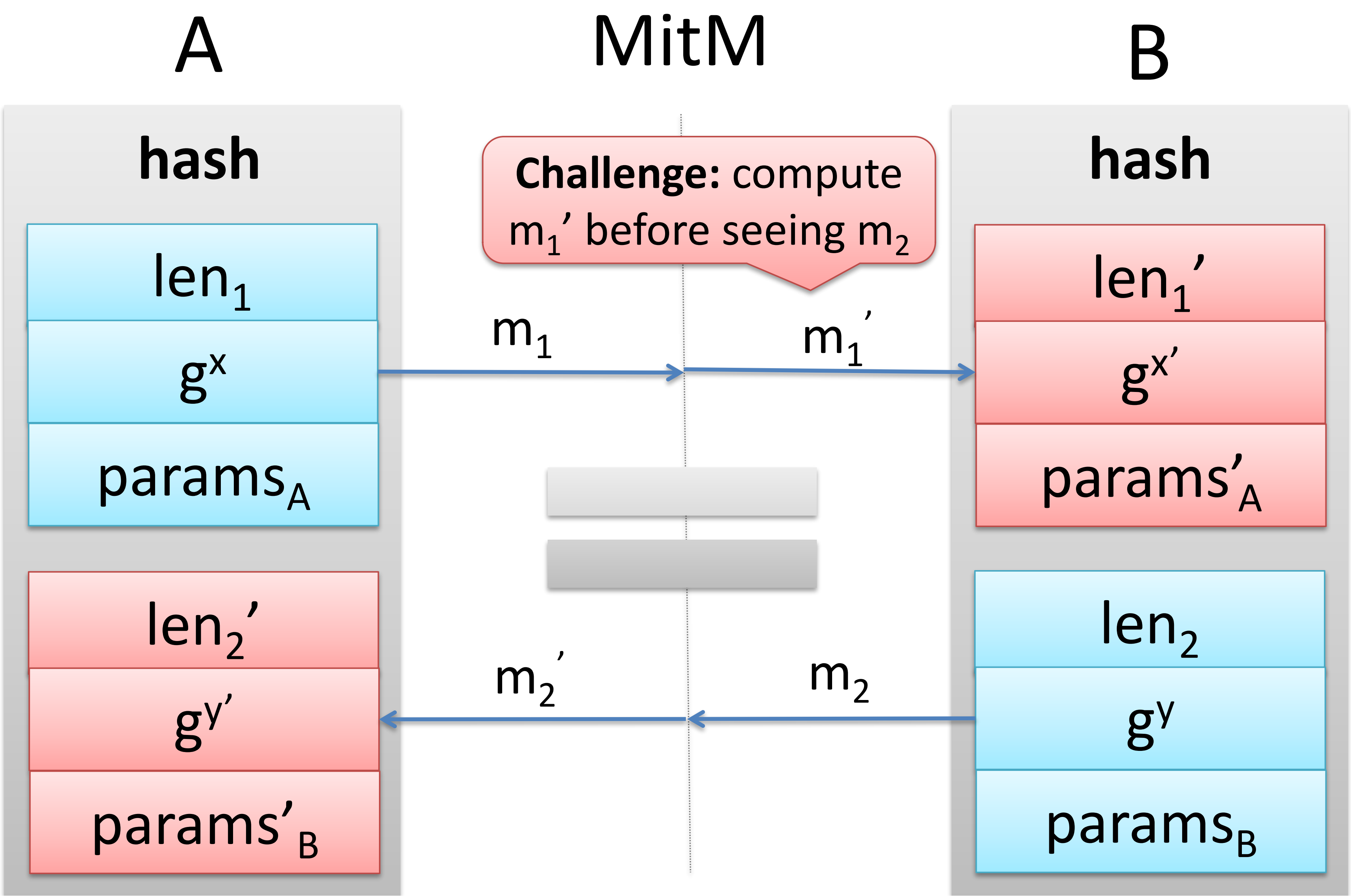
Computing a Transcript Collision

$$\textbf{hash}(m_1 \mid m'_2) = \textbf{hash}(m'_1 \mid m_2)$$

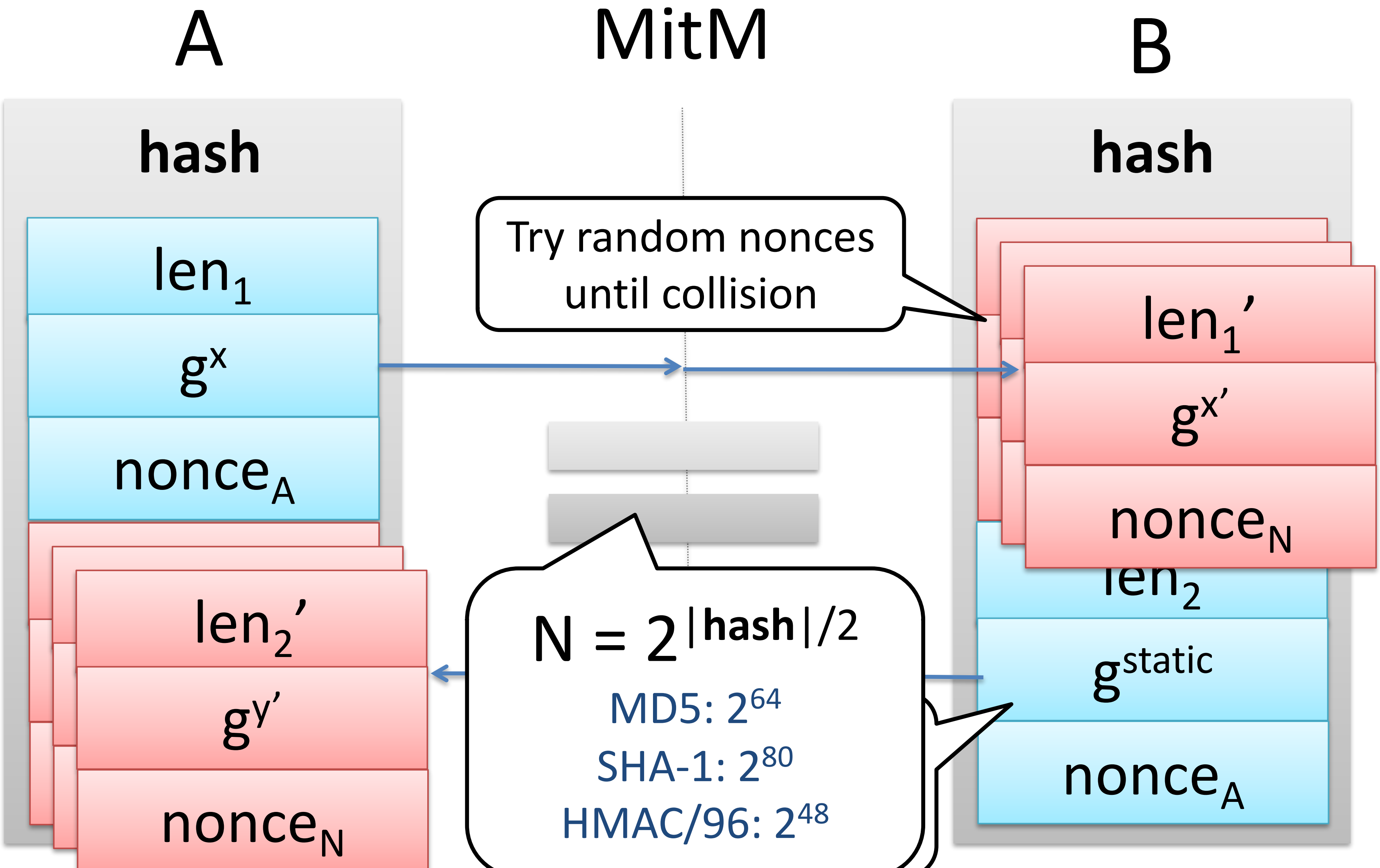
- We need to compute a collision, *not a pre-image*
  - Attacker controls parts of both transcripts
  - If we know the black bits, can we compute the red bits?
  - This can sometimes be set up as a **generic collision**

- If we're lucky, we can set up a **shortcut** collision
  - **Common-prefix**: collision after a shared transcript prefix
  - **Chosen-prefix:** collision after attacker-controlled prefixes

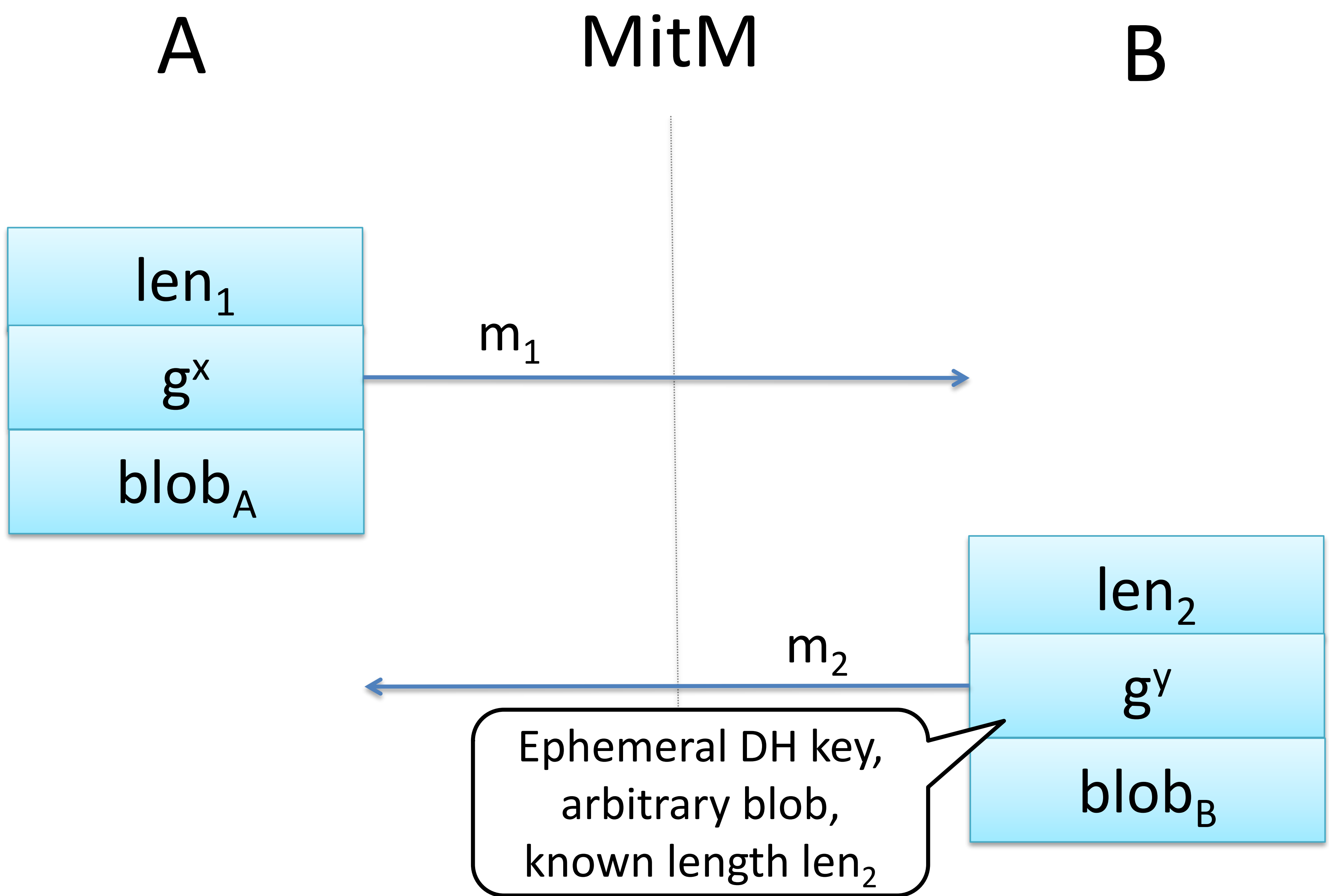
Computing Transcript Collisions

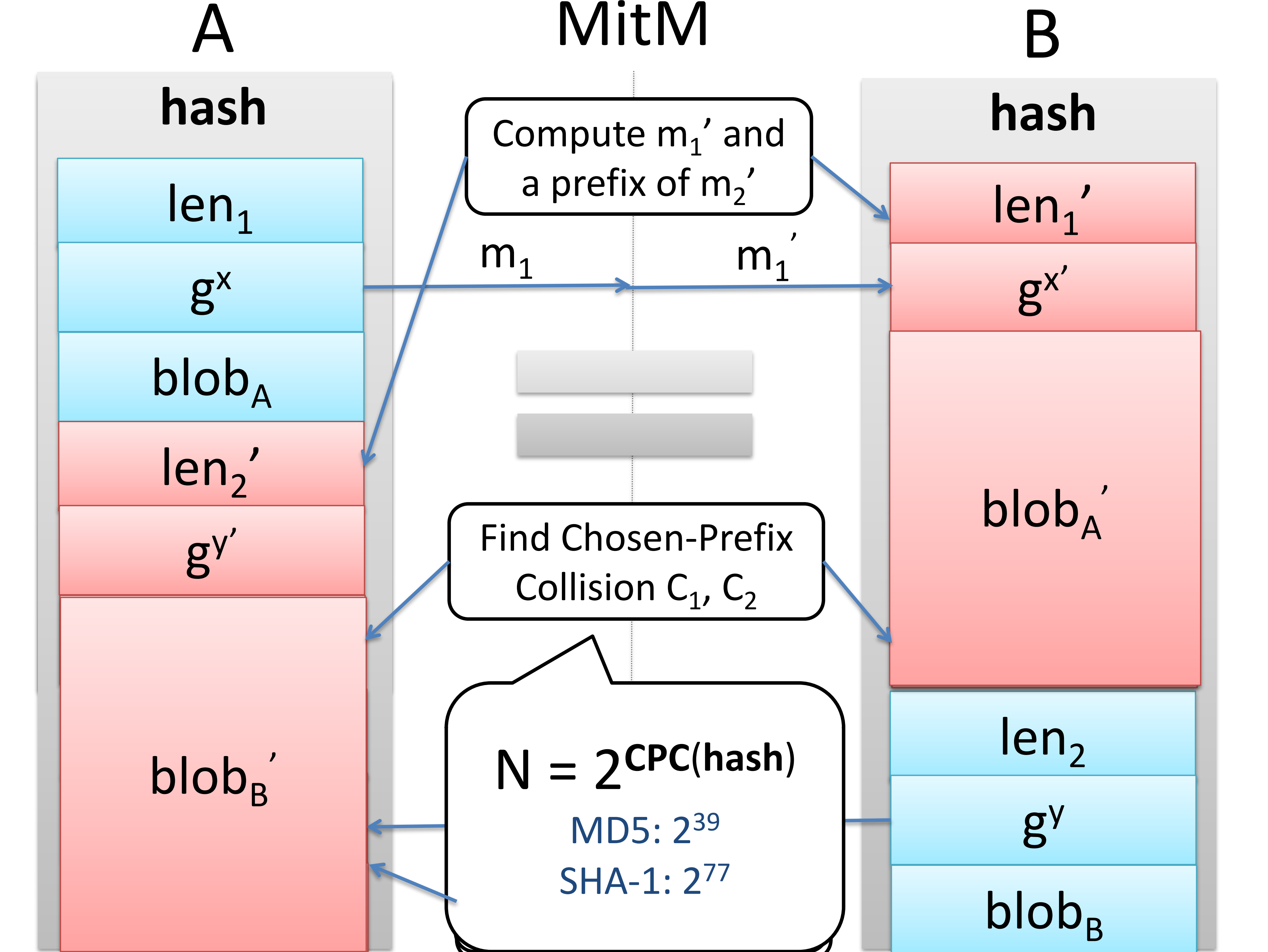


Generic Transcript Collisions



Chosen-Prefix Transcript Collisions





A
MitM
B
hash
hash
Compute m1' and a prefix of m2'
len1
gx
blobA
len2'
gy'
blobB'
m1
m1'
len1'
gx'
blobA'
len2
gy
blobB
Find Chosen-Prefix Collision C1, C2
N = 2CPC(hash)
MD5: 239
SHA-1: 277

Weak Hash Functions in TLS

TLS <= 1.1 uses MD5 and SHA-1 for signatures

- RSA signatures over MD5(t) || SHA-1(t)
- DSA signatures over SHA-1(t)

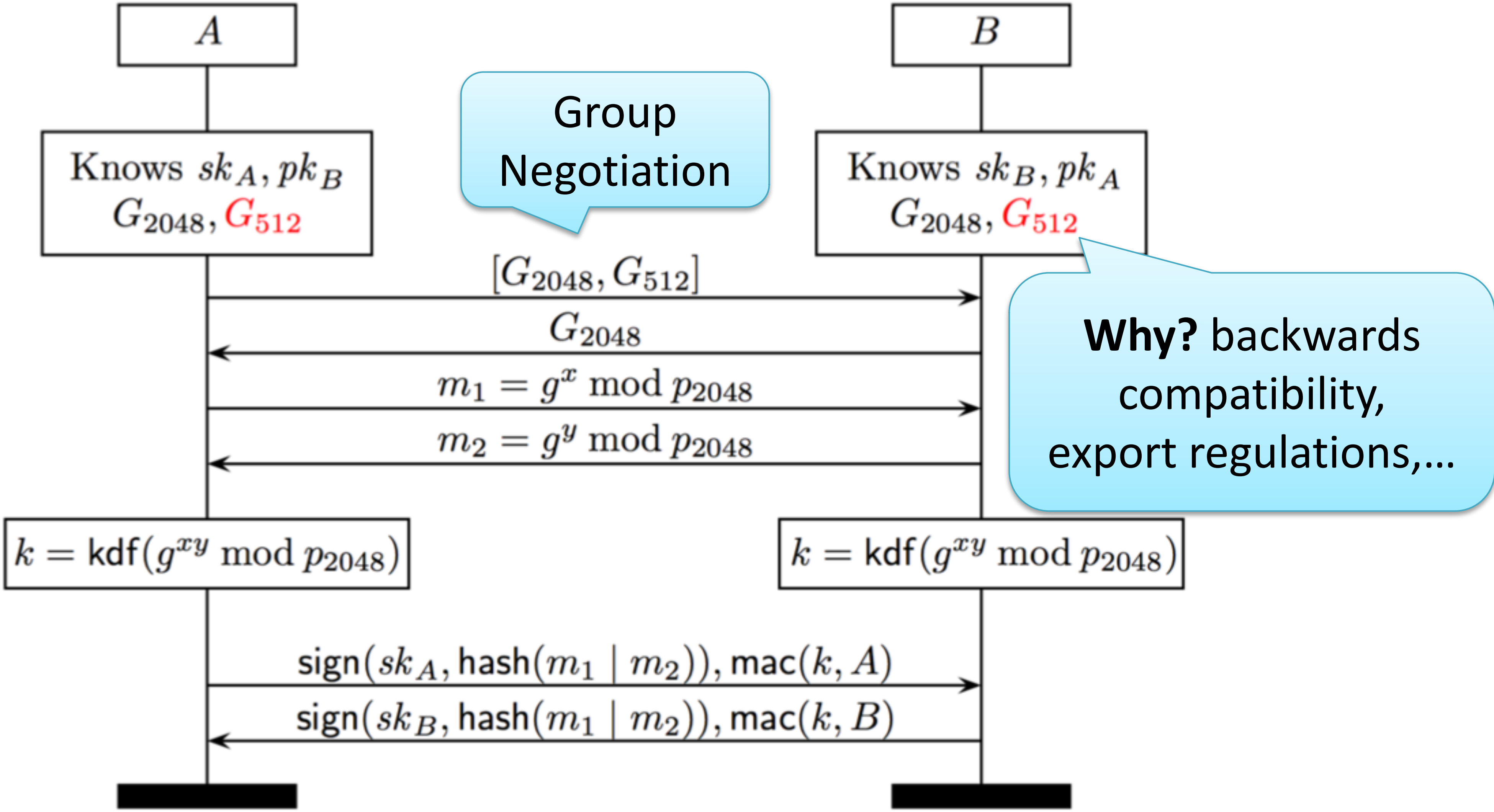
TLS 1.2 introduces signatures with SHA-2 but allows negotiation of MD5, SHA-1

- RSA signatures over MD5(t), or SHA-1(t),
  or SHA-256(t), or SHA-224(t), or SHA-384(t), or SHA-512(t)
- (EC)DSA signatures only over SHA-1(t)

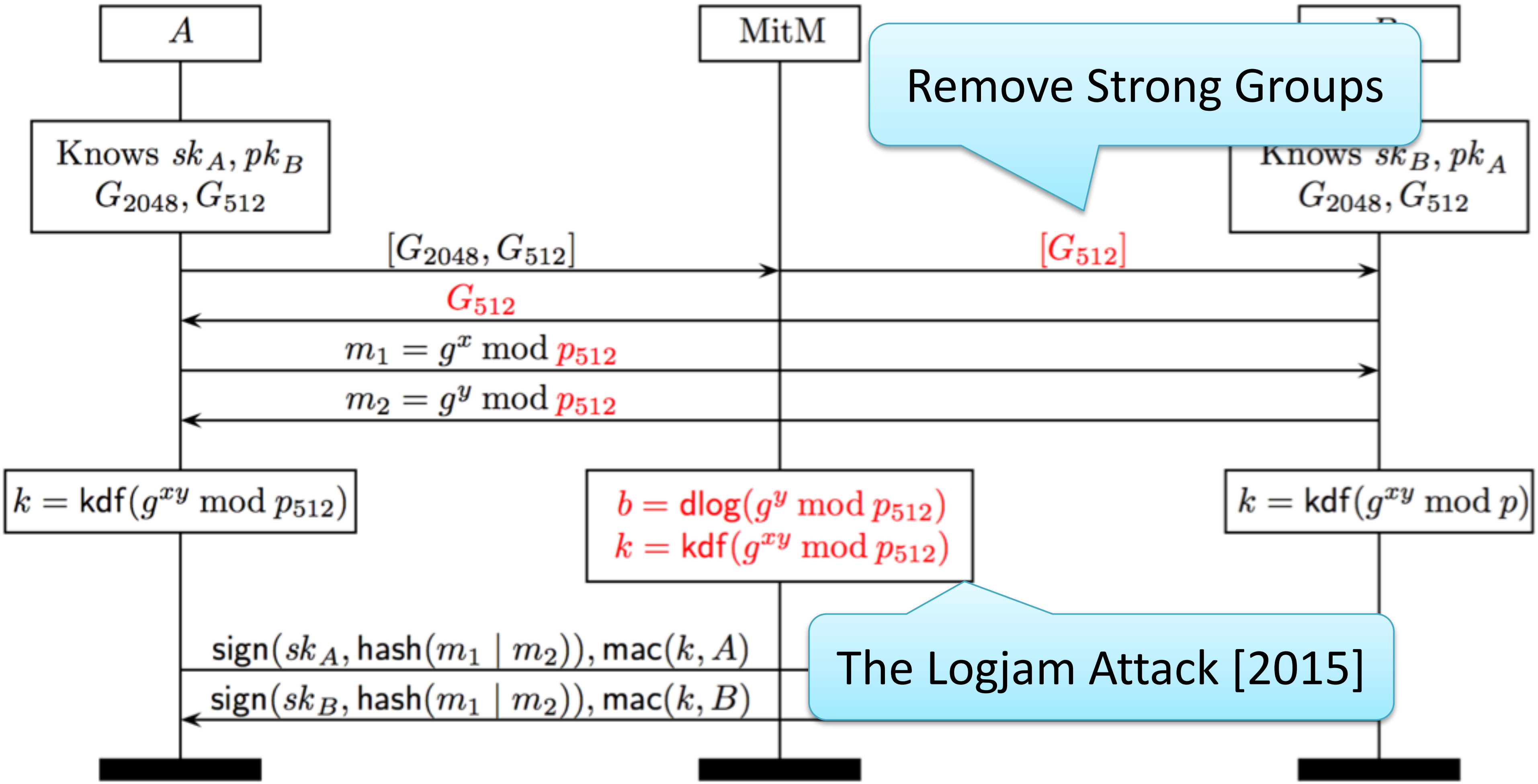
TLS 1.2 client signatures using RSA-MD5 are vulnerable to transcript collision attacks

Exploiting
Logical Flaws:
Downgrade Attacks on Agile Key Exchange

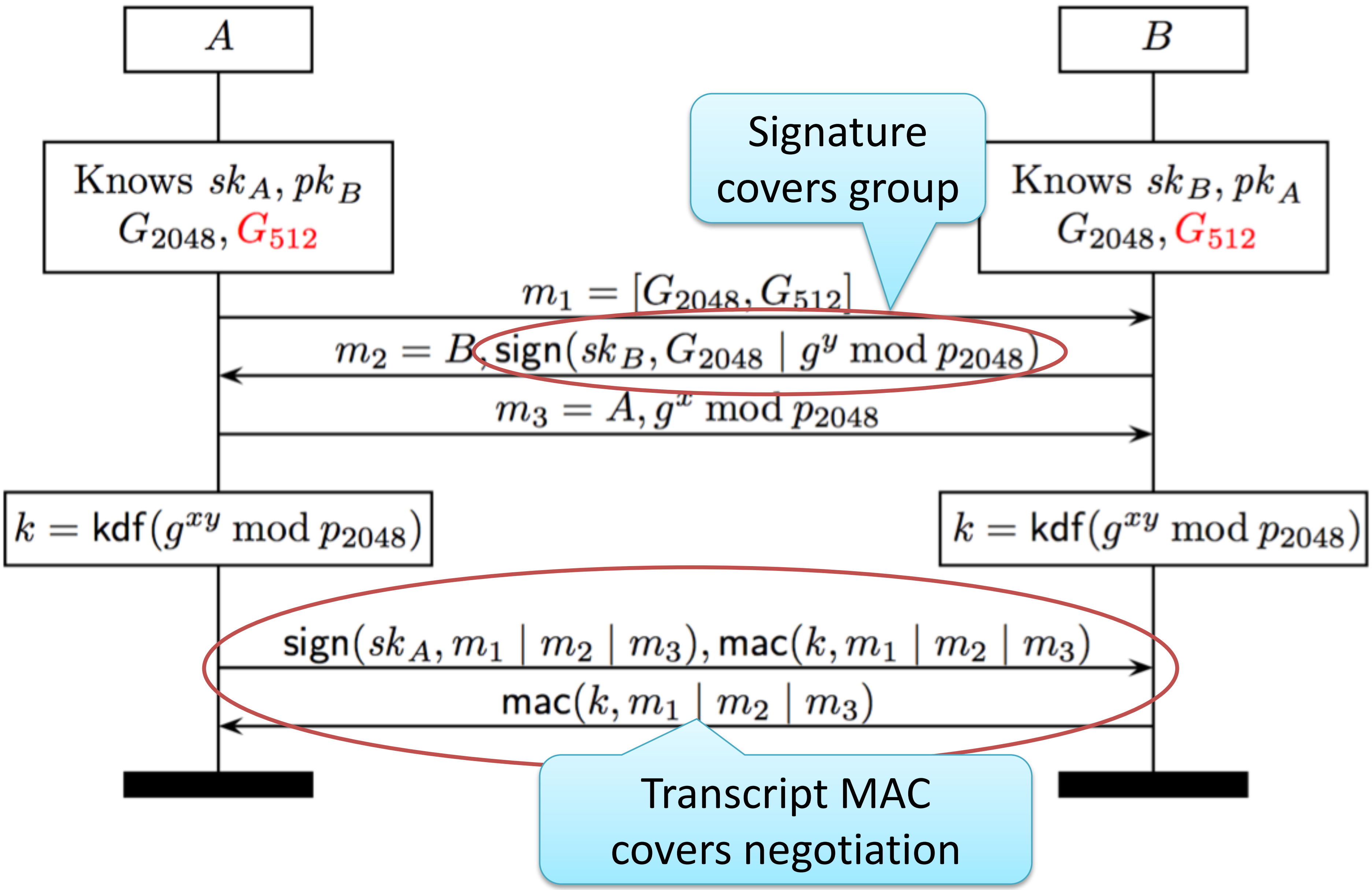
Agility: Negotiating DH Groups



Logjam: DH Group Downgrade Attack



TLS Variant of SIGMA

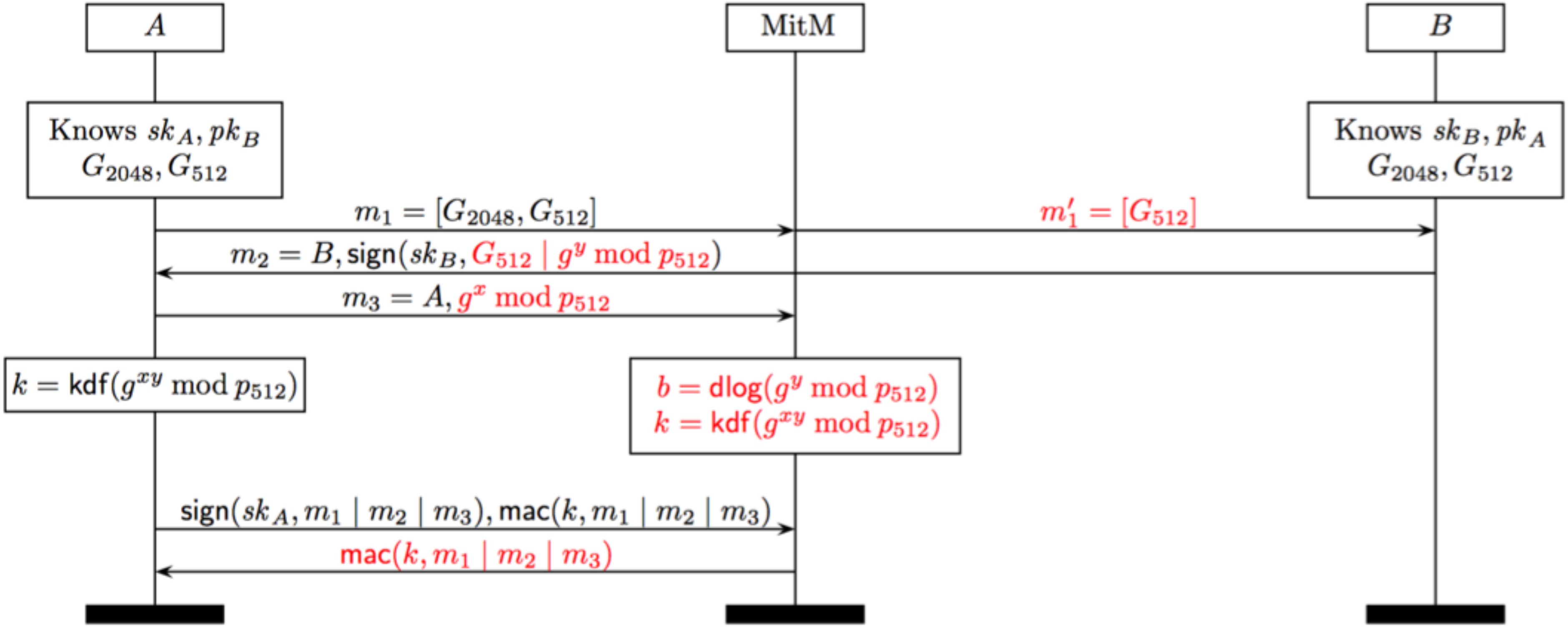


MACing the Handshake Transcript

TLS 1.2: mac the full transcript
to prevent tampering

– **mac**(k, $[G_{2048}, G_{512}]$ | G_{512} | m_1 | m_2)

Logjam Still Works



MACing the Handshake Transcript

TLS 1.2: mac the full transcript
to prevent tampering

- **mac**(k, $[G_{2048}, G_{512}]$ | G_{512} | m_1 | m_2)
- but it is too late, because we already used G_{512}
  k = **kdf**(g^{xy} mod p_{512})
- so, the attacker can forge the **mac**

- *The TLS 1.2 downgrade protection mechanism itself depends on downgradeable parameters.*
  - hence, the only fix is to find and disable all weak parameters: groups, curves, mac algorithms,...

What went wrong?

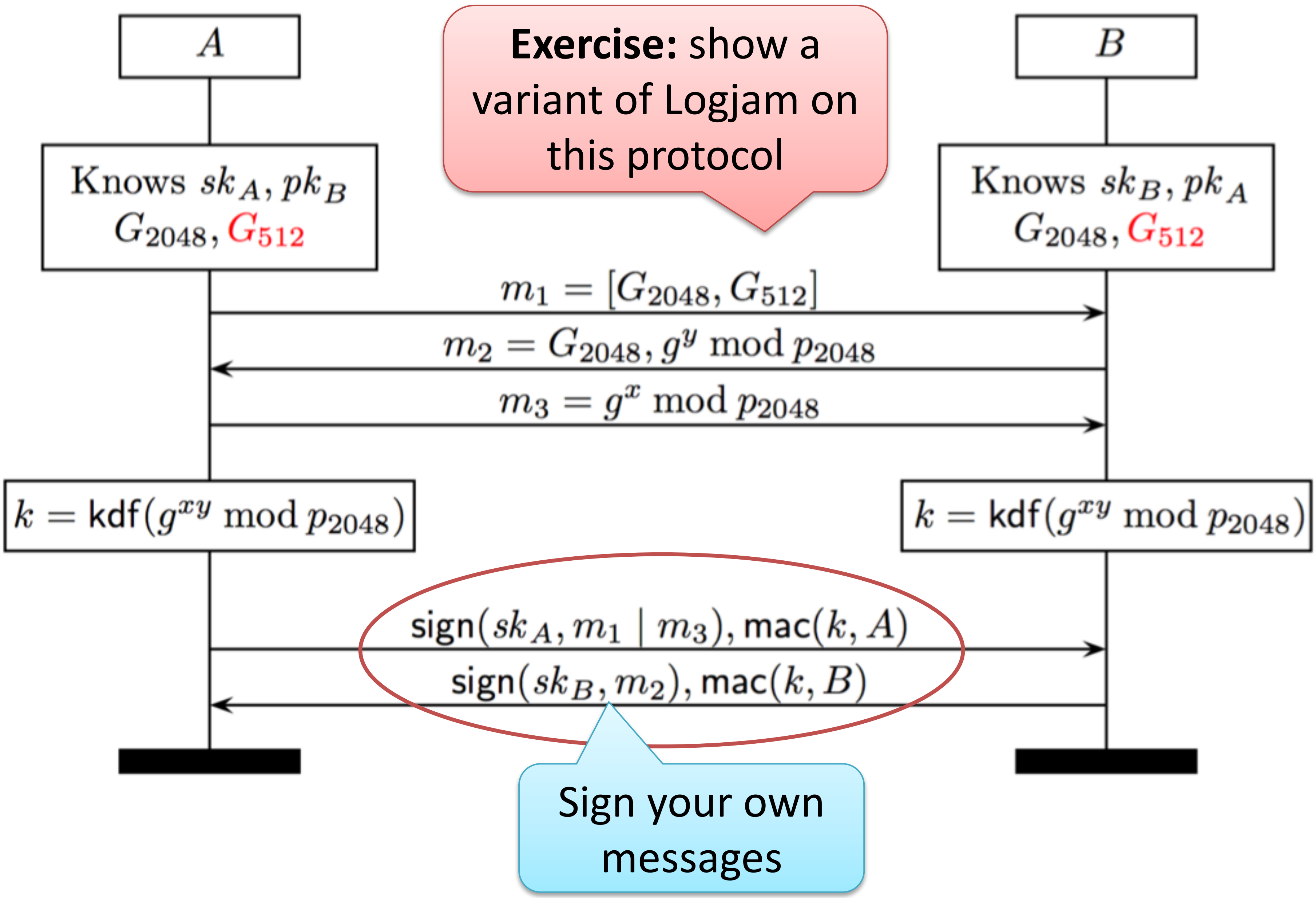
- Cryptographic weakness
  - Problem: Continued support for weak DH groups
  - Countermeasure: Ban all weak groups

- Logical protocol flaw
  - Problem: Downgrade attack on agile key exchange
  - Countermeasure: Protect integrity of key exchange even if the negotiated DH group is weak

Signing the Handshake Transcript

- **IKEv1**: both A and B sign the offered groups
  - $\mathbf{sign}(sk_B, \mathbf{hash}([G_{2048}, G_{512}] \mid m_1 \mid m_2))$
- **IKEv2**: each signs its own messages
  - $\mathbf{sign}(sk_A, \mathbf{hash}([G_{2048}, G_{512}] \mid m_1))$
  - $\mathbf{sign}(sk_B, \mathbf{hash}(G_{512} \mid m_2))$
- **SSH-2 and TLS 1.3**: sign everything
  - $\mathbf{sign}(k, \mathbf{hash}([G_{2048}, G_{512}] \mid G_{512} \mid m_1 \mid m_2))$

IKEv2 Variant of SIGMA



Signing the Handshake Transcript

- **IKEv1**: both A and B sign the offered groups
  - **sign**(sk_B, **hash**($[G_{2048},G_{512}] \mid m_1 \mid m_2$))
  - no agreement on chosen group!
- **IKEv2**: each signs its own messages
  - **sign**(sk_A, **hash**($[G_{2048},G_{512}] \mid m_1$))
  - **sign**(sk_B, **hash**($G_{512} \mid m_2$))
  - no agreement on offered groups!
- **SSH-2 and TLS 1.3**: sign everything
  - **sign**(k, **hash**($[G_{2048},G_{512}] \mid G_{512} \mid m_1 \mid m_2$))
  - works! (only if **hash** is collision-resistant)

Hash Function Downgrade (SLOTH)

TLS 1.2 introduces signatures with SHA-2

but allows negotiation of MD5, SHA-1

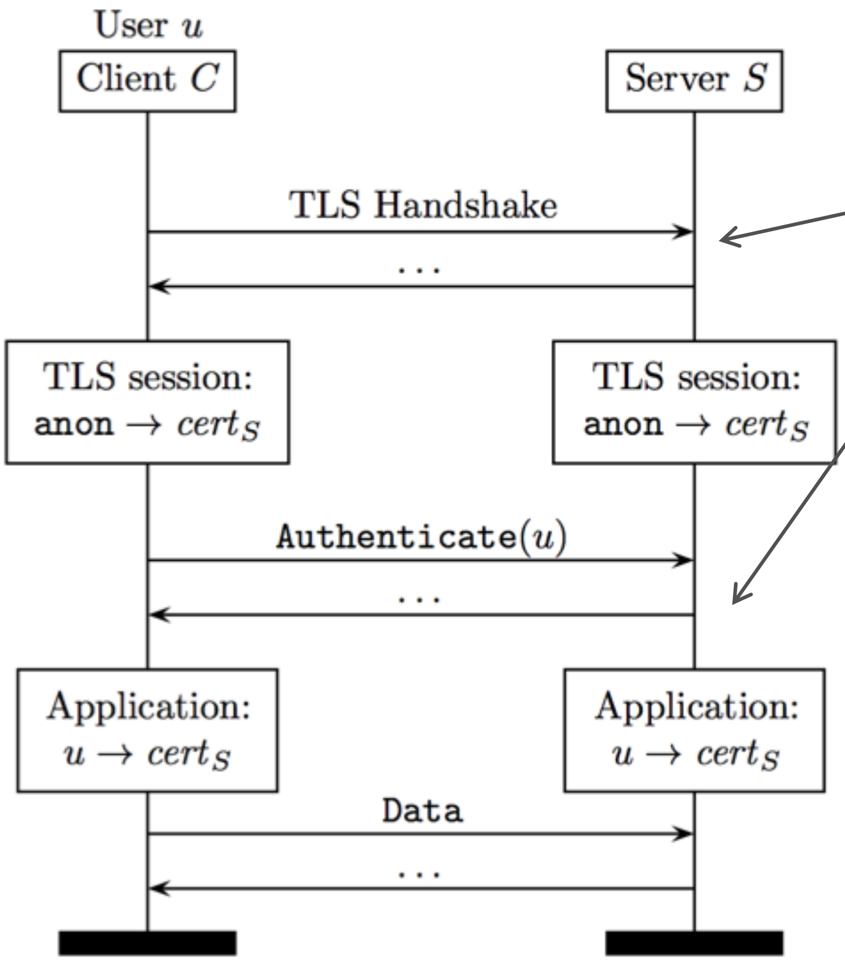
- Attacker can downgrade TLS 1.2 connection from SHA-256 to MD5, and then apply transcript collision attacks (SLOTH)

What went wrong?

- Crypto Weakness:
  Continued support for RSA-MD5 signatures
- Logical Protocol flaw:
  Downgrade attack on signature algorithms extension
- Implementation bug:
  OpenSSL, GnuTLS, NSS accept MD5 signatures even if disabled

Exploiting
Logical Flaws:
Triple Handshake Attacks

User authentication over TLS



Application-level Authentication

- *Outer*: server-authenticated TLS
- *Inner*: user authentication

Many examples of this pattern

- SASL, GSSAPI, EAP, …
- TLS Renegotiation with client certificate

Inner authentication *endorses* unauthenticated TLS channel

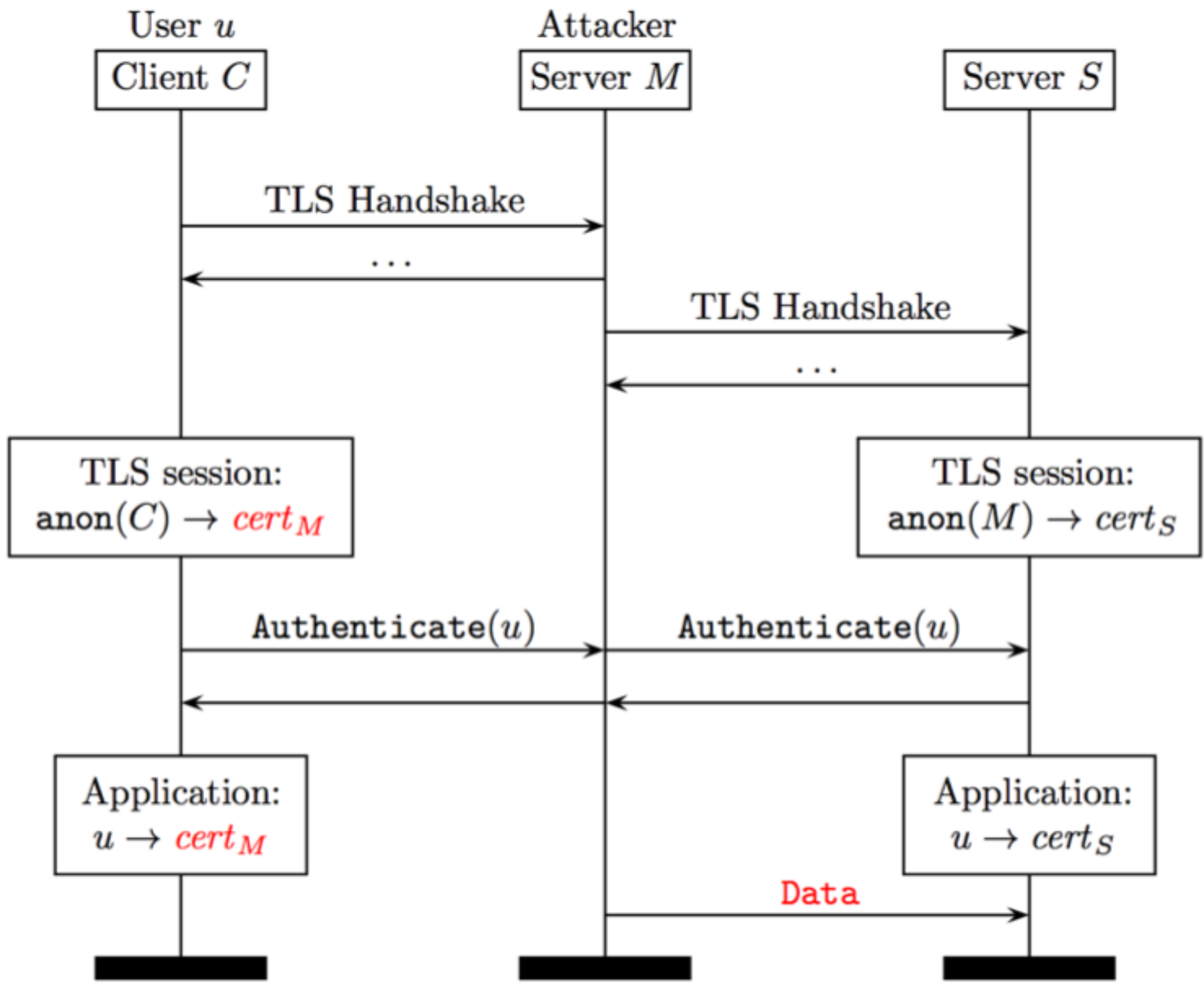
- *Need to strongly bind the two protocol layers together!*

Generic credential forwarding attack

Simplified version of [Asokan, Niemi, Nyberg'02]

- Suppose *u* uses same authentication credential at both *M* and *S*
- *M* forwards *S's* authentication challenge to *C*
- *M* forwards *C's* response to *S*

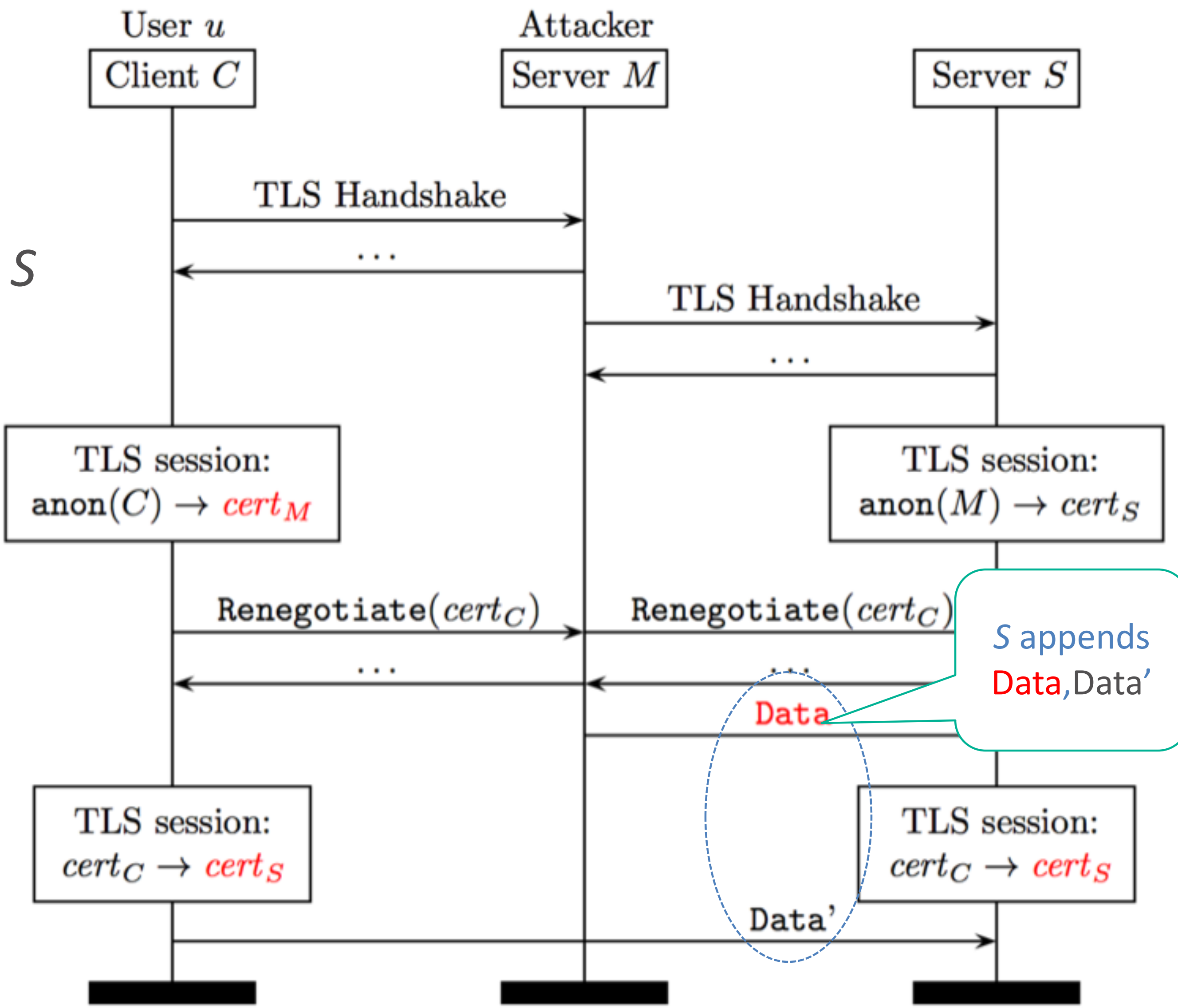
- *M can* log in as *u* at *S*!



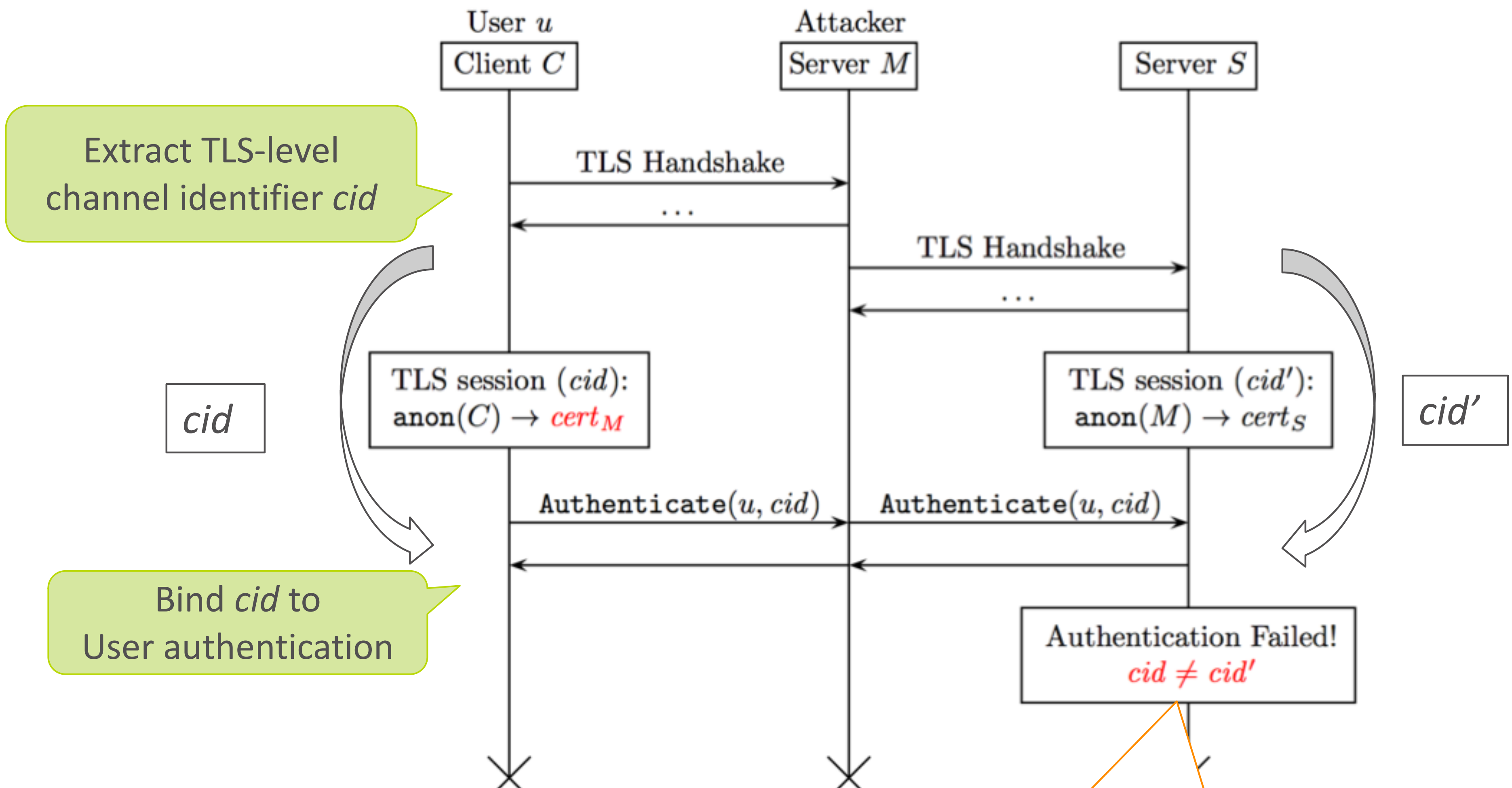
TLS renegotiation attack [2009]

Martin Rex's Version

- Suppose *u* uses same client cert to log in to both *M* and *S*
- *M* forwards *S's* renegotiation request to *C*
- *M* forwards renego handshake between *C and S*
- *S concatenates data sent by M to data sent by u!*



Binding user auth to TLS channels



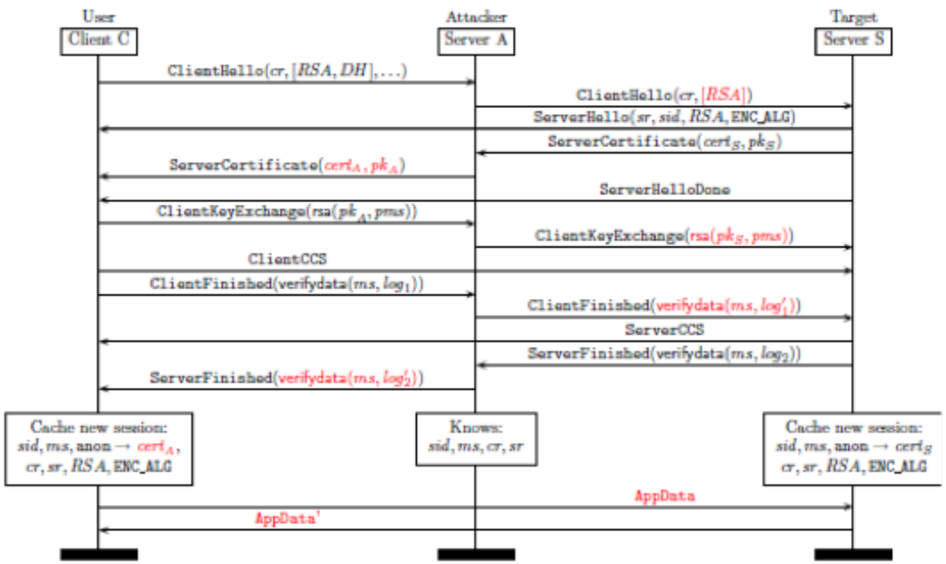
Computing a channel identifier (*cid*):

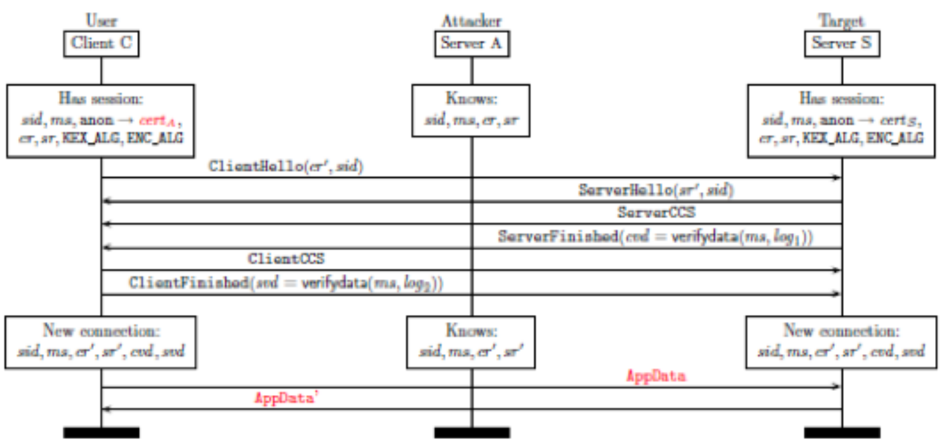
- *f(master secret)* (EAP)
- *f(handshake log)* (Renegotiation Indication,SASL)

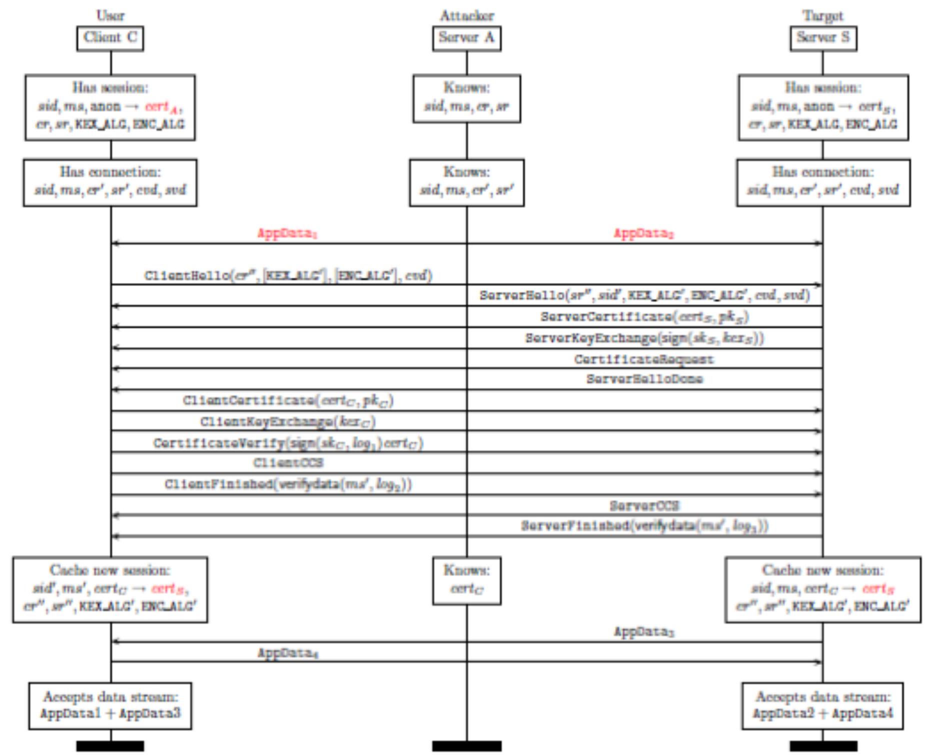
Does not work if *M* can ensure that cid = cid'

Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS

Karthikeyan Bhargavan*, Antoine Delignat-Lavaud*, Cédric Fournet†, Alfredo Pironti* and Pierre-Yves Strub‡
*INRIA Paris-Rocquencourt †Microsoft Research ‡IMDEA Software Institute







Details, demos at:

http://secure-resumption.com

Triple Handshake attack: step 1

Key Synchronization Attack

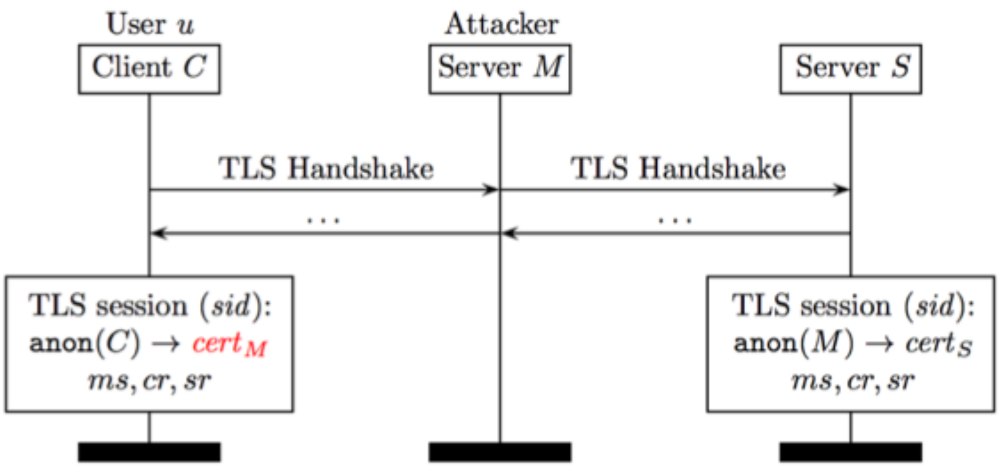
A malicious server *M* can ensure that the master secrets in two different connections from *C-M* and *M-S* are the same

RSA Key Synchronization

M re-encrypts *C's* premaster secret under *S's* public key
M forces same ciphersuite and nonces on the two handshakes

DHE Key Synchronization

M chooses a "bad" (non-prime) Diffie-Hellman group



Does not break single handshake theorems

"If a client completes with an honest server…"

Breaks EAP compound authentication (reenables 2002 attack)

The master secret is not a good channel identifier (it isn't *contributive*)

Renegotiation indication channel identifier (handshake log) still works.

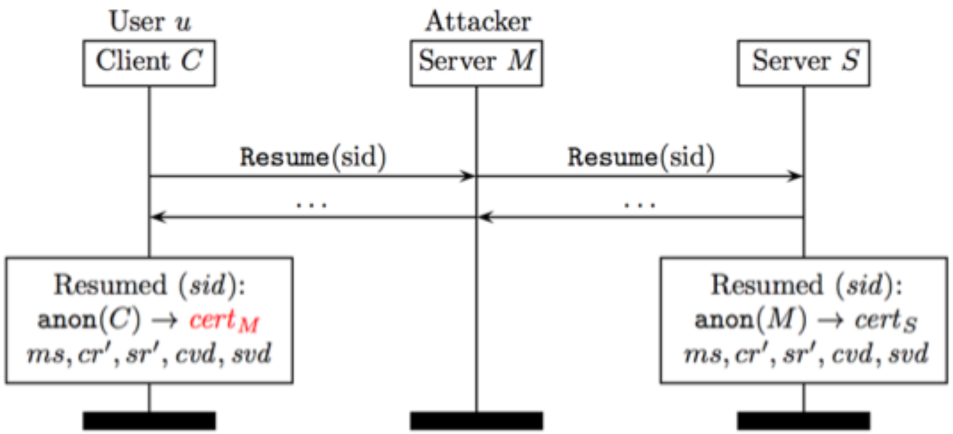
Triple Handshake attack: step 2

Transcript Synchronization Attack

After resumption, a malicious server *M* can ensure that the master secrets, keys, and handshake logs on two different connections from *C*-*M* and *M-S are the same*

Abbreviated agreement

Transcript depends only on master secret, ciphersuite, session ID (no certificates)



Does not break session resumption theorem

"If the server in the original handshake was honest…"

Breaks transcript-based channel identifiers

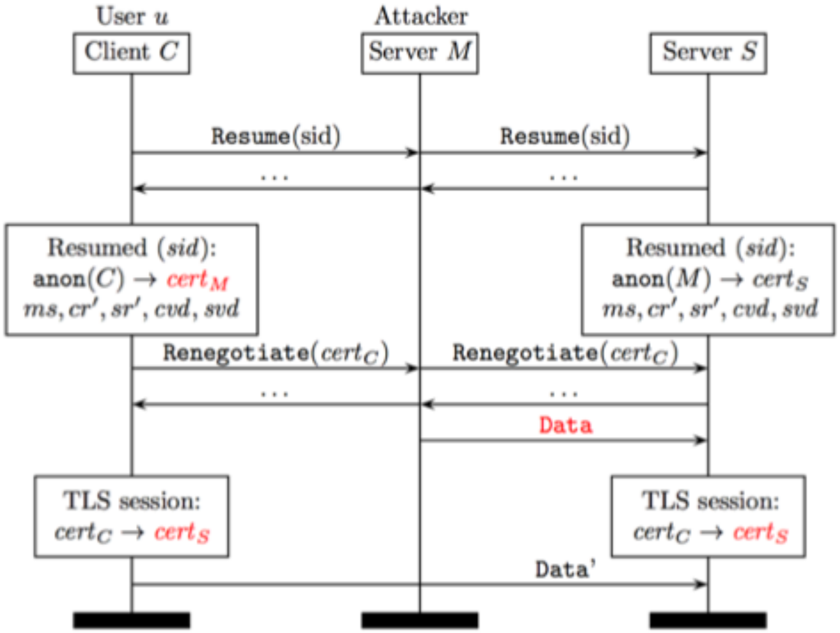
After resumption, handshake log is not a good channel identifier

Breaks tls-unique (SASL), renegotiation indication

Triple Handshake attack: step 3

User Impersonation Attack (reenables 2009 attack)

cid = hash(abbreviated handshake log) same on both connections
So *M* can forward renegotiation between *C* and *S* unchanged.



Surely this must break Giesen's multi-handshake theorem?

Renegotiation with honest peer implies agreement on abbreviated handshake, *but not on original handshake*
Theorem needs honest peer in original handshake for agreement on all three

Impact

A malicious website can impersonate any user who uses client certificates on any other website that requires client certificate auth, and supports resumption and renegotiation

What went wrong?

- Logical protocol flaw
  - Problem: Key synchronization attack on RSA/DHE
  - Countermeasure: Independent keys per connection

- Logical protocol flaw
  - Problem: Transcript synchronization after resumption
  - Countermeasure: Independent master secrets per session

Exploiting Implementation Bugs: State Machine Attacks

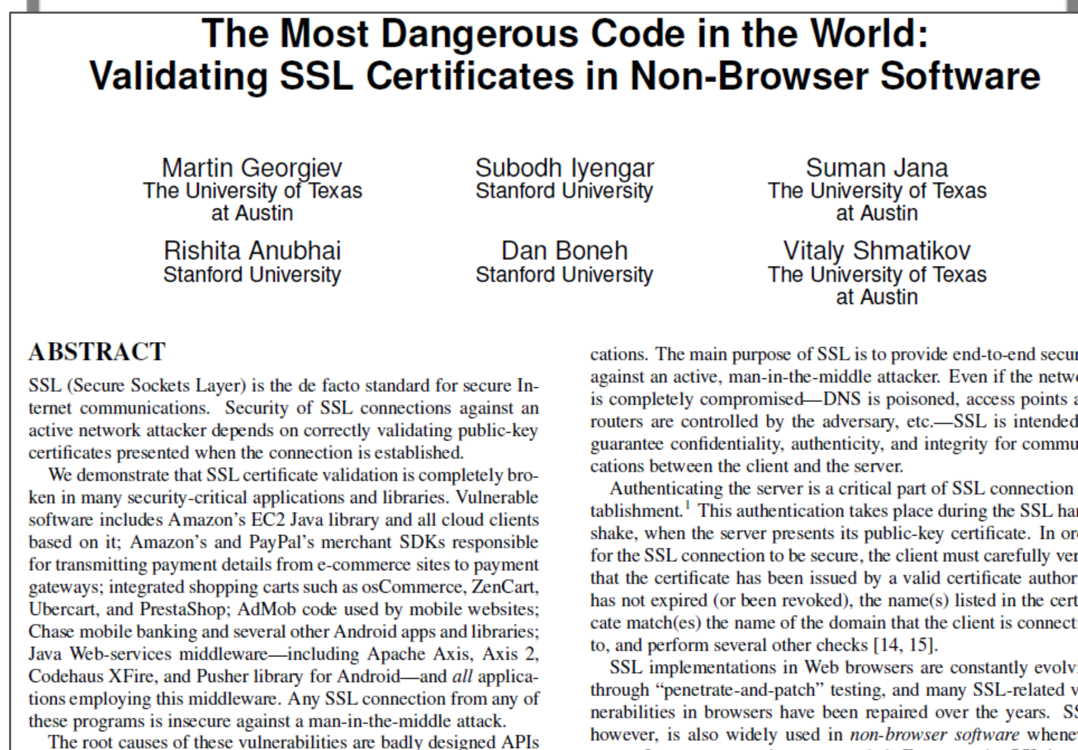
TLS Implementation Bugs

Memory safety
Buffer overruns leak secrets

Missing checks
Forgetting to verify signature/MAC/certificate bypasses crypto guarantees

Certificate validation
ASN.1 parsing, wildcard certificates

State machine attacks
Confusions between modes



The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software

Martin Georgiev, The University of Texas at Austin
Subodh Iyengar, Stanford University
Suman Jana, The University of Texas at Austin
Rishita Anubhai, Stanford University
Dan Boneh, Stanford University
Vitaly Shmatikov, The University of Texas at Austin

ABSTRACT

SSL (Secure Sockets Layer) is the de facto standard for secure Internet communications. Security of SSL connections against an active network attacker depends on correctly validating public-key certificates presented when the connection is established.

We demonstrate that SSL certificate validation is completely broken in many security-critical applications and libraries. Vulnerable software includes Amazon's EC2 Java library and all cloud clients based on it; Amazon's and PayPal's merchant SDKs responsible for transmitting payment details from e-commerce sites to payment gateways; integrated shopping carts such as osCommerce, ZenCart, Ubercart, and PrestaShop; AdMob code used by mobile websites; Chase mobile banking and several other Android apps and libraries; Java Web-services middleware—including Apache Axis, Axis 2, Codehaus XFire, and Pusher library for Android—and *all* applications employing this middleware. Any SSL connection from any of these programs is insecure against a man-in-the-middle attack.

The root causes of these vulnerabilities are badly designed APIs

cations. The main purpose of SSL is to provide end-to-end secu against an active, man-in-the-middle attacker. Even if the netw is completely compromised—DNS is poisoned, access points a routers are controlled by the adversary, etc.—SSL is intended guarantee confidentiality, authenticity, and integrity for commu cations between the client and the server.

Authenticating the server is a critical part of SSL connection tablishment.[1] This authentication takes place during the SSL ha shake, when the server presents its public-key certificate. In or for the SSL connection to be secure, the client must carefully ver that the certificate has been issued by a valid certificate author has not expired (or been revoked), the name(s) listed in the cert cate match(es) the name of the domain that the client is connect to, and perform several other checks [14, 15].

SSL implementations in Web browsers are constantly evolv through "penetrate-and-patch" testing, and many SSL-related v nerabilities in browsers have been repaired over the years. S however, is also widely used in *non-browser software* whene

Recall: the many modes of TLS

Protocol versions

- TLS 1.2, TLS 1.1, TLS 1.0, SSLv3, SSLv2

Key exchanges

- ECDHE, FFDHE, RSA, PSK, …

Authentication modes

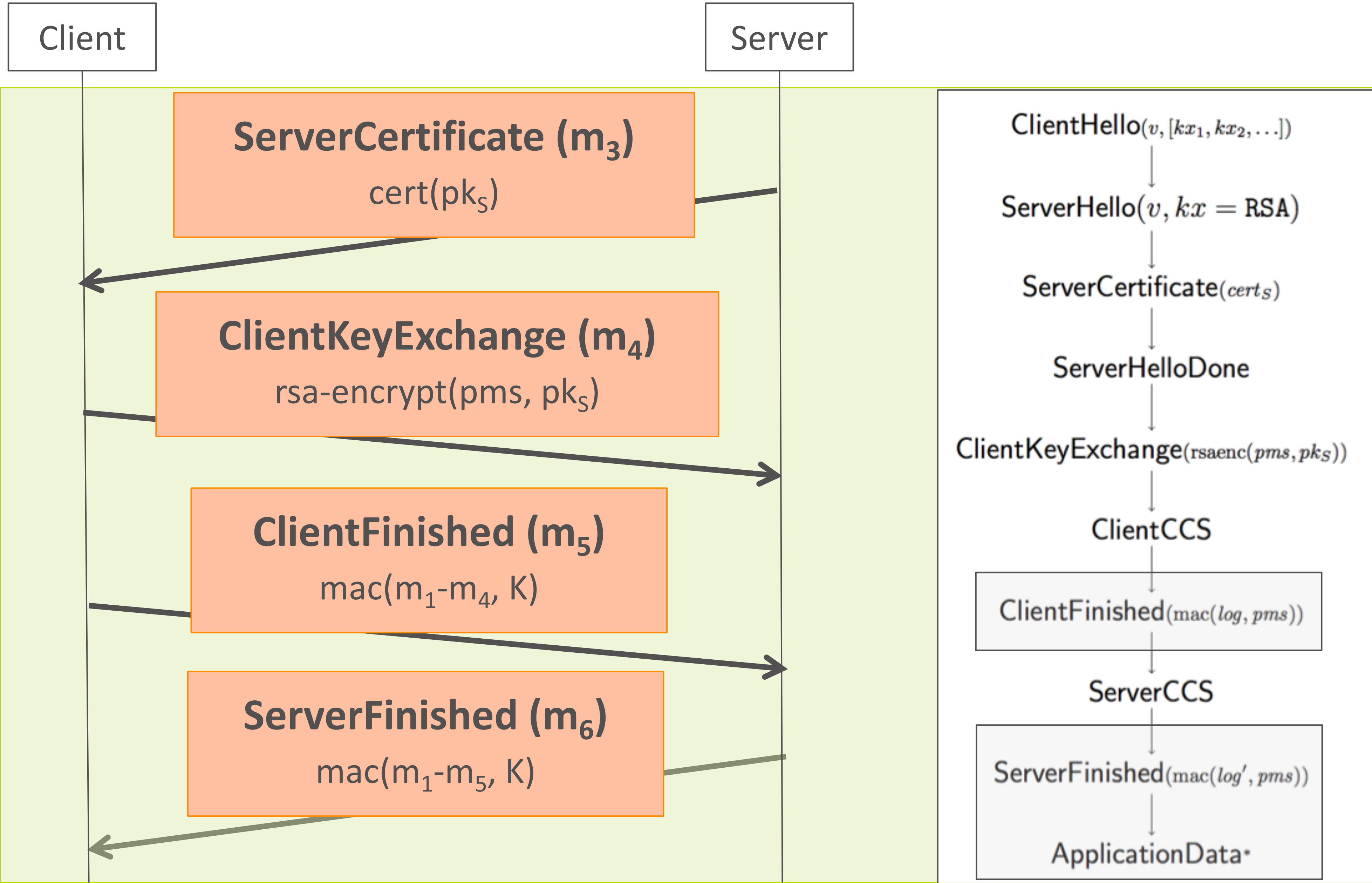
- ECDSA, RSA signatures, PSK,…

Authenticated Encryption Schemes

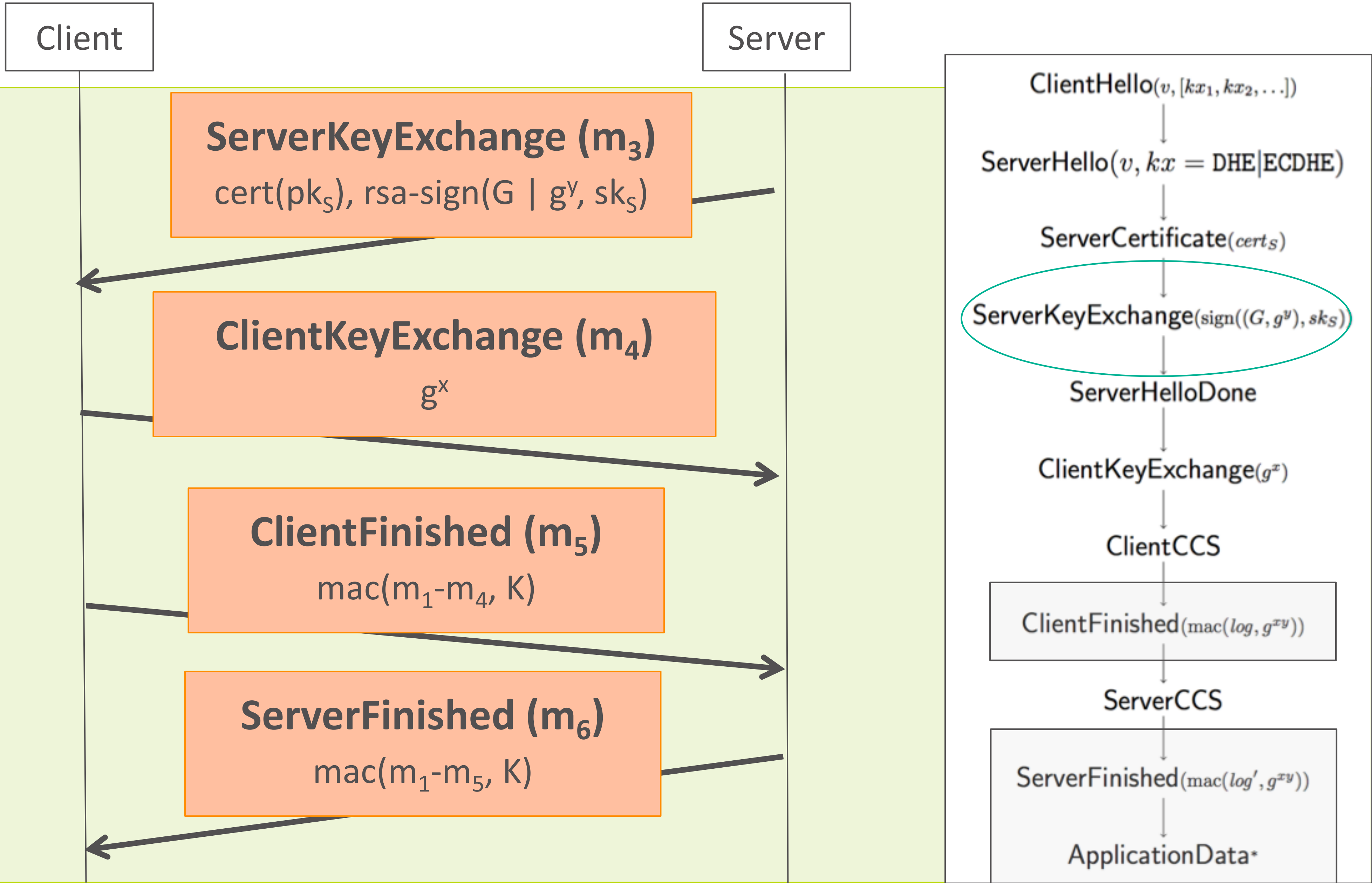
- AES-GCM, CBC MAC-Encode-Encrypt, RC4,…

100s of possible protocol combinations!

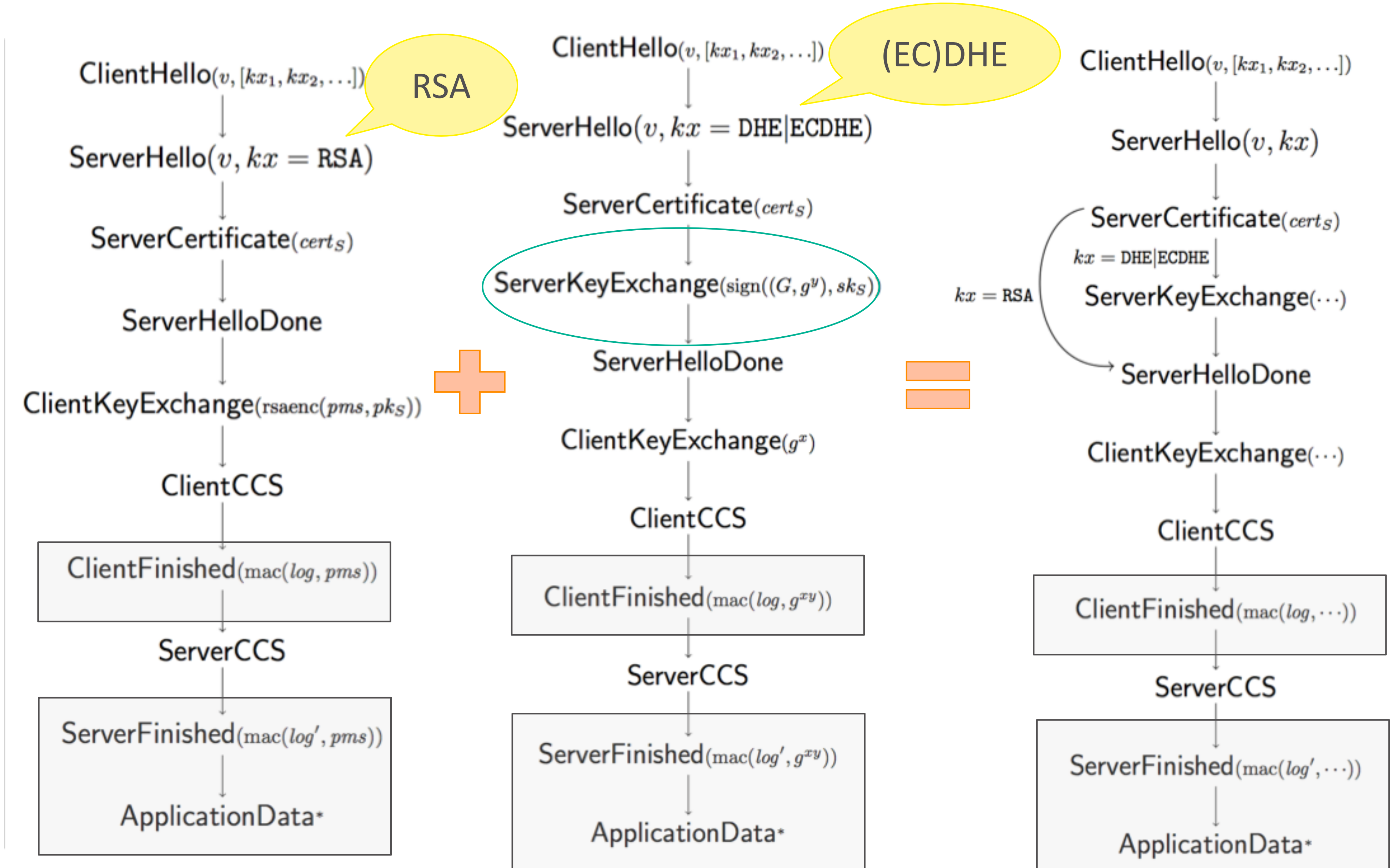
Implementing RSA Handshake



Implementing DHE Handshake



Composing Handshakes



TLS State Machine

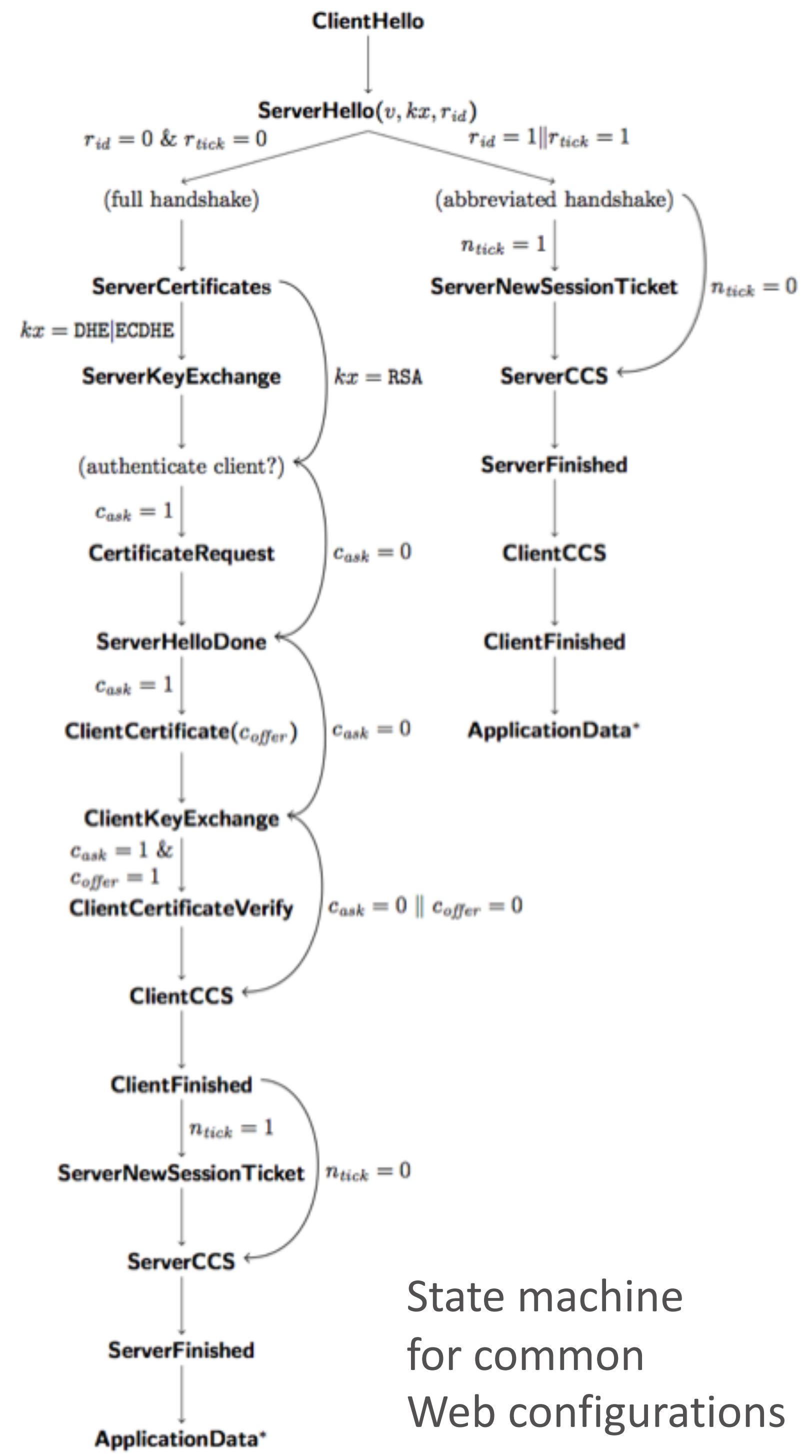
RSA + DHE + ECDHE

+ Session Resumption

+ Client Authentication

- Covers most features used on the Web
- Already quite a complex combination of protocols!

Do implementations conform to this state machine?

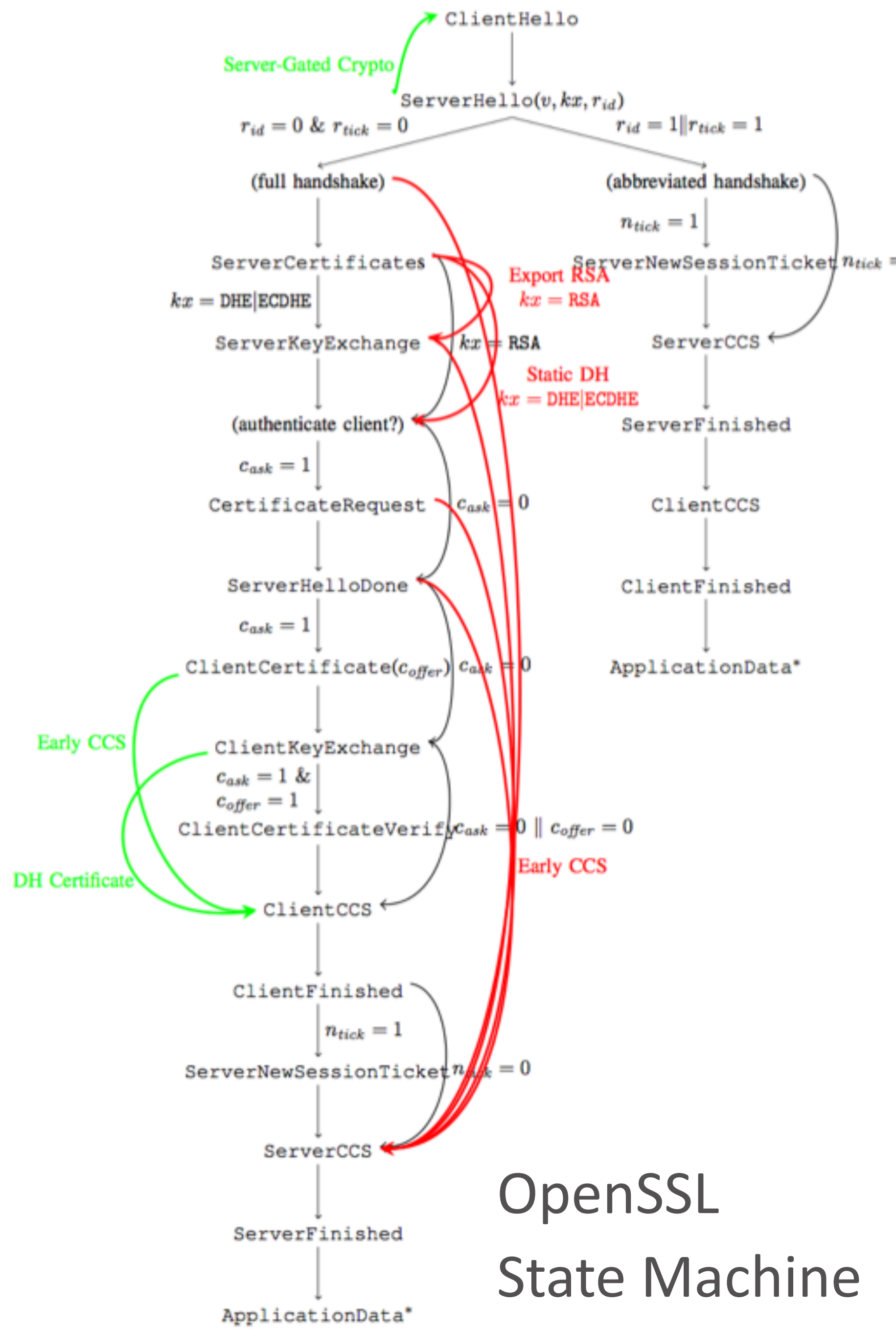


State machine for common Web configurations

Many, Many Bugs

Unexpected state transitions in OpenSSL, NSS, Java, …

- Required messages can be skipped
- Unexpected messages can be received



OpenSSL State Machine

Many, Many Bugs

Unexpected state transitions in OpenSSL, NSS, Java, …

- Required messages can be skipped
- Unexpected messages can be received

How come all these bugs?

- In independent code bases, sitting in there for years
- CVEs for many libraries
- Are they exploitable?

Java State Machine

Culprit: Underspecified State Machine

TLS specifies a ladder diagram with optional messages

- Relies on the Finished messages to ensure agreement

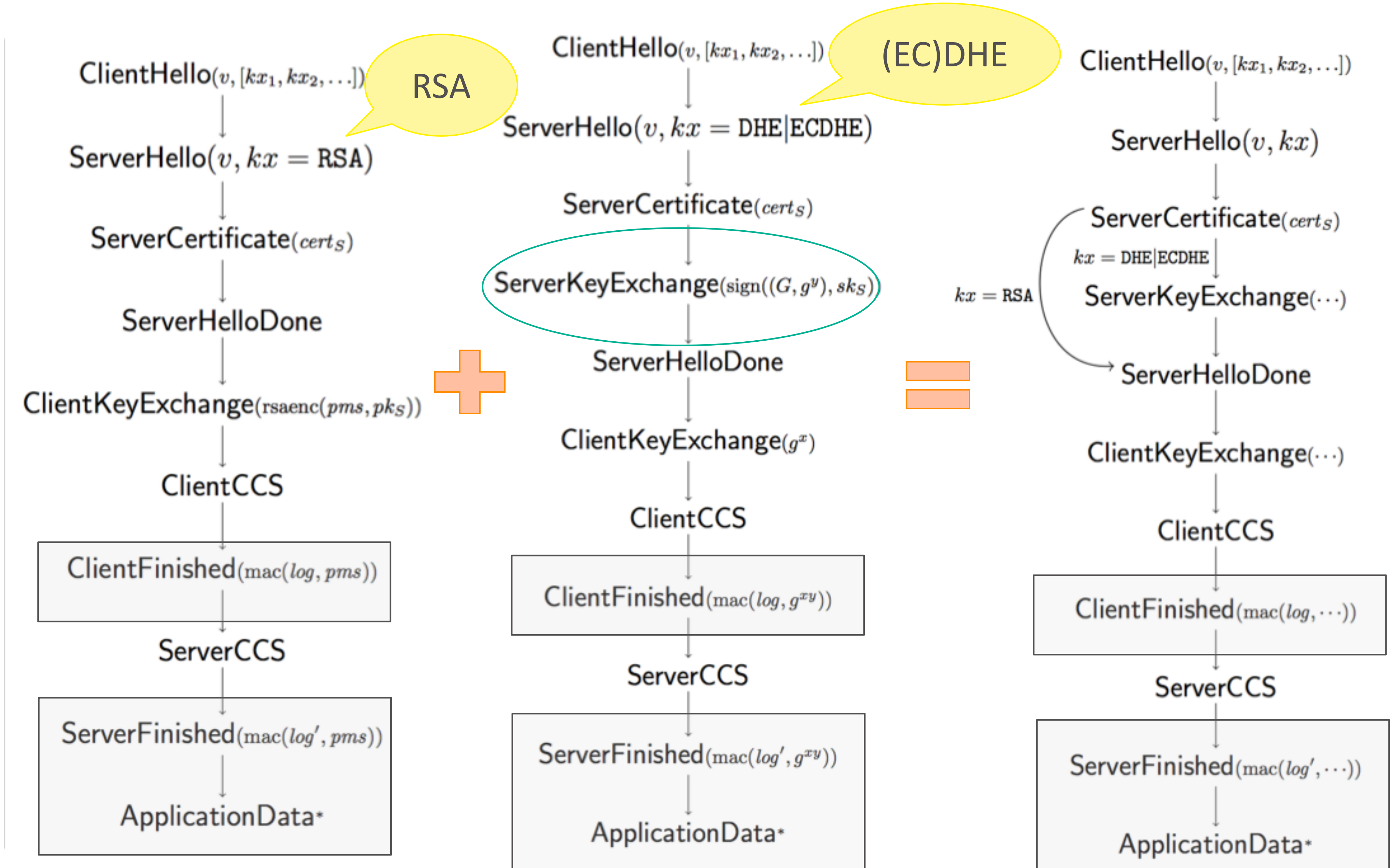
```
RFC 5246                          TLS                         August 2008


      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               ServerKeyExchange*
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                     -------->
                                               [ChangeCipherSpec]
                                   <--------             Finished
      Application Data             <------->     Application Data

             Figure 1.  Message flow for a full handshake
```

Composing Key Exchanges



Composing with Optional Messages

Treat ServerKeyExchange as optional

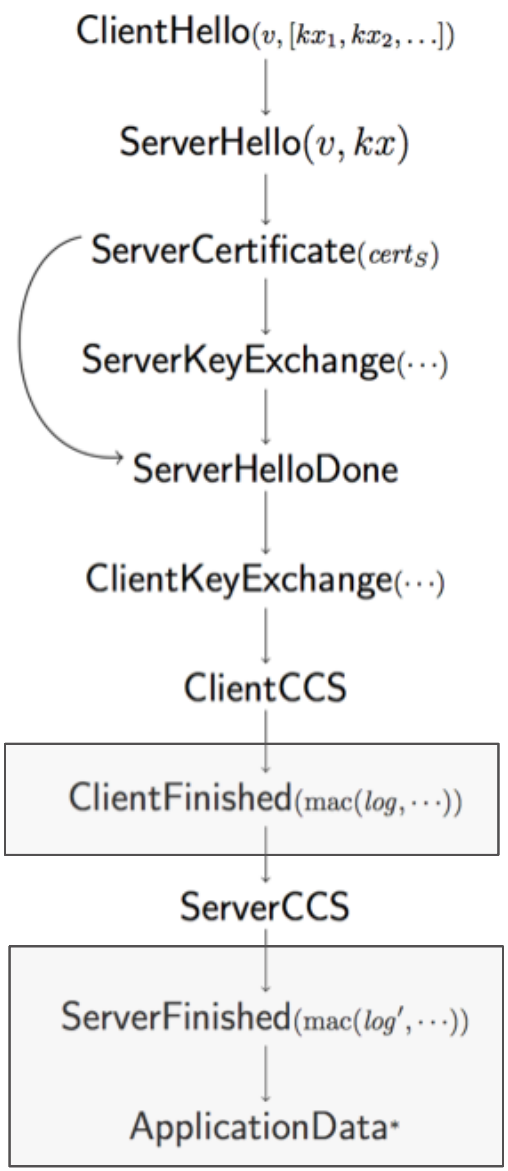
- Server decides to send it or not
- Client tries to handle both cases
- Consistent with Postel's principle for the Internet: *"be liberal in what you accept"* (not for security!)

Unexpected cases at the client

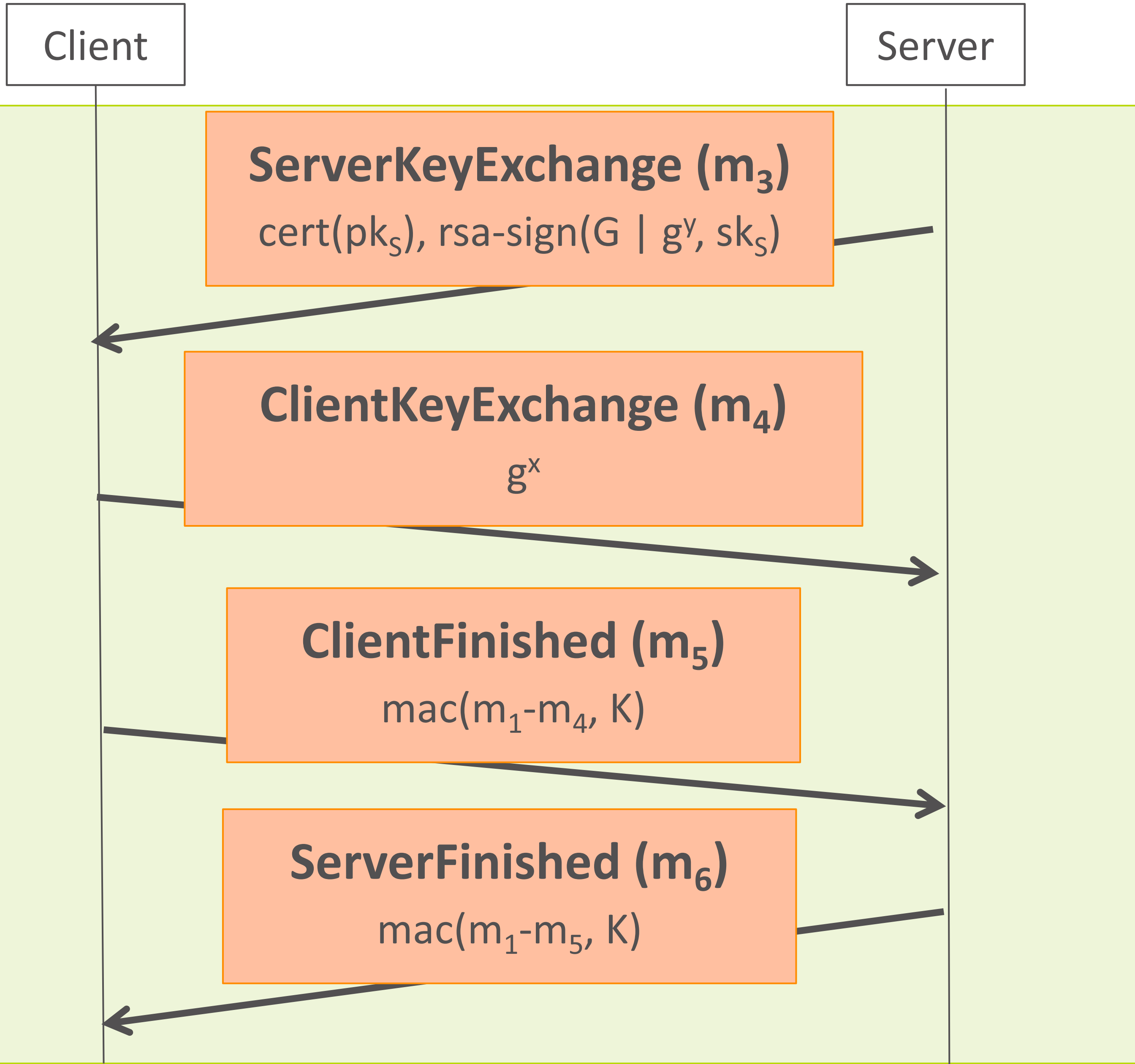
- Server skips ServerKeyExchange in DHE
- Server sends ServerKeyExchange in RSA

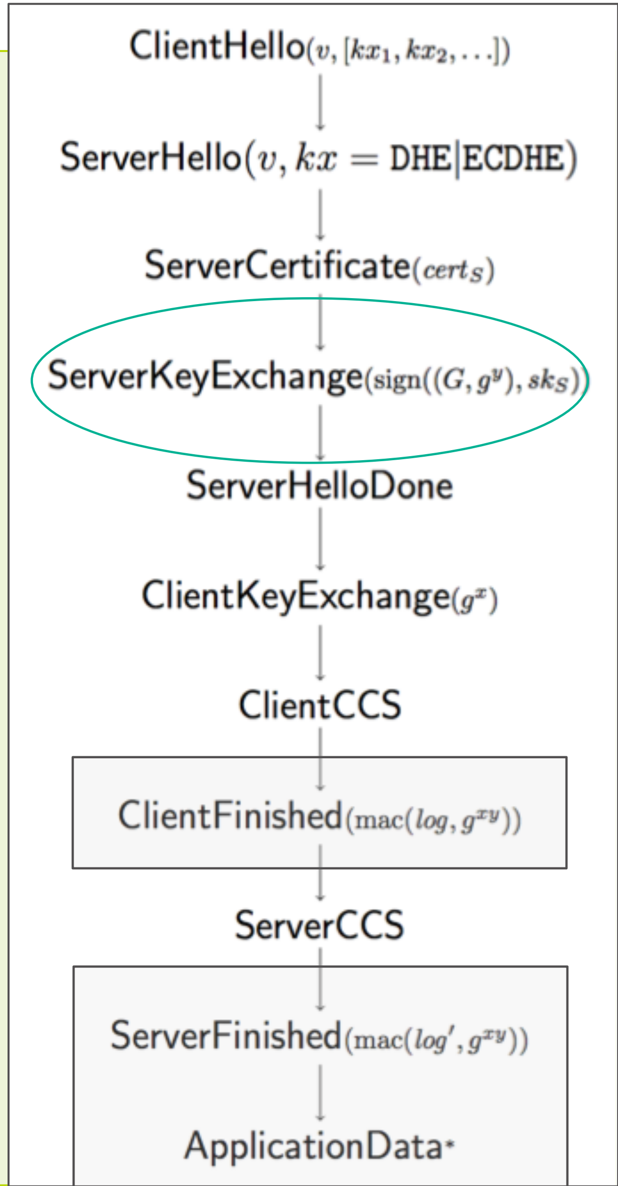
Clients should reject these cases

- But they don't, so we are not running the TLS handshake any more



Recall: DHE Handshake

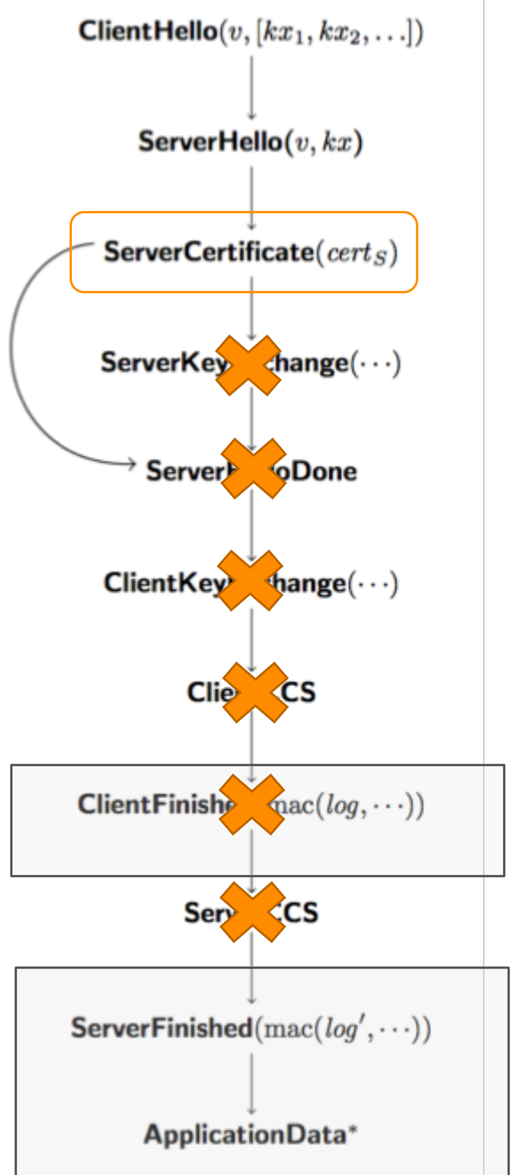




SKIPping Inconvenient Messages

Network attacker impersonates api.paypal.com to a JSSE client

1. Send PayPal's cert
2. SKIP ServerKeyExchange
   (bypass server signature)
3. SKIP ServerHelloDone
4. SKIP ServerCCS
   (bypass encryption)
5. Send ServerFinished
   using uninitialized MAC key
   (bypass handshake integrity)
6. Send ApplicationData
   (unencrypted) as S.com

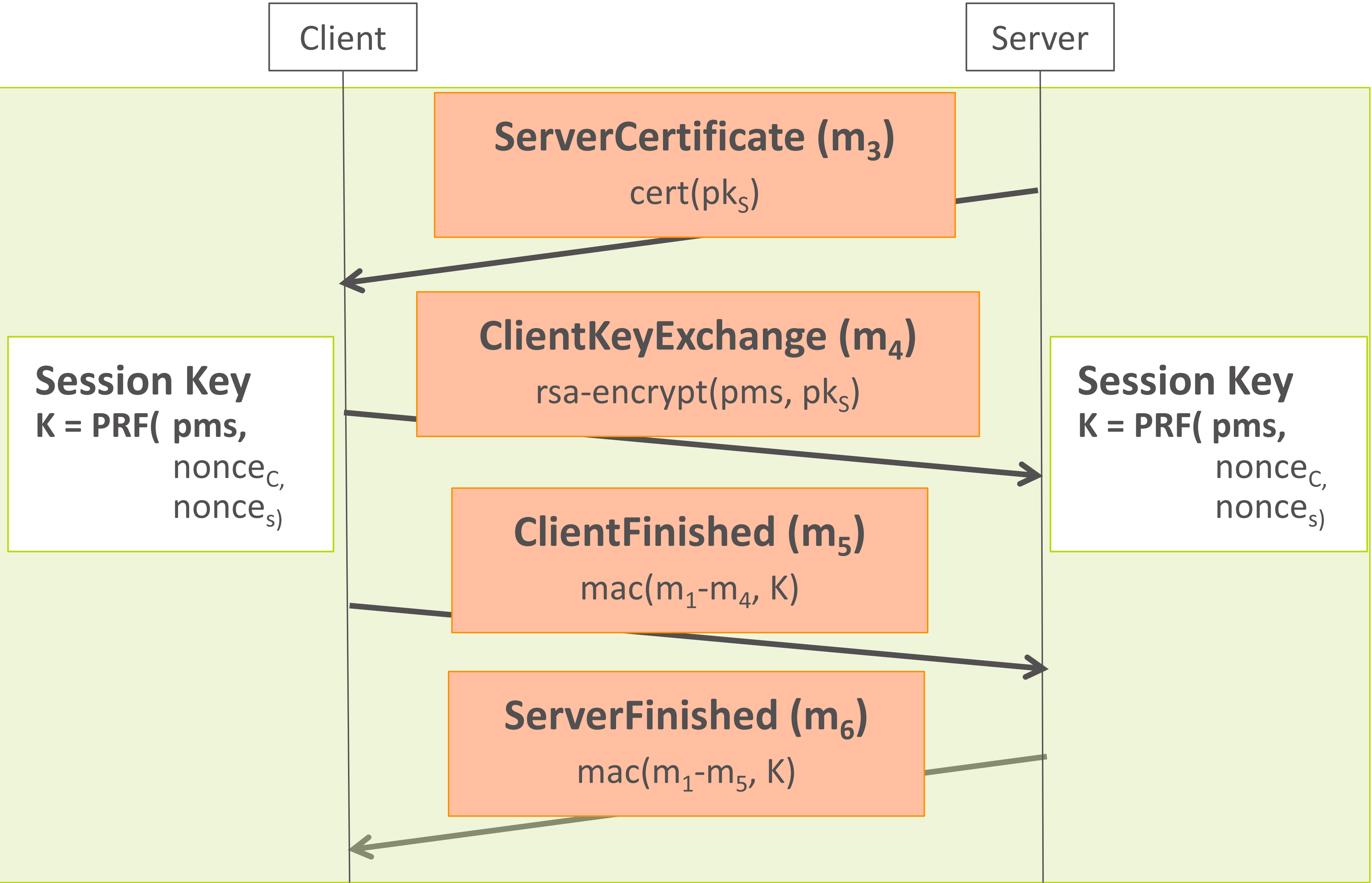


SKIP Impact

- A network attacker can impersonate *any* server (Paypal, Amazon, Google) to *any* Java TLS client (built with JSSE)
- Affects all versions of Java until Jan 2015 CPU (CVE-2014-6593)
- Other state machine bugs found in a dozen popular TLS libraries

Exploiting
Crypto Weaknesses +
Logical Flaws +
Implementation Bugs:
FREAK: Factoring RSA Keys

RSA Key Transport



RSA Key Transport

- Client chooses secret pms,
  adds maximum protocol version pv_{max},
  pads according to RSA PKCS#1 v1.5,
  and encrypts with server's public key pk_S
  $\text{rsa-pkcs1-encrypt}(pms, pk_S)$
  $= [pad \mid pv_{max} \mid pms]^e \bmod pq$

- Server decrypts, checks pad and protocol version,
  computes session key from pms

Security: In theory, relies on hardness of factoring pq

RSA Factoring Challenge

RSA Number	Decimal digits	Binary digits	Cash prize offered	Factored on	Factored by
RSA-100	100	330	US$1,000[4]	April 1, 1991[5]	Arjen K. Lenstra
RSA-110	110	364	US$4,429[4]	April 14, 1992[5]	Arjen K. Lenstra and M.S. Manasse
RSA-120	120	397	$5,898[4]	July 9, 1993[6]	T. Denny et al.
RSA-129 [**]	129	426	$100 USD	April 26, 1994[5]	Arjen K. Lenstra et al.
RSA-130	130	430	US$14,527[4]	April 10, 1996	Arjen K. Lenstra et al.
RSA-140	140	463	US$17,226	February 2, 1999	Herman te Riele et al.
RSA-150 [*] ?	150	496		April 16, 2004	Kazumaro Aoki et al.
RSA-155	155	512	$9,383[4]	August 22, 1999	Herman te Riele et al.
RSA-160	160	530		April 1, 2003	Jens Franke et al., University of Bonn
RSA-170 [*]	170	563		December 29, 2009	D. Bonenberger and M. Krone [***]
RSA-576	174	576	$10,000 USD	December 3, 2003	Jens Franke et al., University of Bonn
RSA-180 [*]	180	596		May 8, 2010	S. A. Danilov and I. A. Popovyan, Moscow State University[7]
RSA-190 [*]	190	629		November 8, 2010	A. Timofeev and I. A. Popovyan
RSA-640	193	640	$20,000 USD	November 2, 2005	Jens Franke et al., University of Bonn
RSA-200 [*] ?	200	663		May 9, 2005	Jens Franke et al., University of Bonn
RSA-210 [*]	210	696		September 26, 2013[8]	Ryan Propper
RSA-704 [*]	212	704	$30,000 USD	July 2, 2012	Shi Bai, Emmanuel Thomé and Paul Zimmermann
RSA-220	220	729		May 13, 2016	S. Bai, P. Gaudry, A. Kruppa, E. Thomé and P. Zimmermann

Best Generic Technique: Number Field Sieve (NFS)

- Try CADO-NFS: http://cado-nfs.gforge.inria.fr/

How long does factoring take with the number field sieve?

Answer 3

512-bit RSA: 7 months — large academic effort [Cavallar et al., 1999]

768-bit RSA: 2.5 years — large academic effort [Kleinjung et al., 2009]

512-bit RSA: 2.5 months — single machine [Moody, 2009]

512-bit RSA: 72 hours — single Amazon EC2 machine [Harris, 2012]

512-bit RSA: 7 hours — Amazon EC2 cluster [Heninger, 2015]

512-bit RSA: $<$ 4 hours — Amazon EC2 cluster

Factoring as a Service
Financial Crypto 2016
[Valenta et al. '16]

Factoring RSA keys in TLS

RSA encryption used in TLS 1.0-1.2

$rsa\text{-}pkcs1\text{-}encrypt(pms, pk_S)$
$= [pad \mid pv_{max} \mid pms]^e \bmod pq$

- If pq can be factored into p and q, an attacker can break TLS encryption, integrity
- 512-bit keys and 768-bit keys can be factored

Browsers now reject < 1024-bit RSA certs

- They will soon require >= 2048 bits
- So nobody still accepts 512-bit RSA keys, right?

Export-Grade Ciphers in TLS

In the 1990s, cryptography exports were controlled

- All software had two versions: domestic and export
- Export RSA keys, Diffie-Hellman groups limited to 512 bits
- Export symmetric crypto limited to 40 bit keys

International Traffic in Arms Regulations [April 1, 1992 version]

```
Category XIII--Auxiliary Military Equipment ...

(1) Cryptographic (including key management) systems, equipment, assemblies,
modules, integrated circuits, components or software with the capability of
maintaining secrecy or confidentiality of information or information
systems...
```

Commerce Control List [current]

```
a.1.b.1. Factorization of integers in excess of 512 bits (e.g., RSA);
```

Export-Grade Ciphers in TLS

TLS 1.0 included many Export-grade ciphers

- TLS_RSA_EXPORT_WITH_RC4_40_MD5
- TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
- TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
- TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

To support these, every TLS server had two sets of keys

- A 2048-bit RSA key for TLS_RSA +
  a 512-bit RSA key for TLS_RSA_EXPORT
- A 1025-bit DH group for TLS_DHE +
  a 512-bit DH group for TLS_DHE_EXPORT
- E.g. OpenSSL created a 512-bit RSA_EXPORT on startup

RSA_EXPORT support on the Web

In 2000, EXPORT deprecated in TLS 1.1, not used since

- (Dead) code still exists in OpenSSL and other libraries

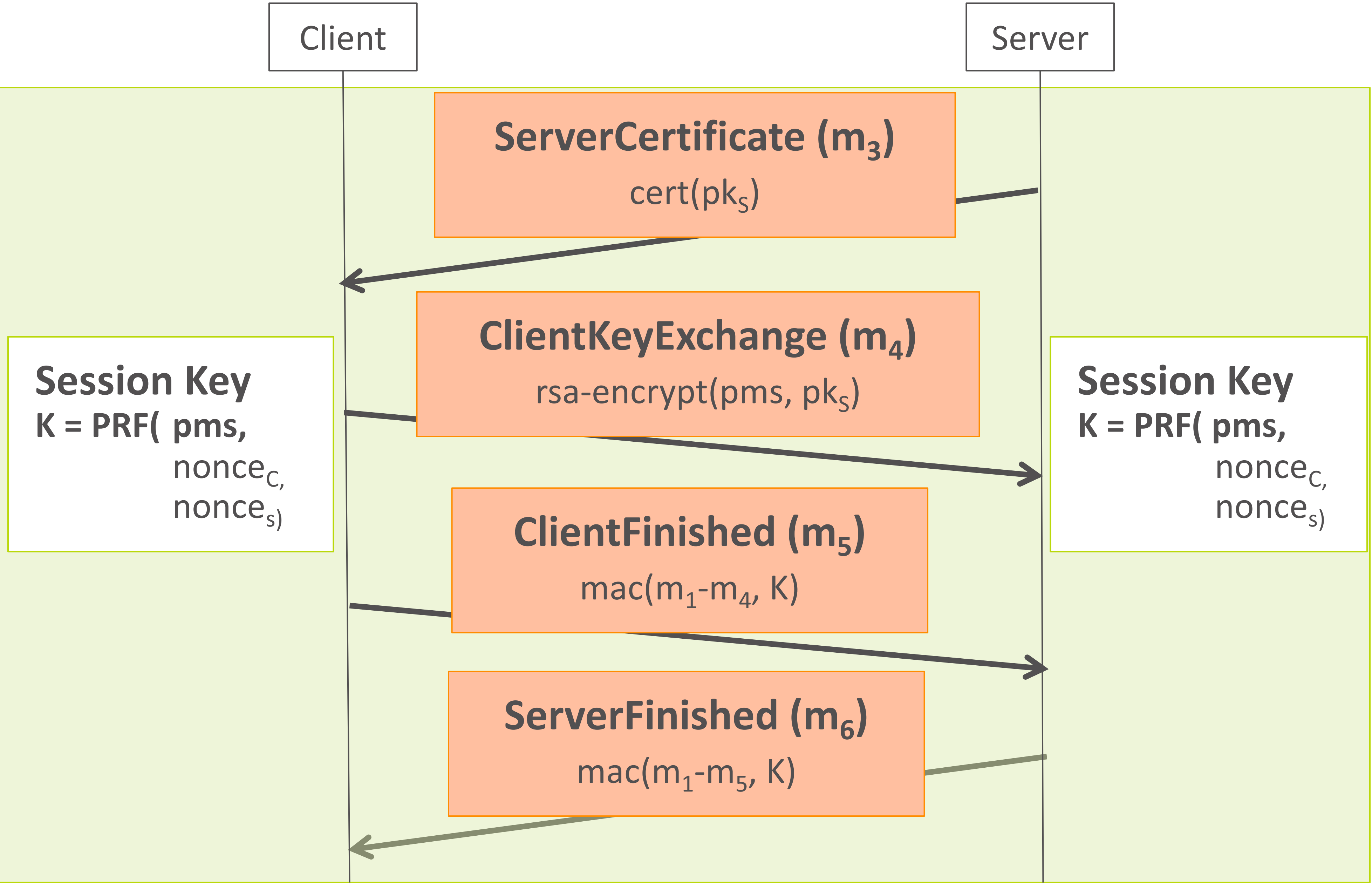
In Mar 2015, many TLS servers still allow RSA_EXPORT!

- 8.9M (26.3%) HTTPS servers support EXPORT ciphers
- 36.7% of HTTPS servers with browser-trusted certificates
- 9.6% of Alexa top 1M HTTPS servers
- Reason: backwards compatibility with old TLS clients

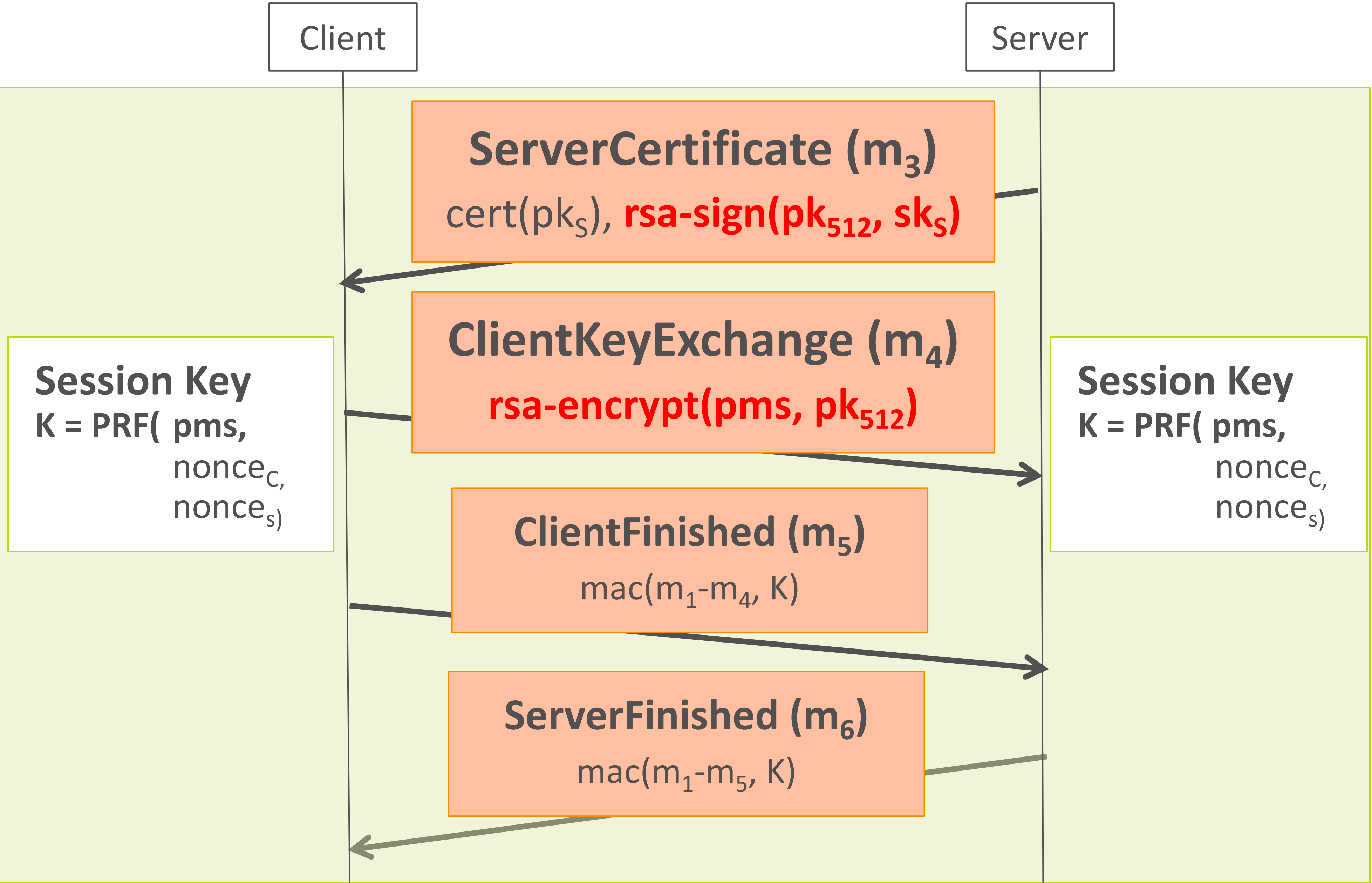
Modern browsers do not support or offer RSA_EXPORT

- EXPORT ciphers are never negotiated, so problem solved?
- An implementation bug reenables RSA_EXPORT in clients!

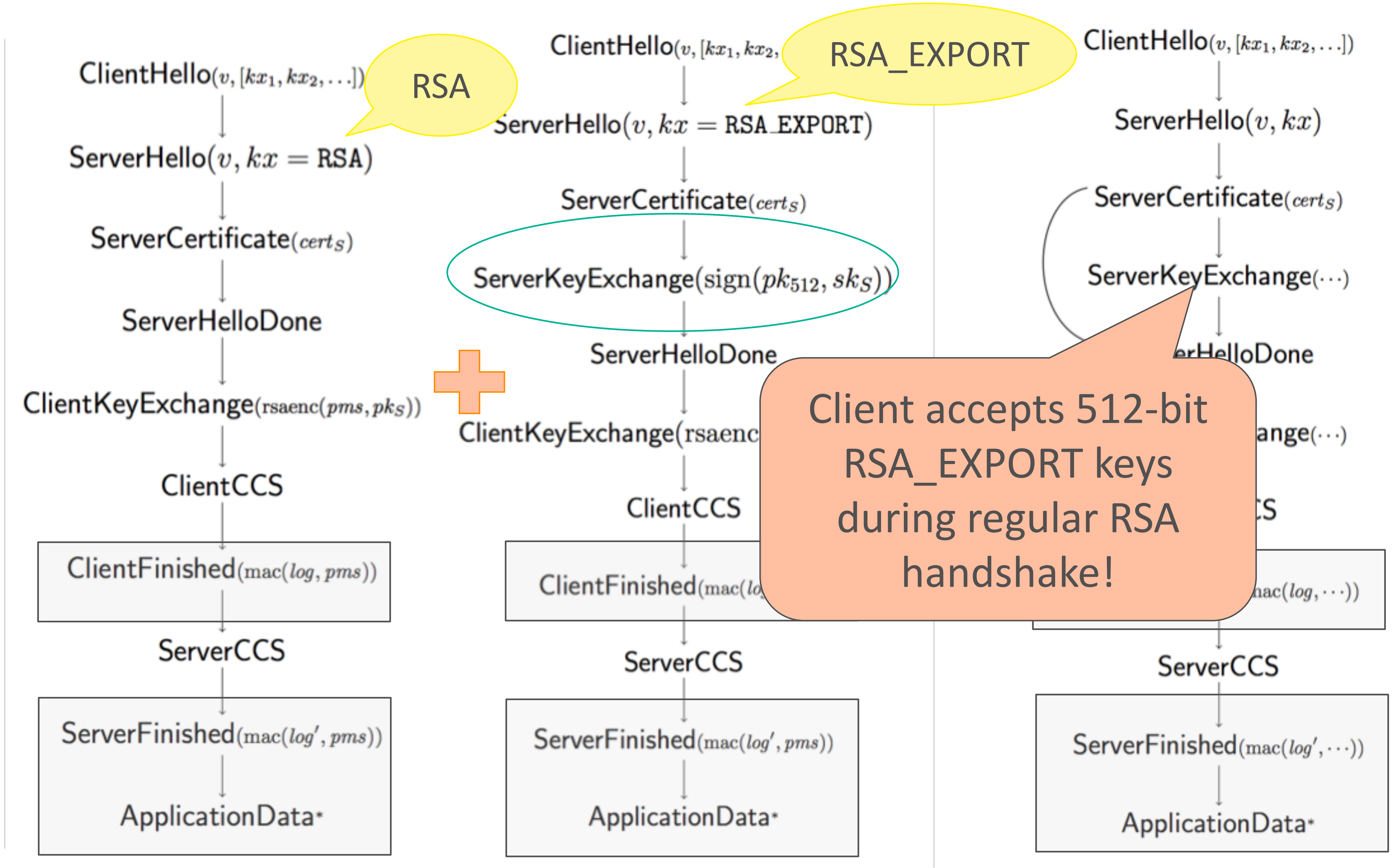
RSA Key Transport



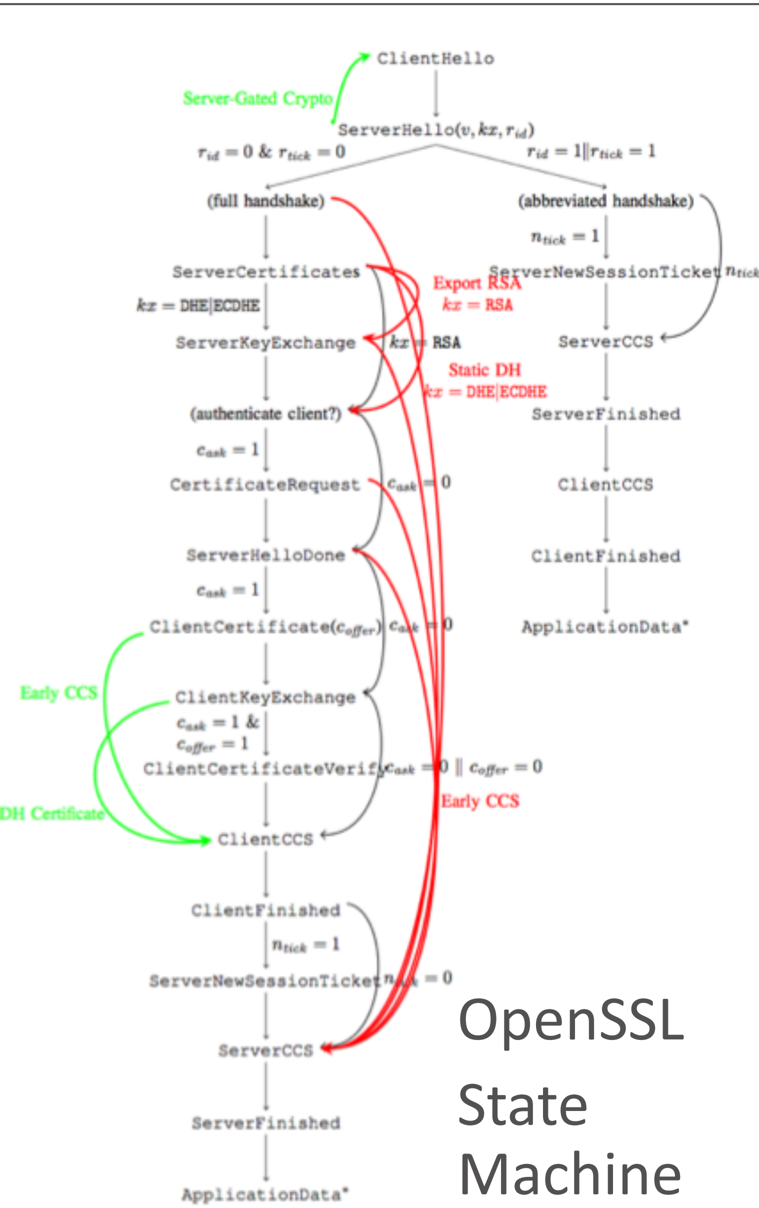
RSA_EXPORT Key Transport

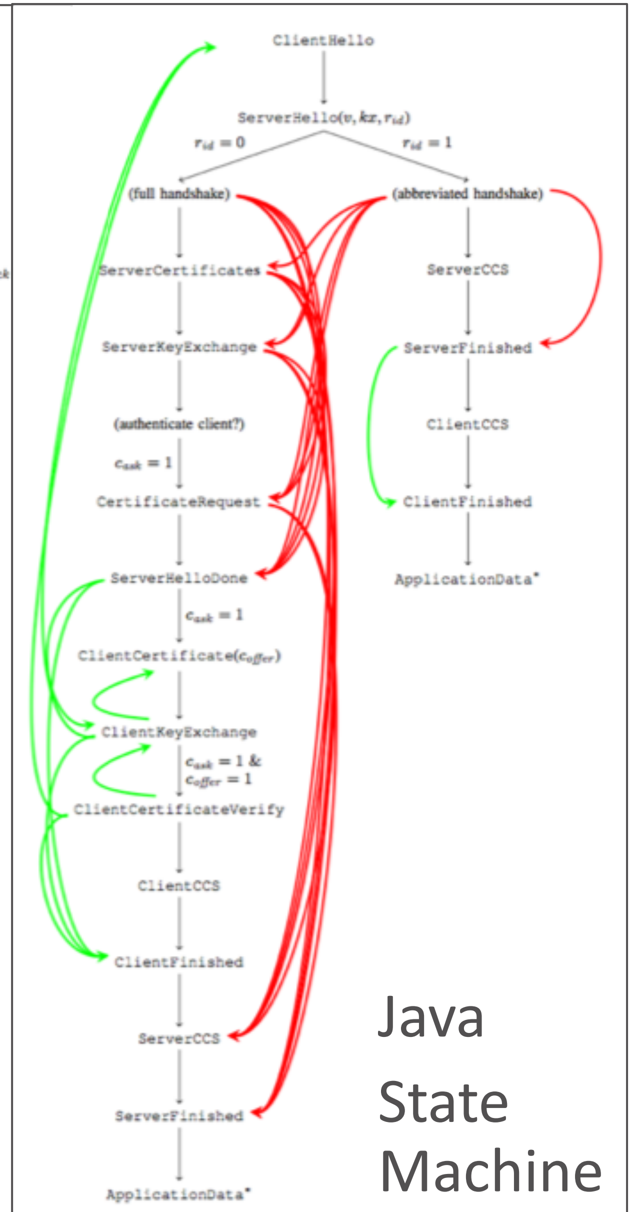


Badly Composing RSA + RSA_EXPORT



RSA_EXPORT State Machine Bugs in TLS





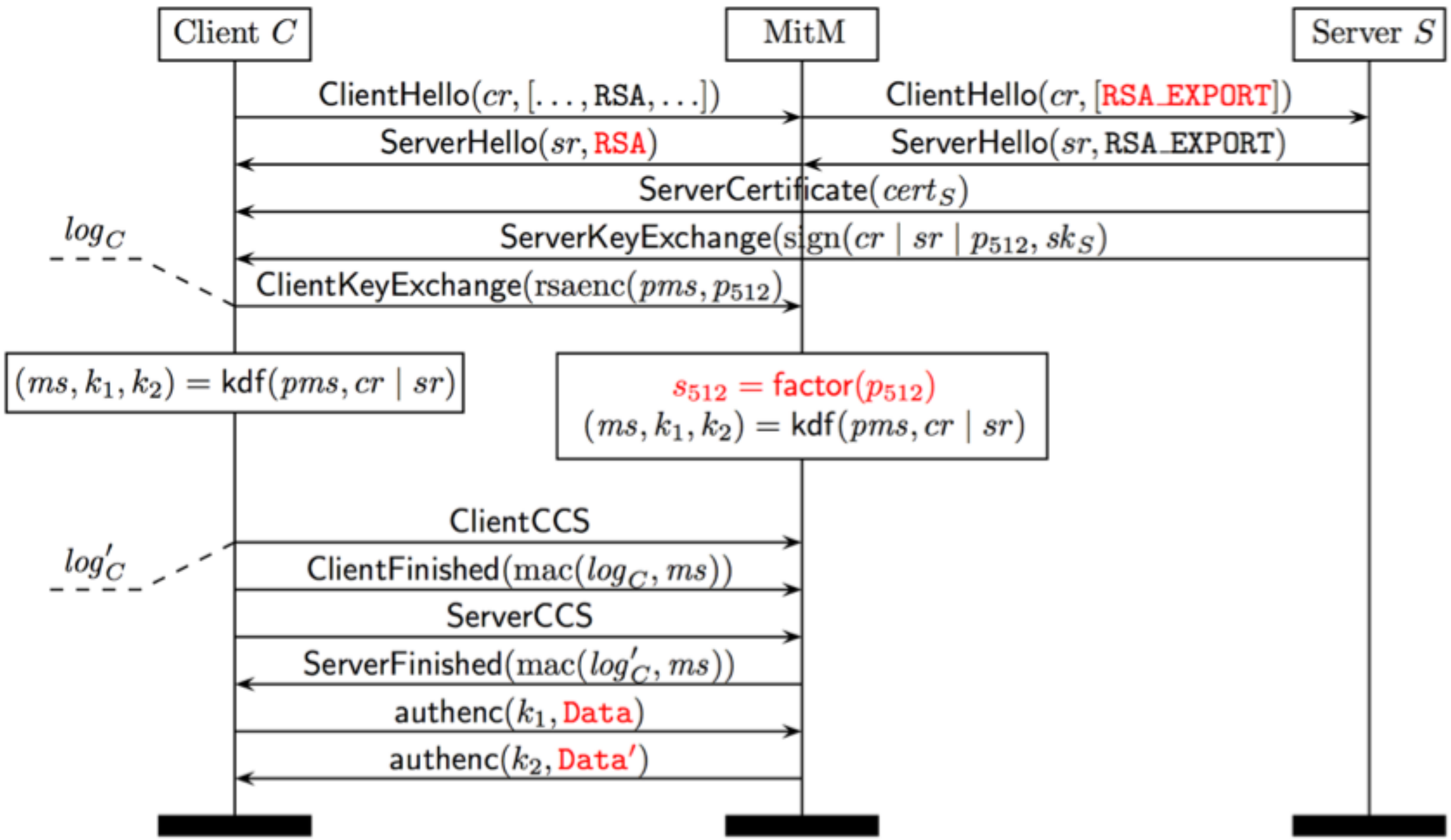
Affected Software

- OpenSSL, used by: Chrome, Opera, BlackBerry
- Schannel: Microsoft .NET, IE
- SecureTransport: Safari, iOS
- Oracle Java JSSE
  IBM Java JSSE
  Mono TLS

FREAK: Downgrade to RSA_EXPORT

A man-in-the-middle attacker can:

- impersonate servers that support RSA_EXPORT,
- at buggy clients that accept RSA_EXPORT keys in RSA handshakes



What went wrong?

- Cryptographic weakness
  - Problem: Continued support for RSA_EXPORT
  - Countermeasure: Disable EXPORT ciphersuites

- Logical protocol flaw
  - Problem: Signature ambiguity between RSA/RSA_EXPORT
  - Countermeasure: Signatures should cover transcript

- Implementation bug
  - Problem: Clients accept EXPORT even if disables
  - Countermeasure: Fix state machine composition

Part I: Summary

Real-world attacks exploit a combination of:

- Cryptographic weaknesses
- Logical protocol flaws
- Implementation bugs

Vulnerabilities in less-studied modes can break strong provably secure modes of the protocol

- Too many modes and corner cases to prove by hand

A need for automated protocol verification

- Tools for finding protocol flaws and implementation bugs
- Machine-checked proofs for real-world protocols

End of Part I