Protecting TLS 1.3 from Legacy Vulnerabilities

from theoretical security to verified deployments

Karthik Bhargavan

Inria

http://prosecco.inria.fr

+

many co-authors at INRIA, Microsoft Research, …

The Three Lives of TLS 1.3

PROTOCOL

STANDARD

DEPLOYMENT

Abstract Protocol Model

- Cryptographic proofs
- Symbolic analyses

Published Protocol Standard

- Concrete message formats
- Many, many ciphersuites
- Interoperability hacks

Deployed Protocol Code

- Configuration & negotiation
- Protocol state machine
- Crypto library
- Error handling

Verifying the TLS 1.3 Standard

`draft-5 (2015)`	*Crypto Proofs* [Dowling+, …]
`draft-10 (2016)`	*Crypto Proofs* [Krawczyk+, Li+, …], *Symbolic Analysis* [Cremers+]
`draft-20 (2017)`	*Crypto Proofs* [Bhargavan+,…], *Symbolic Analysis* [Cremers+, Bhargavan+]
`rfc8846 (2018)`	*Ready for deployment?*

Verifying TLS 1.3 Deployments

What goes wrong in TLS Deployments?

- **Incorrect Configuration:** Lingering Legacy Crypto [e.g. RC4, PKCS#1v1.5]
- **Insecure Composition:** Bad interactions between different versions/modes
- **Buggy Implementations:** State machine flaws, Side-channel attacks
- Often, a combination of all of the above is exploited in a ***downgrade attack***
  [e.g. POODLE, LOGJAM, FREAK, SLOTH, DROWN]

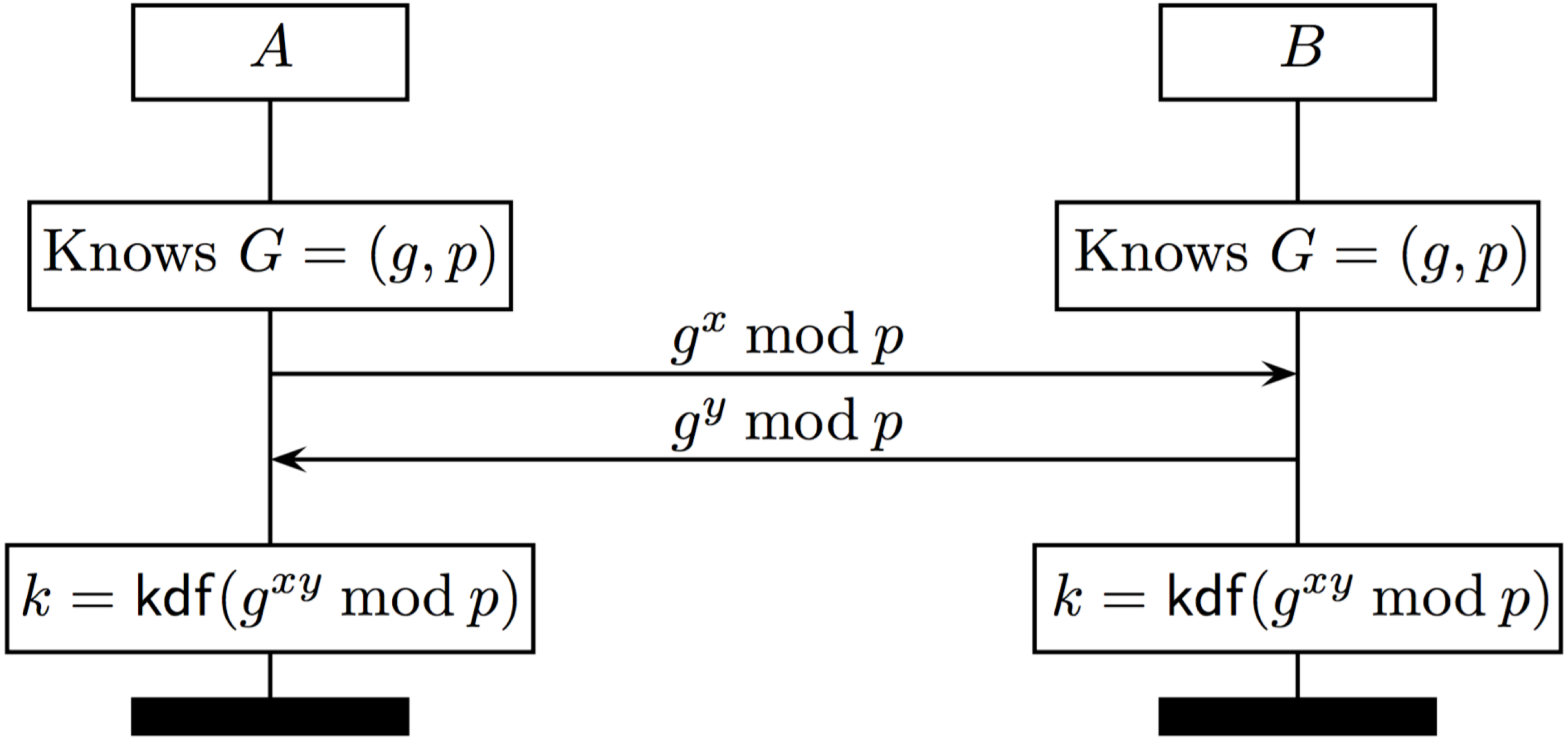
Verifying TLS 1.3 Deployments

We need to verify that TLS 1.3 deployments preserve our theorems

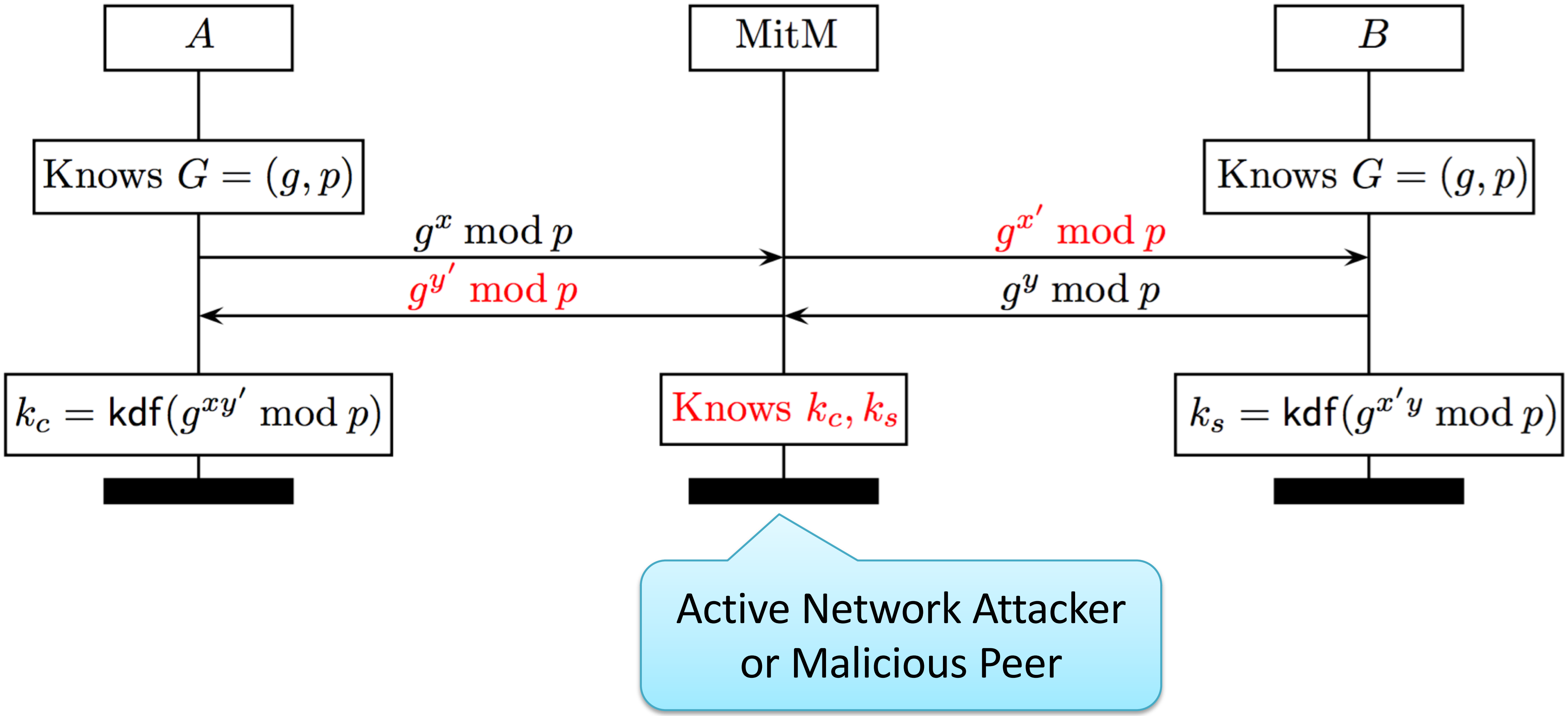
- **Downgrade Resilience for TLS 1.3**
  [Bhargavan, Brzuska, Fournet, Green, Kohlweiss, Zanella-Béguelin, S&P'16]
- **Symbolic Analysis of full TLS 1.3 composed with TLS 1.2**
  [Bhargavan, Blanchet, Kobeissi, S&P'17]
- **Verified implementations of TLS 1.3 and TLS 1.2**
  [Project Everest: Delignat-Lavaud et al., S&P'17]

Downgrade Attacks on
Agile Authenticated Key Exchange

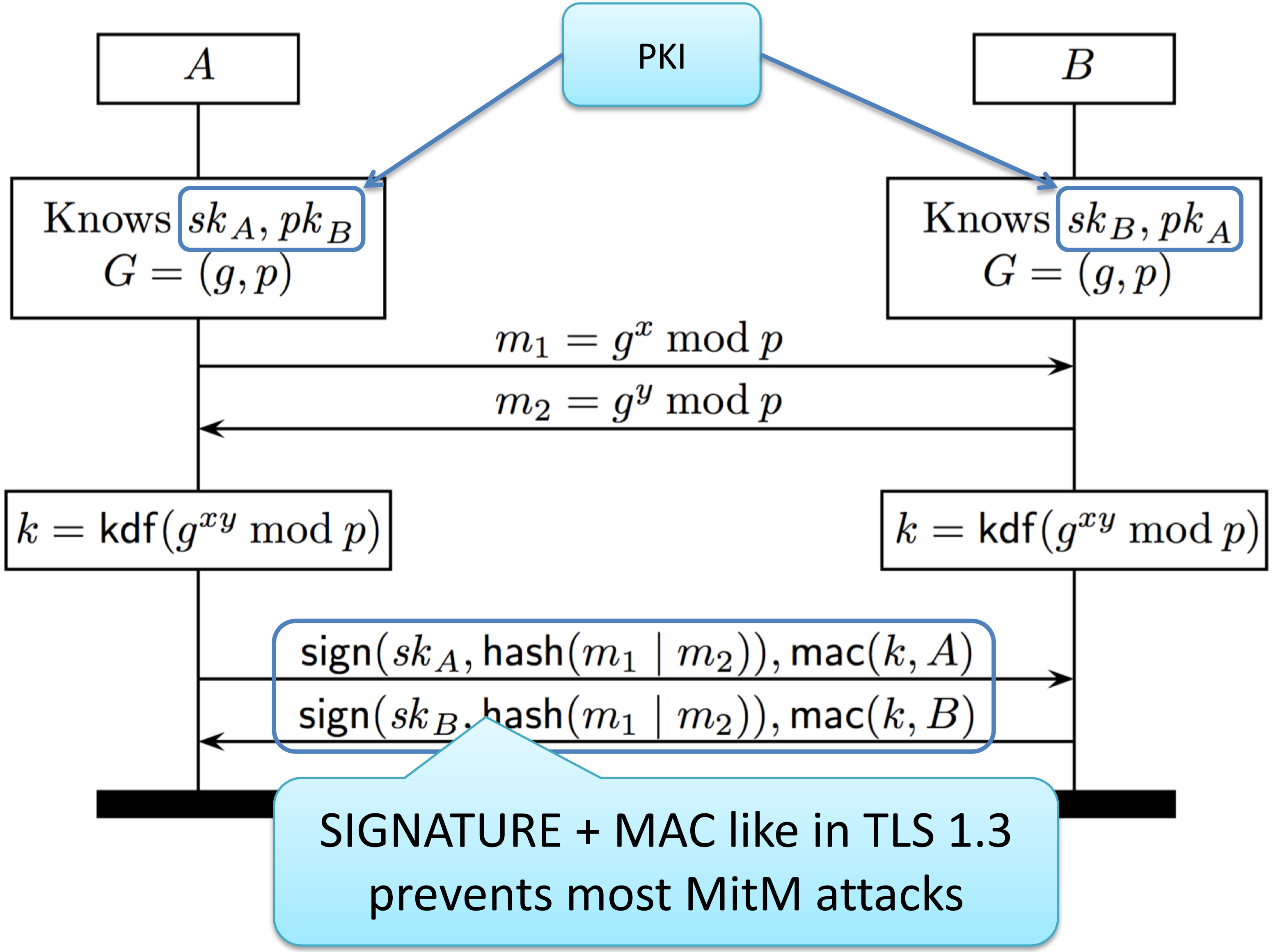
Diffie-Hellman key exchange



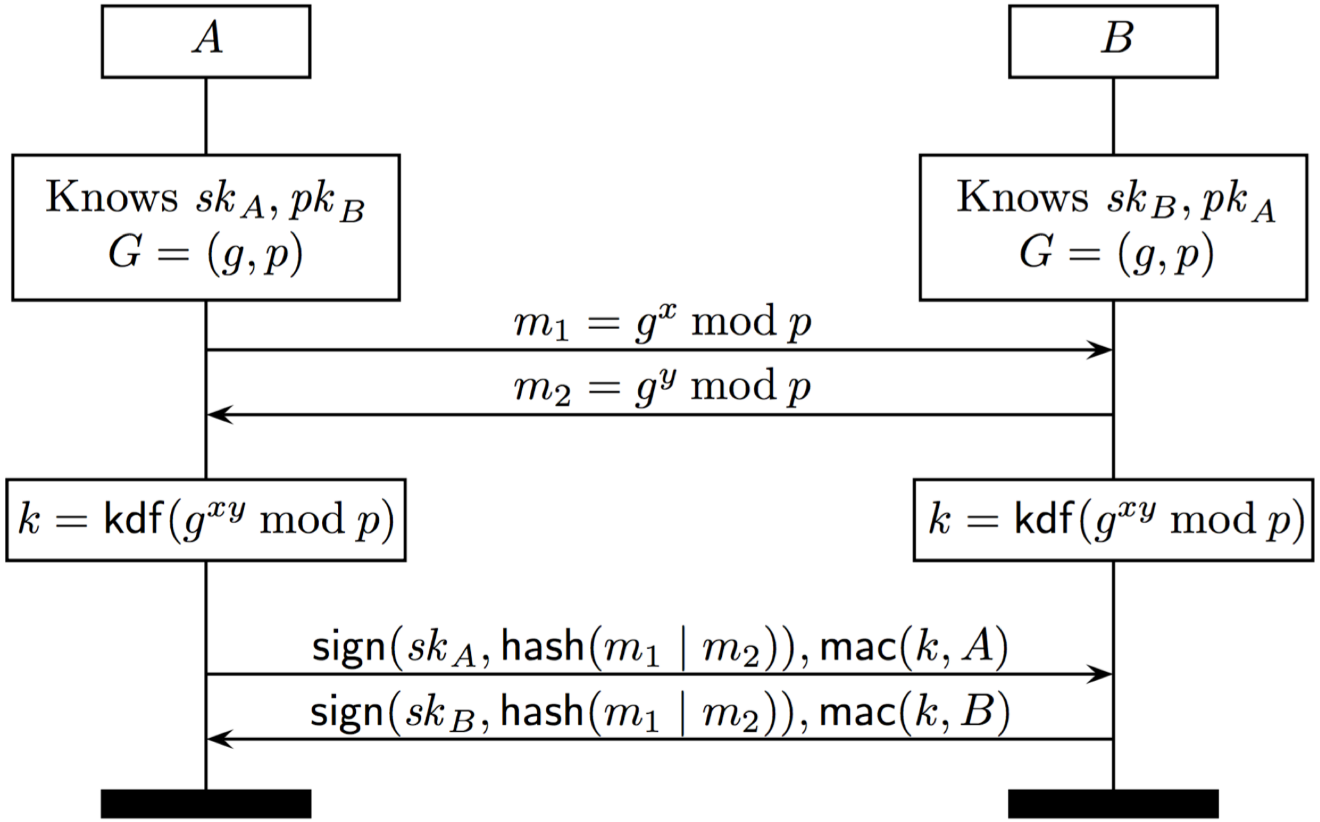
Classic man-in-the-middle attack



Authenticated Diffie-Hellman (SIGMA)



Core Cryptographic Constructions



Diffie-Hellman Key Exchange

- *Assumption:* GapDH/ODH/…

Hash Function

- *Assumption:* Collision resistance/…

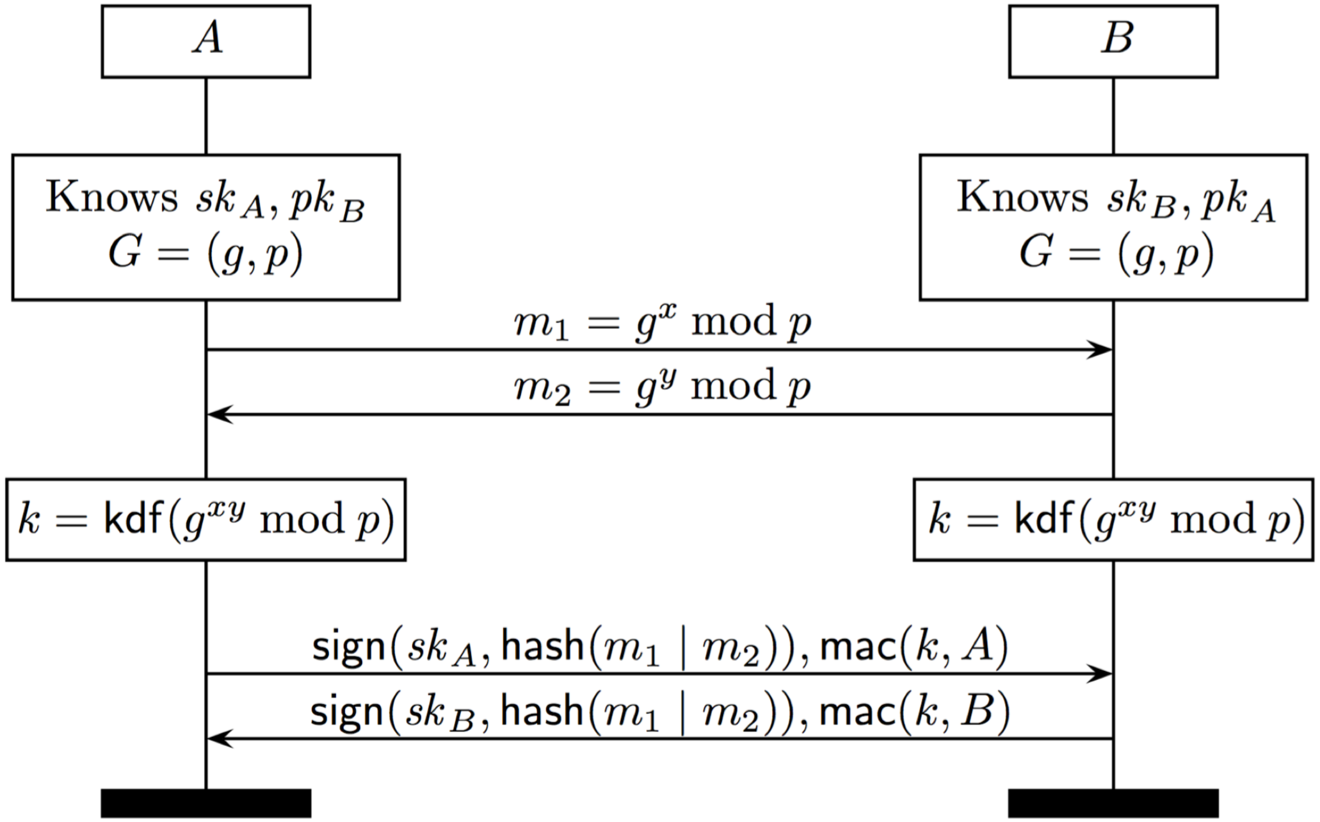
Digital Signature Scheme

- *Assumption:* UF-CMA/…

Message Authentication Code

- *Assumption:* PRF/…

Configuration: Supported Crypto Algorithms



Diffie-Hellman Group

- EC-256, DH-2048, DH-512

Hash Function

- SHA-256, SHA-1, MD5

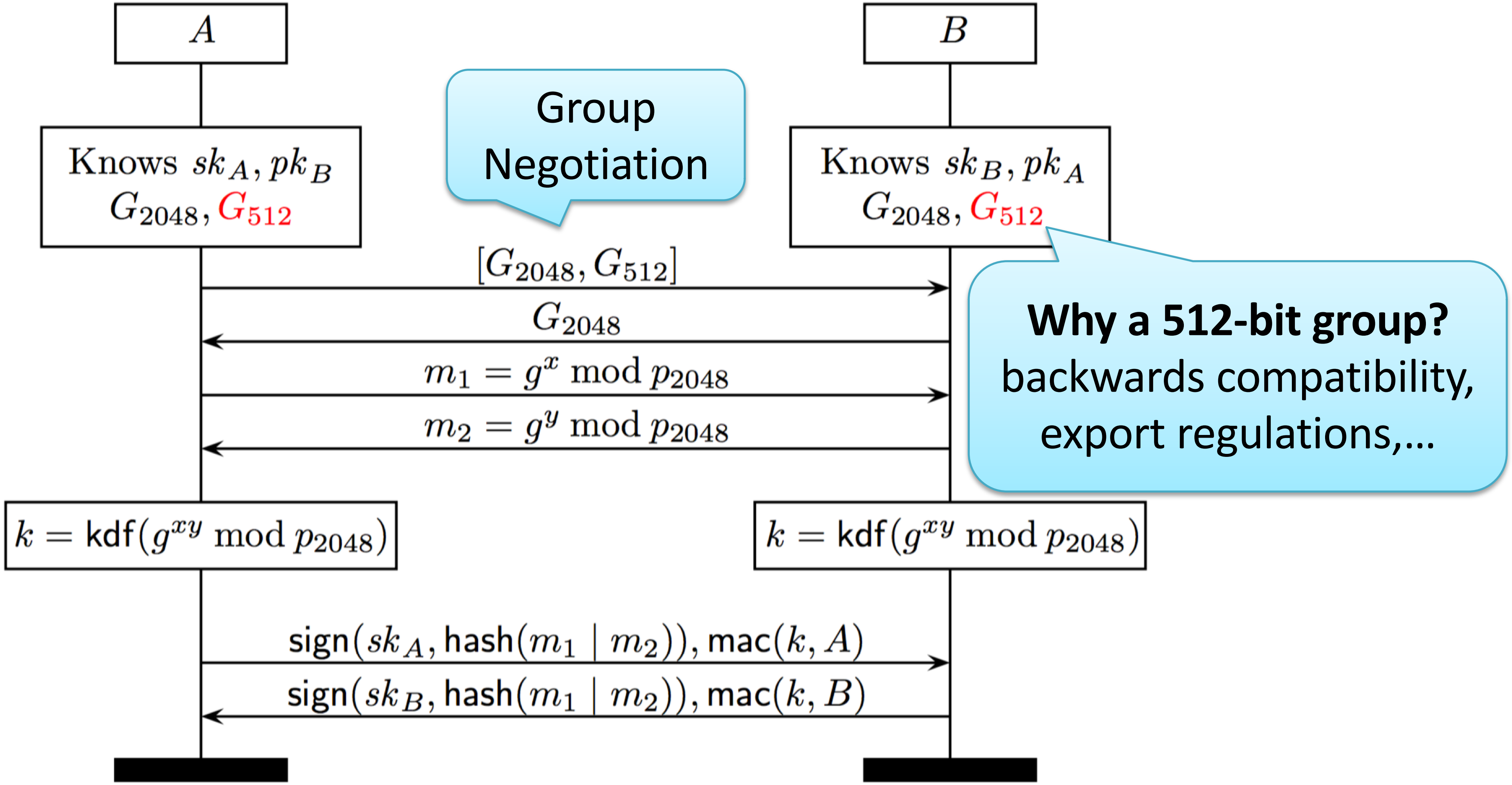
Digital Signature Scheme

- RSA-PSS, ECDSA, RSA-PKCS#1

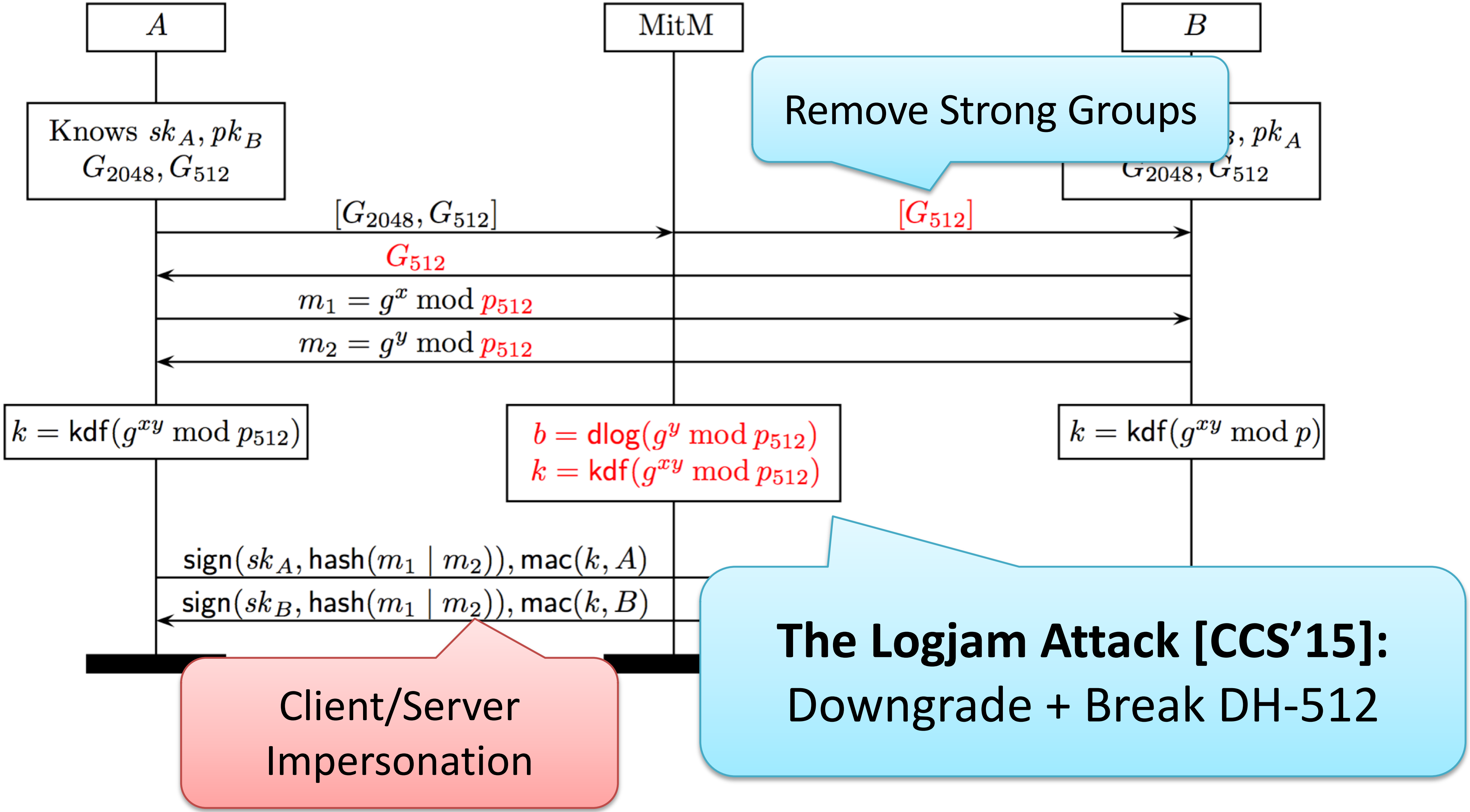
Message Authentication Code

- HMAC-SHA256, Truncated HMAC

Negotiation: Choosing a Diffie-Hellman Group



Logjam: DH group downgrade attack



Downgrade Protections in TLS 1.2

In TLS 1.2, both client and server MAC the full transcript to prevent tampering:

$$\textbf{mac}(k, [G_{2048}, G_{512}] \mid G_{512} \mid m_1 \mid m_2)$$

But it's too late, we already used G_{512} to compute k

$$k = \textbf{kdf}(g^{xy} \bmod p_{512})$$

so, the attacker can compute k and forge the MAC

The TLS 1.2 downgrade protection mechanism itself depends on downgradeable parameters!

- No easy fix except disabling all weak Diffie-Hellman groups

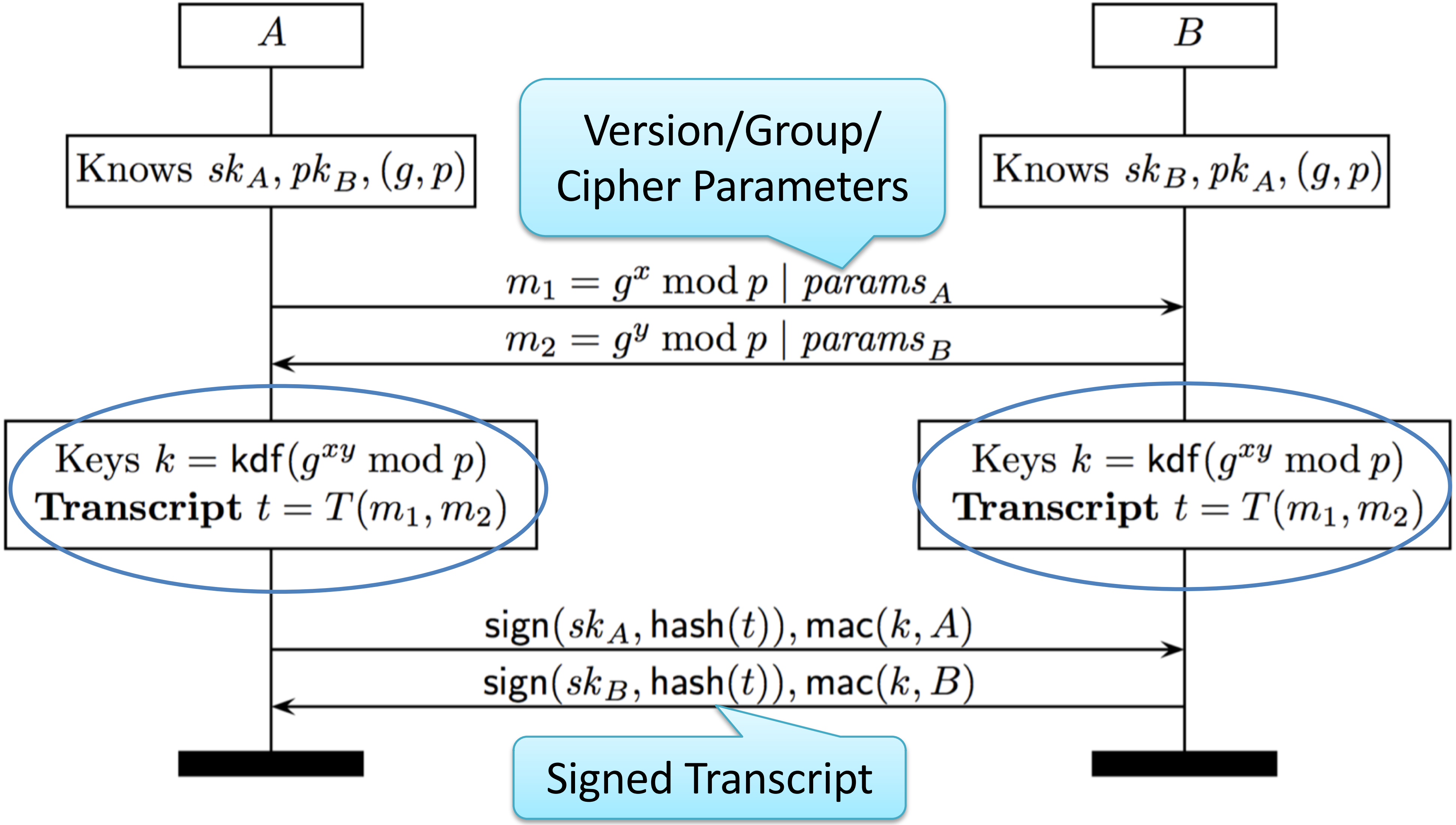
Downgrade Protections in TLS 1.3

Sign the full handshake transcript

- **sign**(k, **hash**($[G_{2048}, G_{512}] \mid G_{512} \mid m_1 \mid m_2$))
- Prevents Logjam in TLS 1.3

- Does this prevent other downgrade attacks?

SIGMA with Generic Negotiation



Downgrade Protections in TLS 1.3

Sign the full handshake transcript

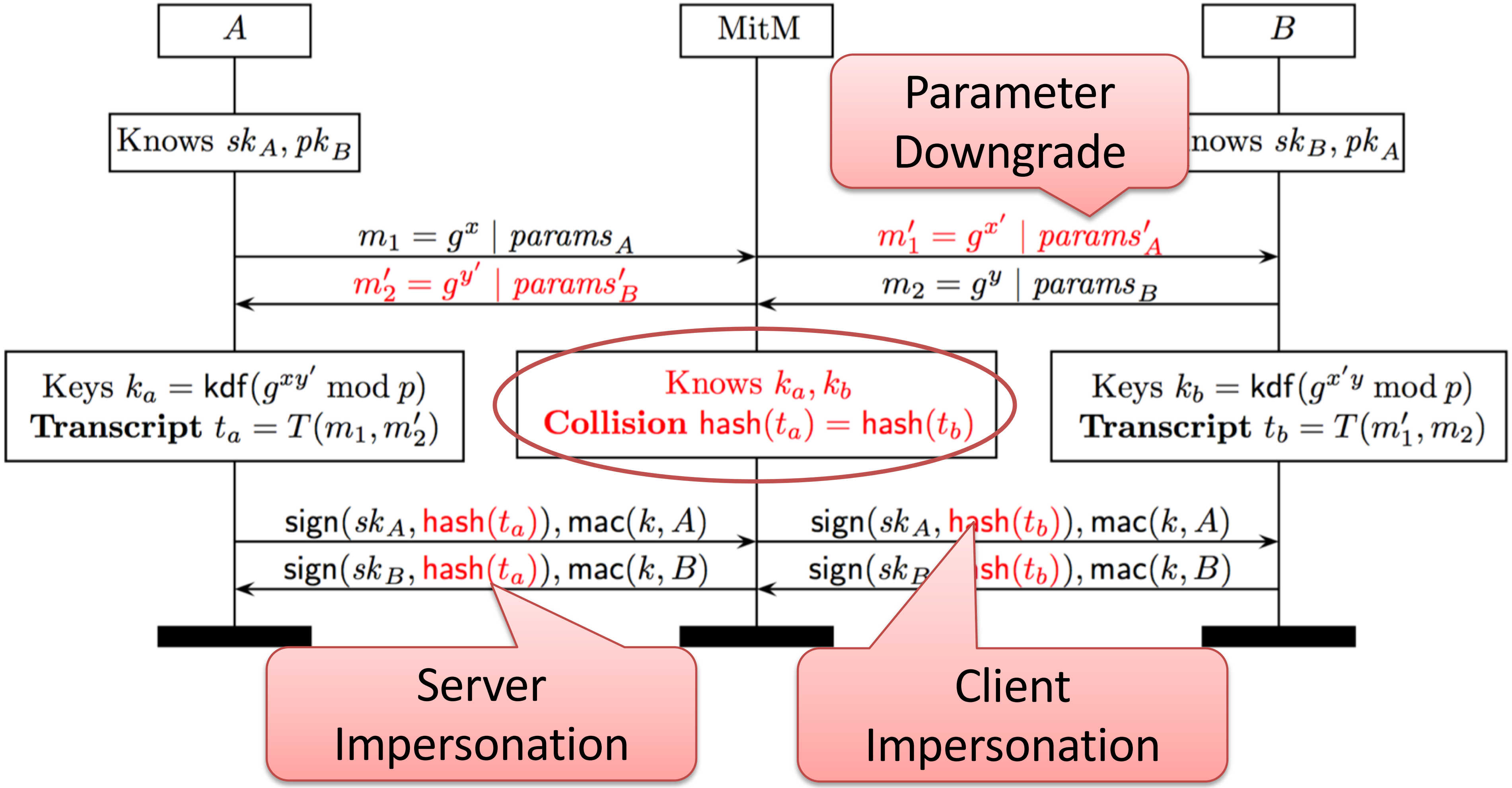
- **sign**(sk_B, **hash**(m_1 | m_2))

How weak can this **hash** function be?

- do we really need collision resistance?
- do we only need 2^{nd} preimage resistance?
- E.g. is it still safe to use MD5, SHA-1 in TLS 1.3 signatures?

SLOTH: Transcript Collision Attacks

[Bhargavan, Leurent, NDSS'16]



Signature/Hash Function Downgrade in TLS 1.3

TLS 1.3 signs the full transcript to prevent tampering

- **sign**(sk_B, **hash**(m_1 | m_2))
- This prevents many downgrade attacks including Logjam

TLS 1.3 cannot prevent signature/hash function downgrades

- We need to eliminate all weak signature schemes from TLS 1.3
- We need to eliminate all weak hash functions from TLS 1.3
- We still need to protect against TLS 1.3 → TLS 1.2 downgrades
- Otherwise, an attacker hop down to TLS 1.2 and bypass TLS 1.3

Proving Downgrade Resilience for TLS 1.3

[Bhargavan, Brzuska, Fournet, Green, Kohlweiss, Zanella-Béguelin, IEEE S&P 2016]

Agile Key Exchange Protocols

- We consider two party AKE protocols ($I \rightarrow R$)
- Key exchange inputs:
  - *config*$_I$ & *config*$_R$: supported versions, ciphers, etc.
  - *creds*$_I$ & *creds*$_R$: long-term private keys
- Key exchange outputs:
  - *uid*: unique session identifier
  - *k*: session key
  - *mode*: negotiated version, cipher, etc.

Agile AKE Security Goals

- ***Partnering***
  at most one honest partner exists with same *uid*
- ***Agreement***
  if my negotiated *mode* uses only strong algorithms,
  then my partner and I agree on *k* and *mode*
- ***Confidentiality***
  if my negotiated *mode* uses only strong algorithms,
  the key *k is* only known to me and my partner
- ***Authenticity***
  if my intended peer is authenticated and honest,
  and my negotiated *mode* uses only strong algorithms,
  then at least one partner with same *uid* exists

Agile Agreement vs. Downgrade Attacks

- ***Agreement***
  if my negotiated *mode* uses only strong algorithms,
  then my partner and I agree on *k* and *mode*

- Agreement does not guarantee that the protocol *will* negotiate a strong mode
  – It does not forbid Logjam-like attacks
  – Only protects against downgrades if all algorithms in the **intersection** of $config_I$ & $config_R$ are strong
  – What if $config_I$,$config_R$ both include a weak algorithm ?

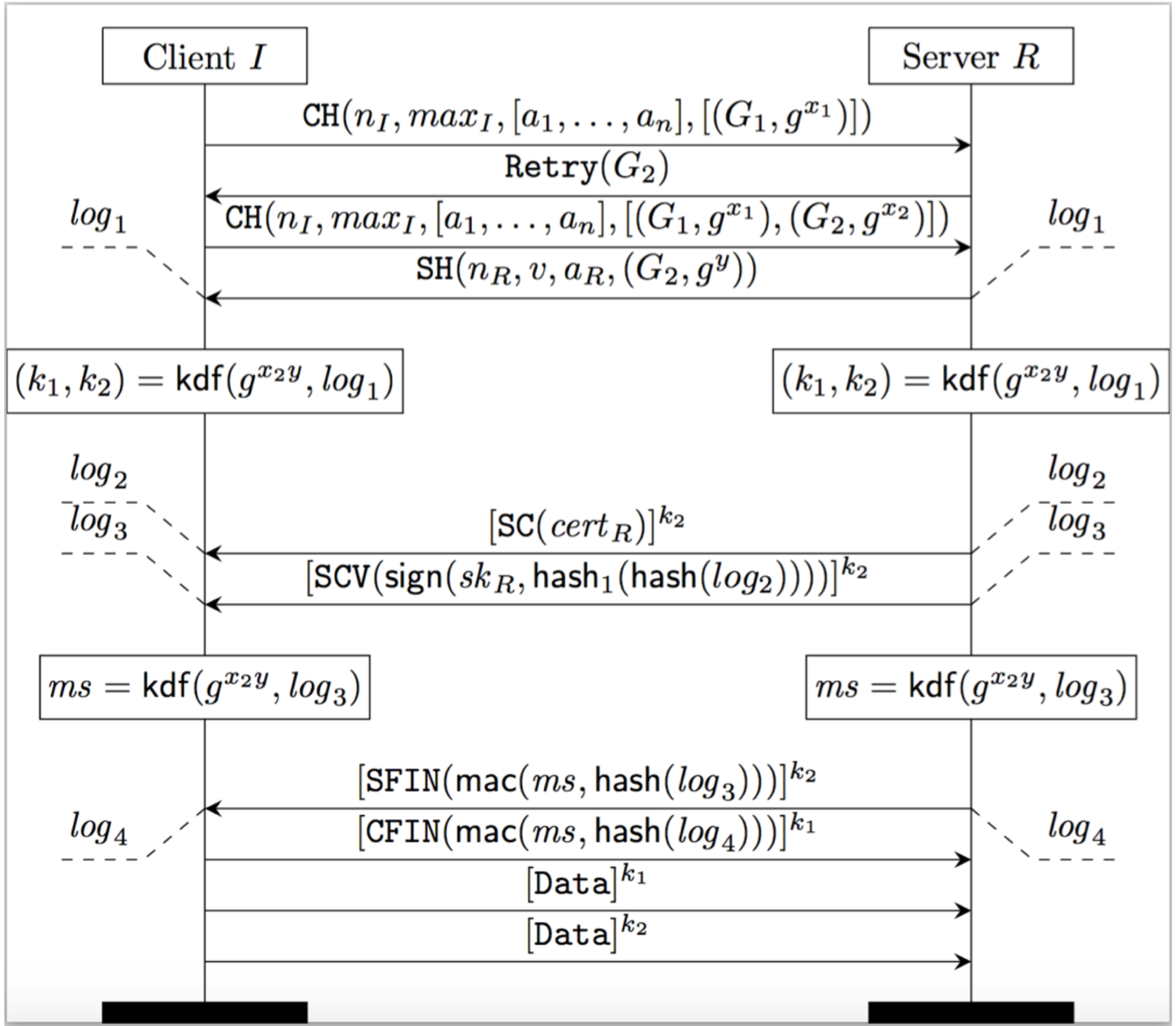
A New Security Goal: Downgrade Resilience

- ***Ideal Negotiation:*** *Nego*($config_I$, $config_R$)
  Informally, the *mode* that would have been negotiated in the absence of an attacker

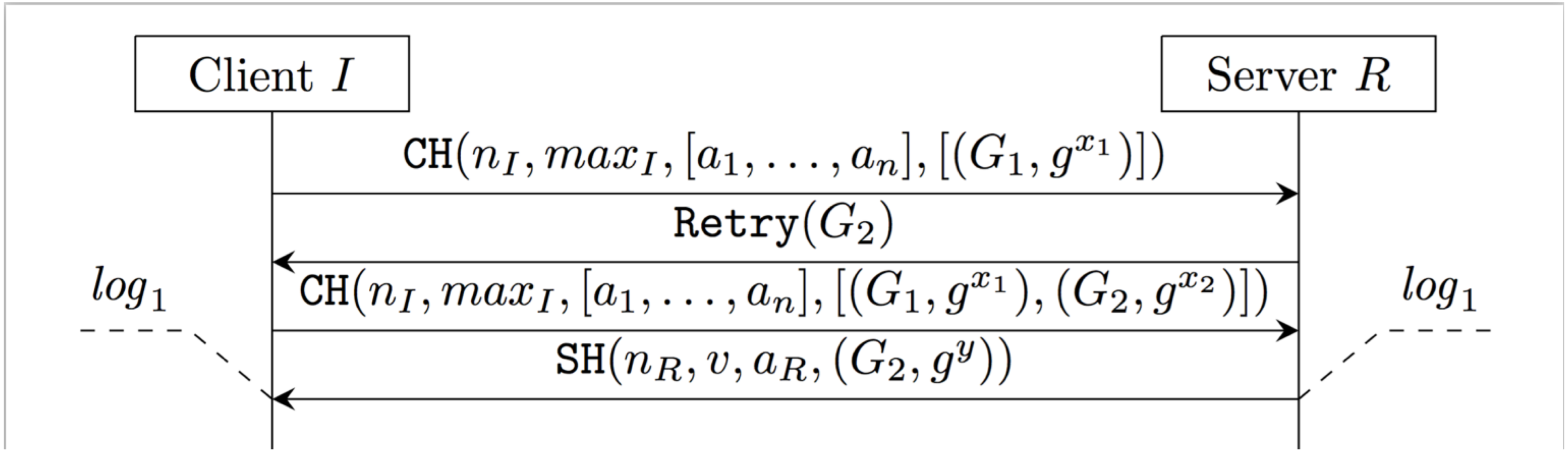
- ***Downgrade Resilience***
  The protocol should negotiate the *ideal* mode even in the presence of the attacker

$$mode = Nego(config_I, config_R)$$

TLS 1.3 Negotiation Sub-Protocol



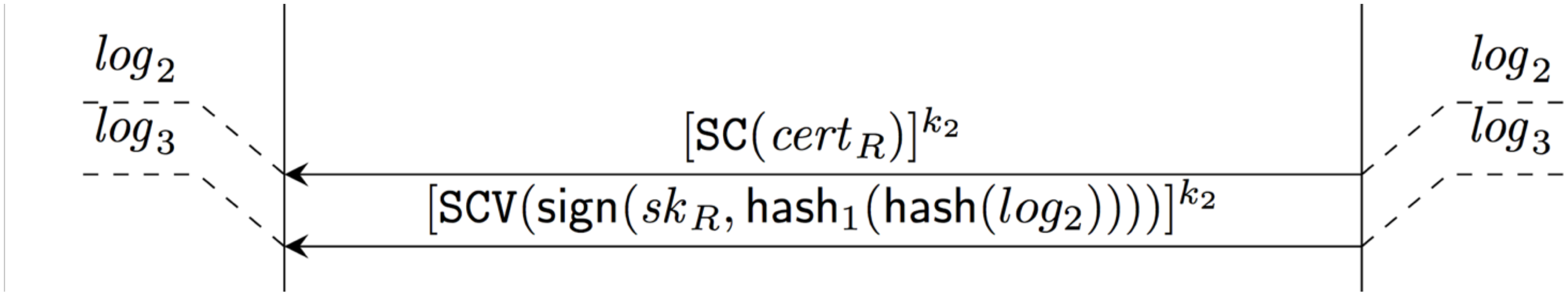
1: Group Negotiation with Retry



Server can ask client to retry with another group

- What if attacker sends a bogus Retry?
- ***Fix:*** The transcript hashes *both* hellos and retry to prevent tampering of Retry messages.

2: Full Transcript Signatures



Client and Server both sign *full* transcript

- Only RSA-PSS/ECDSA/EdDSA signatures allowed
- Only SHA-256 or newer hash algorithms allowed
- Prevents many downgrade attacks e.g. Logjam

3: Preventing Version Downgrade

TLS 1.3 clients and servers will likely also support TLS 1.2

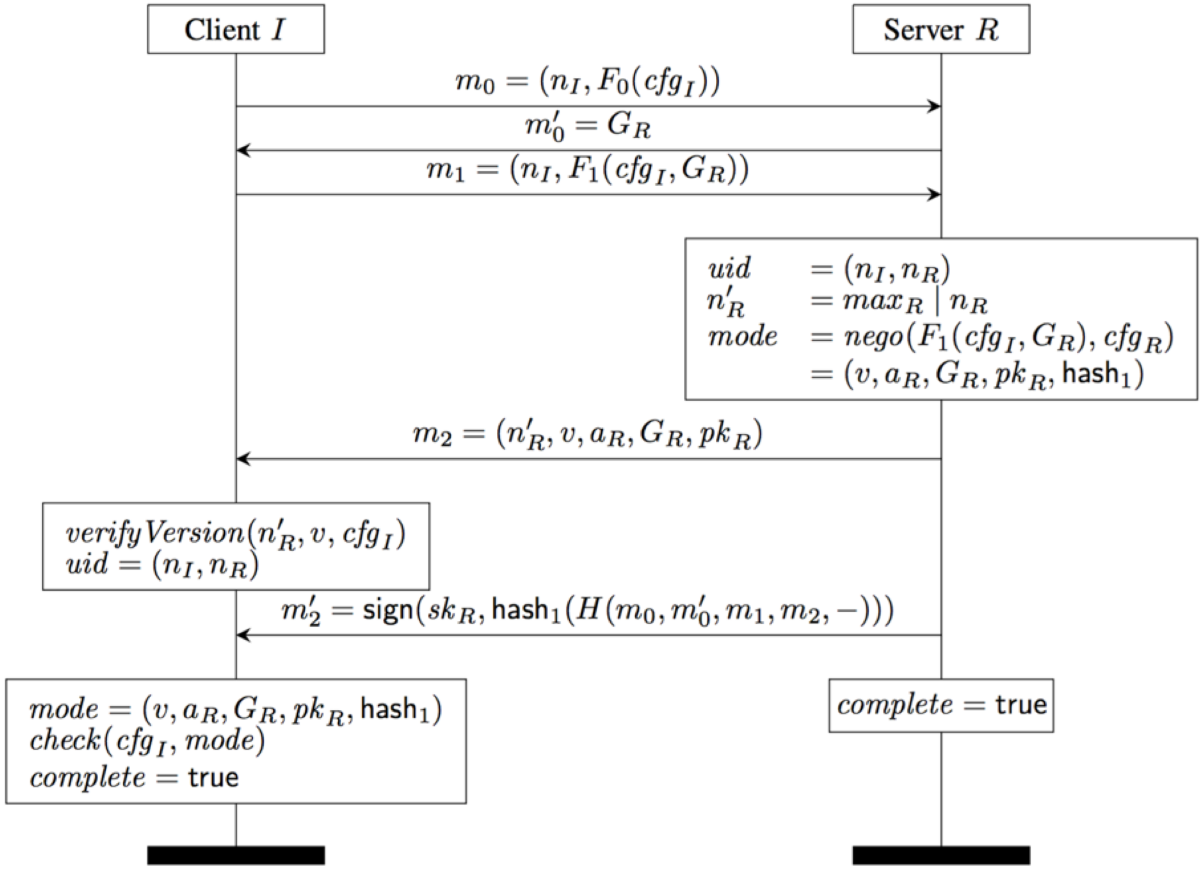
- What if the attacker downgrades all connections to TLS 1.2?

- ***Fix***: the TLS 1.3 server includes a fixed 64-bit pattern in the server nonce when negotiating a lower protocol version
  - Server nonce is signed in all signature ciphersuites in TLS 1.0-1.3
  - Protects downgrades to TLS 1.0-1.2 signature ciphersuites
  - Does not prevent downgrade to RSA encryption ciphersuites

TLS 1.3 Negotiation is Downgrade Resilient

We can prove downgrade resilience for the *negotiation sub-protocol* of TLS 1.3+1.2, if only signature ciphersuites with collision-resistant hash functions are enabled in TLS 1.2.

- Does not account for all of TLS 1.3
- Painful to extend manual crypto proof to full protocol



Symbolically Analyzing full TLS 1.3 + TLS 1.2 (to detect downgrade attacks)

[Bhargavan, Blanchet, Kobeissi, IEEE S&P 2017]

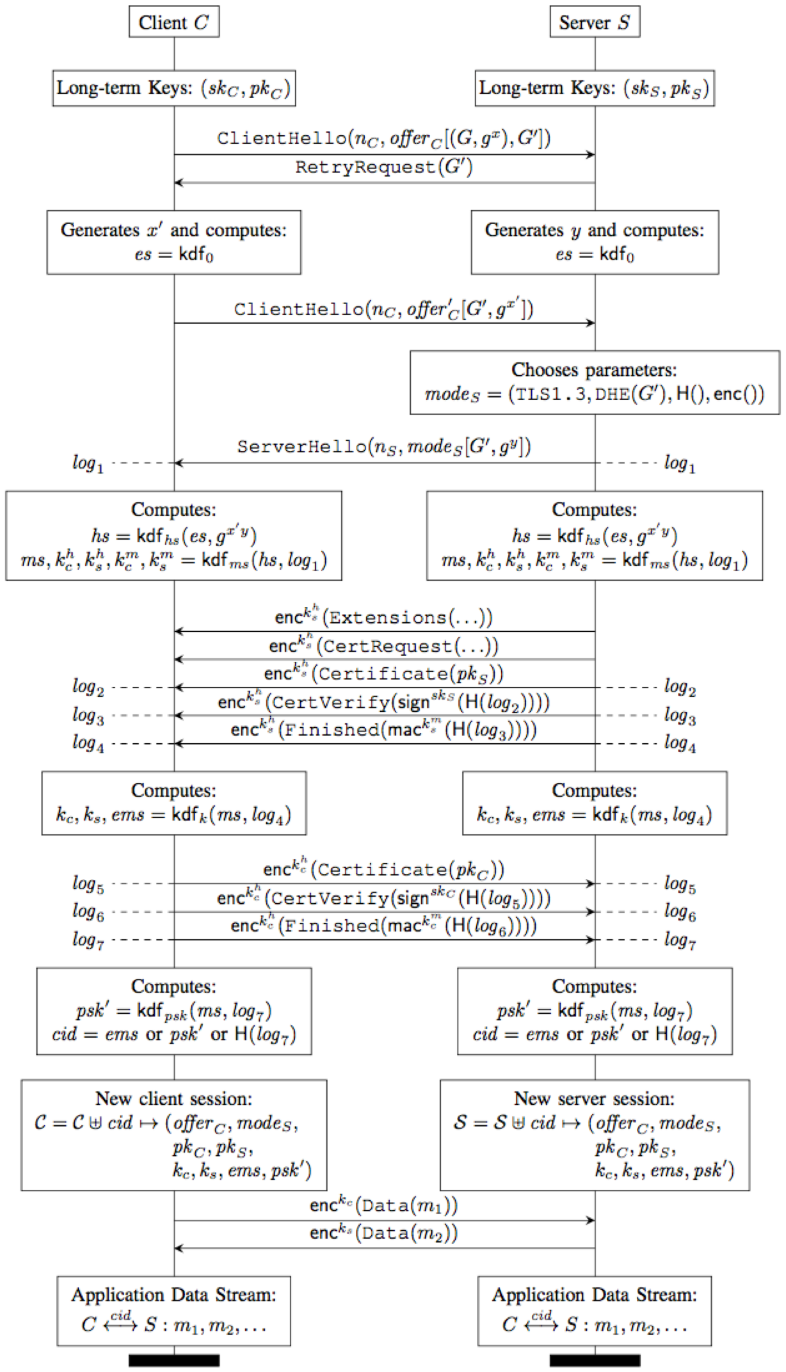
Modeling TLS 1.3 in ProVerif

TLS 1.3 1-RTT handshake

- 12 messages in 3 flights, 16 derived keys, then data exchange

+ 0-RTT + TLS 1.2

- **Protocol model**: 500 lines
- **Threat model**: 400 lines
- **Security goals**: 200 lines



Key Derivation Functions:

$\mathsf{hkdf\text{-}extract}(k, s) = \mathsf{HMAC\text{-}H}^k(s)$

$\mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, h) =$
$\quad \mathsf{HMAC\text{-}H}^s(len_{\mathsf{H}()} \| \text{“TLS 1.3,”} \| l \| h \| \mathtt{0x01})$

$\mathsf{derive\text{-}secret}(s, l, m) = \mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, \mathsf{H}(m))$

1-RTT Key Schedule:

$\mathsf{kdf}_0 = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, 0^{len_{\mathsf{H}()}})$

$\mathsf{kdf}_{hs}(es, e) = \mathsf{hkdf\text{-}extract}(es, e)$

$\mathsf{kdf}_{ms}(hs, log_1) = ms, k_c^h, k_s^h, k_c^m, k_s^m$ where
$\quad ms = \mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{H}()}})$
$\quad hts_c = \mathsf{derive\text{-}secret}(hs, \mathsf{hts_c}, log_1)$
$\quad hts_s = \mathsf{derive\text{-}secret}(hs, \mathsf{hts_s}, log_1)$
$\quad k_c^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{key}, \text{“”})$
$\quad k_c^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{finished}, \text{“”})$
$\quad k_s^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{key}, \text{“”})$
$\quad k_s^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{finished}, \text{“”})$

$\mathsf{kdf}_k(ms, log_4) = k_c, k_s, ems$ where
$\quad ats_c = \mathsf{derive\text{-}secret}(ms, \mathsf{ats_c}, log_4)$
$\quad ats_s = \mathsf{derive\text{-}secret}(ms, \mathsf{ats_s}, log_4)$
$\quad ems = \mathsf{derive\text{-}secret}(ms, \mathsf{ems}, log_4)$
$\quad k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ats_c, \mathsf{key}, \text{“”})$
$\quad k_s = \mathsf{hkdf\text{-}expand\text{-}label}(ats_s, \mathsf{key}, \text{“”})$

$\mathsf{kdf}_{psk}(ms, log_7) = psk'$ where
$\quad psk' = \mathsf{derive\text{-}secret}(ms, \mathsf{rms}, log_7)$

PSK-based Key Schedule:

$\mathsf{kdf}_{es}(psk) = es, k^b$ where
$\quad es = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, psk)$
$\quad k^b = \mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$

$\mathsf{kdf}_{0RTT}(es, log_1) = k_c$ where
$\quad ets_c = \mathsf{derive\text{-}secret}(es, \mathsf{ets_c}, log_1)$
$\quad k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ets_c, \mathsf{key}, \text{“”})$

```
let Server13() =
   (get preSharedKeys(a,b,psk) in
    in(io,ch:msg);
    let CH(cr,offer) = ch in
    let nego(=TLS13,DHE_13(g,gx),hhh,aaa,Binder(m)) = offer in
    let (early_secret:bitstring,kb:mac_key) = kdf_es(psk) in
    let zoffer = nego(TLS13,DHE_13(g,gx),hhh,aaa,Binder(zero)) in
    if m = hmac(StrongHash,kb,msg2bytes(CH(cr,zoffer))) then
    let (kc0:ae_key,ems0:bitstring) =
        kdf_k0(early_secret,msg2bytes(ch)) in
    insert serverSession0(cr,psk,offer,kc0,ems0);

    new sr:random;
    in(io,SH(xxx,mode));
    let nego(=TLS13,DHE_13(=g,eee),h,a,pt) = mode in
    let (y:bitstring,gy:element) = dh_keygen(g) in
    let mode = nego(TLS13,DHE_13(g,gy),h,a,pt) in
    out(io,SH(sr,mode));
    let log = (ch,SH(sr,mode)) in
    get longTermKeys(sn,sk,p) in
    event ServerChoosesVersion(cr,sr,p,TLS13);
    event ServerChoosesKEX(cr,sr,p,TLS13,DHE_13(g,gy));
    event ServerChoosesAE(cr,sr,p,TLS13,a);
    event ServerChoosesHash(cr,sr,p,TLS13,h);

    let gxy = e2b(dh_exp(g,gx,y)) in
    let handshake_secret = kdf_hs(early_secret,gxy) in
    let (master_secret:bitstring,chk:ae_key,shk:ae_key,cfin:mac_key,sfin:mac_key) =
        kdf_ms(handshake_secret,log) in
    out(io,(chk,shk));
```

```
letfun kdf_es(psk:preSharedKey) =
       let es = hkdf_extract(zero,psk2b(psk)) in
       let kb = derive_secret(es,tls13_resumption_psk_binder_key,zero) in
       (es,b2mk(kb)).


letfun kdf_k0(es:bitstring,log:bitstring) =
       let atsc0 = derive_secret(es, tls13_client_early_traffic_secret, log) in
       let kc0   = hkdf_expand_label(atsc0,tls13_key,zero) in
       let ems0   = derive_secret(es,tls13_early_exporter_master_secret,log) in
       (b2ae(kc0),ems0).


letfun kdf_hs(es:bitstring,e:bitstring) =
       let extra = derive_secret(es,tls13_derived,hash(StrongHash,zero)) in
       hkdf_extract(extra,e).

letfun kdf_ms(hs:bitstring,log:bitstring) =
       let extra = derive_secret(hs,tls13_derived,hash(StrongHash,zero)) in
       let ms =   hkdf_extract(hs , zero) in
       let htsc = derive_secret(hs, tls13_client_handshake_traffic_secret, log) in
       let htss = derive_secret(hs, tls13_server_handshake_traffic_secret, log) in
       let kch =  hkdf_expand_label(htsc,tls13_key,zero) in
       let kcm =  hkdf_expand_label(htsc,tls13_finished,zero) in
       let ksh =  hkdf_expand_label(htss,tls13_key,zero) in
       let ksm =  hkdf_expand_label(htss,tls13_finished,zero) in
       (ms,b2ae(kch),b2ae(ksh),b2mk(kcm),b2mk(ksm)).
```

TLS 1.3 model in ProVerif syntax

Defining a Symbolic Threat Model

Classic Needham-Schroeder/Dolev-Yao network adversary

- **Can** read/write any message on public channels
- **Can** participate in some sessions as client or server
- **Can** compromise some long-term keys
- **Cannot** break strong crypto algorithms or guess encryption keys

We extend the model to allow attackers to break weak crypto

- Each primitive is parameterized by an algorithm
- Given a **strong** algorithm, the primitive behaves ideally
- Given a **weak** algorithm, the primitive completely breaks
- Conservative model, may not always map to real exploits

Writing and Verifying Security Goals

We state security queries for data sent between honest peers

- **Secrecy:** messages between honest peers are unknown to an adversary
- **Authenticity:** messages between honest peers cannot be tampered
- **No Replay:** messages between honest peers cannot be replayed
- **Forward Secrecy:** secrecy holds even if the peers' long-term keys are leaked after the session is complete

Secrecy query for msg(conn,S) sent from client C to server S

query not attacker(msg(conn,S))

Refining Security Queries

- **QUERY:** Is msg(conn,S) secret?

query not attacker(msg(conn,S))

- **FALSE**: ProVerif finds a counterexample if S’s private key is compromised

Refining Security Queries

- **QUERY:** Is msg(conn,S) secret
  as long as S is uncompromised?

```
query attacker(msg(conn,S)) ==>
      event(WeakOrCompromisedKey(S))
```

- **FALSE:** ProVerif finds a counterexample if the AE algorithm is weak

Refining Security Queries

- **QUERY:** Is msg(conn,S) secret
  as long as S is uncompromised
  and only strong AE algorithms are used?

```
query attacker(msg(conn,S)) ==>
      event(WeakOrCompromisedKey(S)) ||
      event(ServerChoosesAE(conn,WeakAE))
```

- **FALSE:** ProVerif finds a counterexample if the DH group is weak

Refining Security Queries

- Strongest secrecy query that can be proved in our model

```
query attacker(msg(conn,S)) ==>
        event(WeakOrCompromisedKey(S)) ||
        event(ServerChoosesAE(conn,S,WeakAE)) ||
        event(ServerChoosesKEX(conn,S,WeakDH)) ||
        event(ServerChoosesKEX(conn',S,WeakRSADecryption) ||
        event(ServerChoosesHash(conn',S,WeakHash))
```

- **TRUE:** ProVerif finds no counterexample

Symbolic Security for TLS 1.3 + TLS 1.2

Messages on a TLS 1.3 connection between honest peers are secret:

1. If the connection does not use a weak AE algorithm,
2. the connection does not use a weak DH group,
3. the server **never uses** a weak hash algorithm for signing, and
4. the server **never participates** in TLS 1.2 RSA key exchange

Analysis confirms preconditions for downgrade resilience in TLS 1.3

- Identifies weak algorithms in TLS 1.2 that can harm TLS 1.3 security

Mechanized Crypto Proofs for TLS 1.3

We also model and verify TLS 1.3 in CryptoVerif

- Handshake with PSK and/or (EC)DHE, optional client authentication
- Record protocol with key update, 0-RTT, 0.5-RTT, 1-RTT application data
- We do not model: negotiation, legacy versions, post-handshake auth
- **Full model:** ~5000 lines (including ~2500 lines of assumptions)

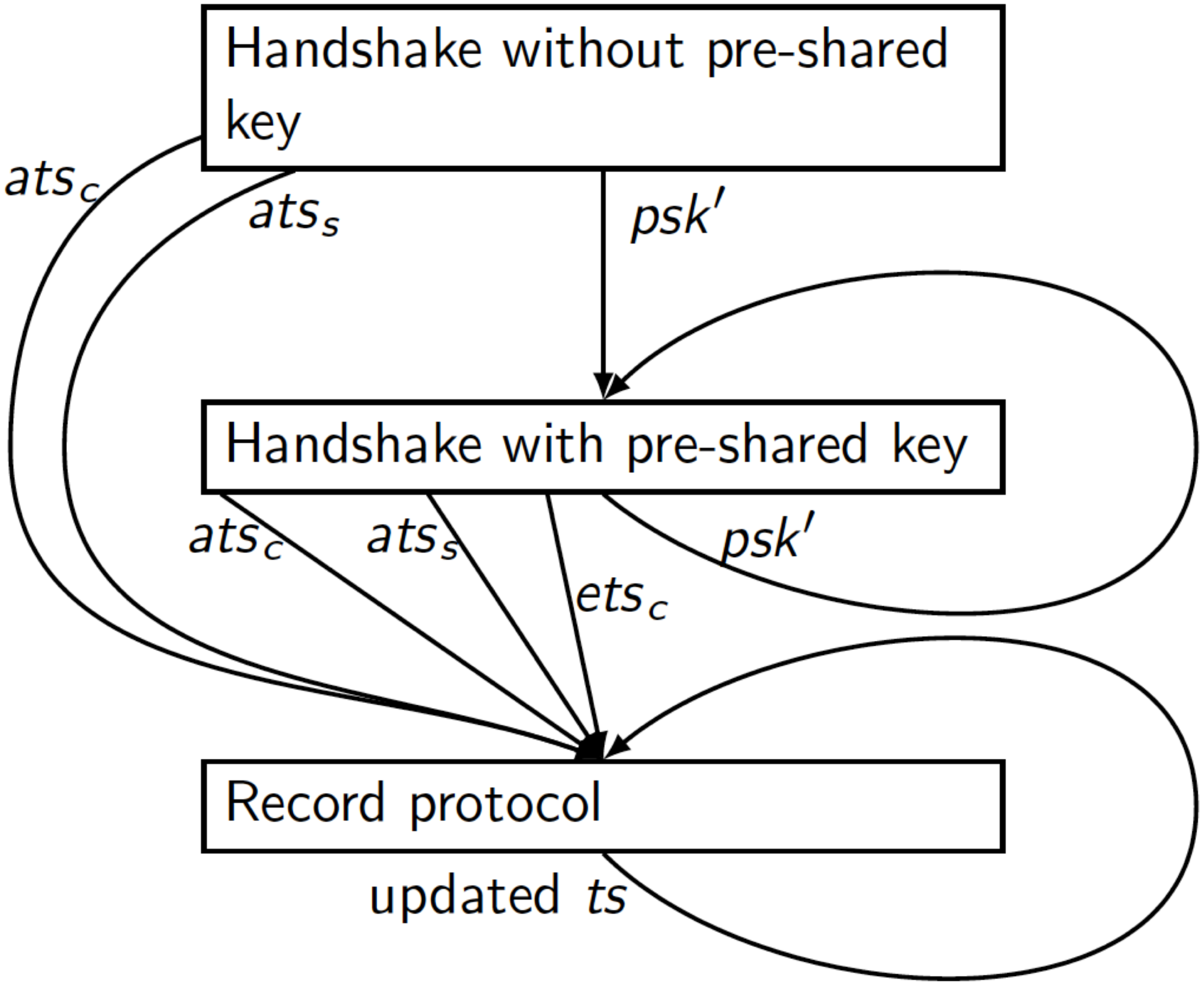
CryptoVerif proofs are semi-automated and require user guidance

- The proof is a sequence of game transformations
- Each step depends on a precise crypto assumption on some primitive

Verification strategy closely follows paper crypto proofs

- Sometimes, the tool's limitations require different assumptions

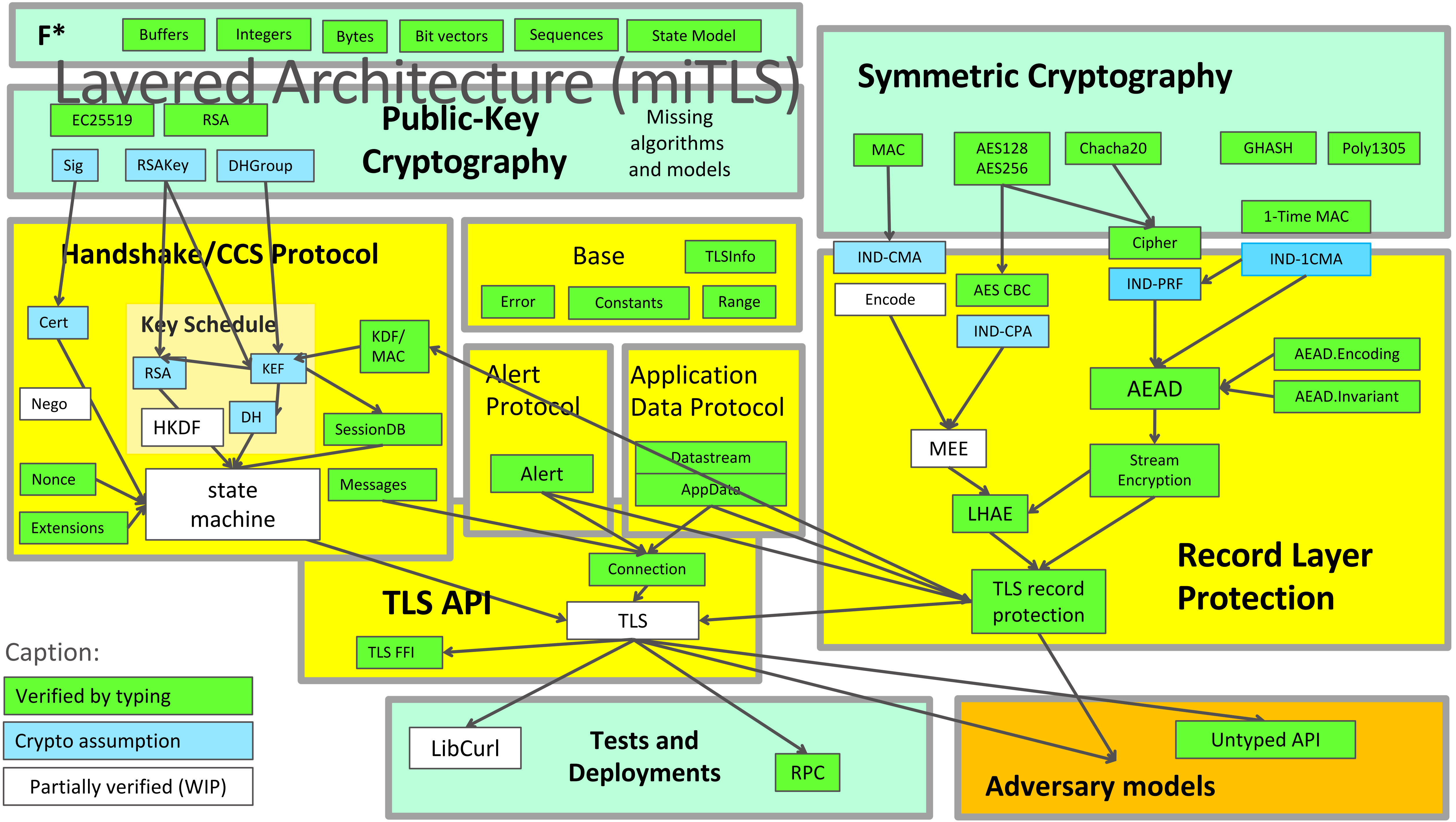
Manual Proof of Composition for full TLS 1.3



Project Everest:

Verifying a full TLS 1.3 + TLS 1.2 Implementation

[Delignat-Lavaud+, IEEE S&P 2017]


Layered Architecture (miTLS)
F*
Buffers
Integers
Bytes
Bit vectors
Sequences
State Model
EC25519
RSA
Public-Key Cryptography
Missing algorithms and models
Sig
RSAKey
DHGroup
Symmetric Cryptography
MAC
AES128 AES256
Chacha20
GHASH
Poly1305
1-Time MAC
Cipher
IND-CMA
IND-1CMA
IND-PRF
Encode
AES CBC
IND-CPA
Handshake/CCS Protocol
Cert
Key Schedule
RSA
KEF
KDF/ MAC
Nego
HKDF
DH
SessionDB
Nonce
state machine
Messages
Extensions
Base
TLSInfo
Error
Constants
Range
Alert Protocol
Alert
Application Data Protocol
Datastream
AppData
AEAD.Encoding
AEAD
AEAD.Invariant
MEE
Stream Encryption
LHAE
Record Layer Protection
TLS record protection
Connection
TLS API
TLS
TLS FFI
Caption:
Verified by typing
Crypto assumption
Partially verified (WIP)
LibCurl
Tests and Deployments
RPC
Untyped API
Adversary models

Everest Verification Toolchain

source code, specs, security definitions,
crypto games & constructions, proofs…

verify all properties
(using automated provers)
then **erase** all proofs

kreMLin

extract low-level code,
with good performance &
(some) side-channel protection

C/C++

gcc, compcert, clang, msvc

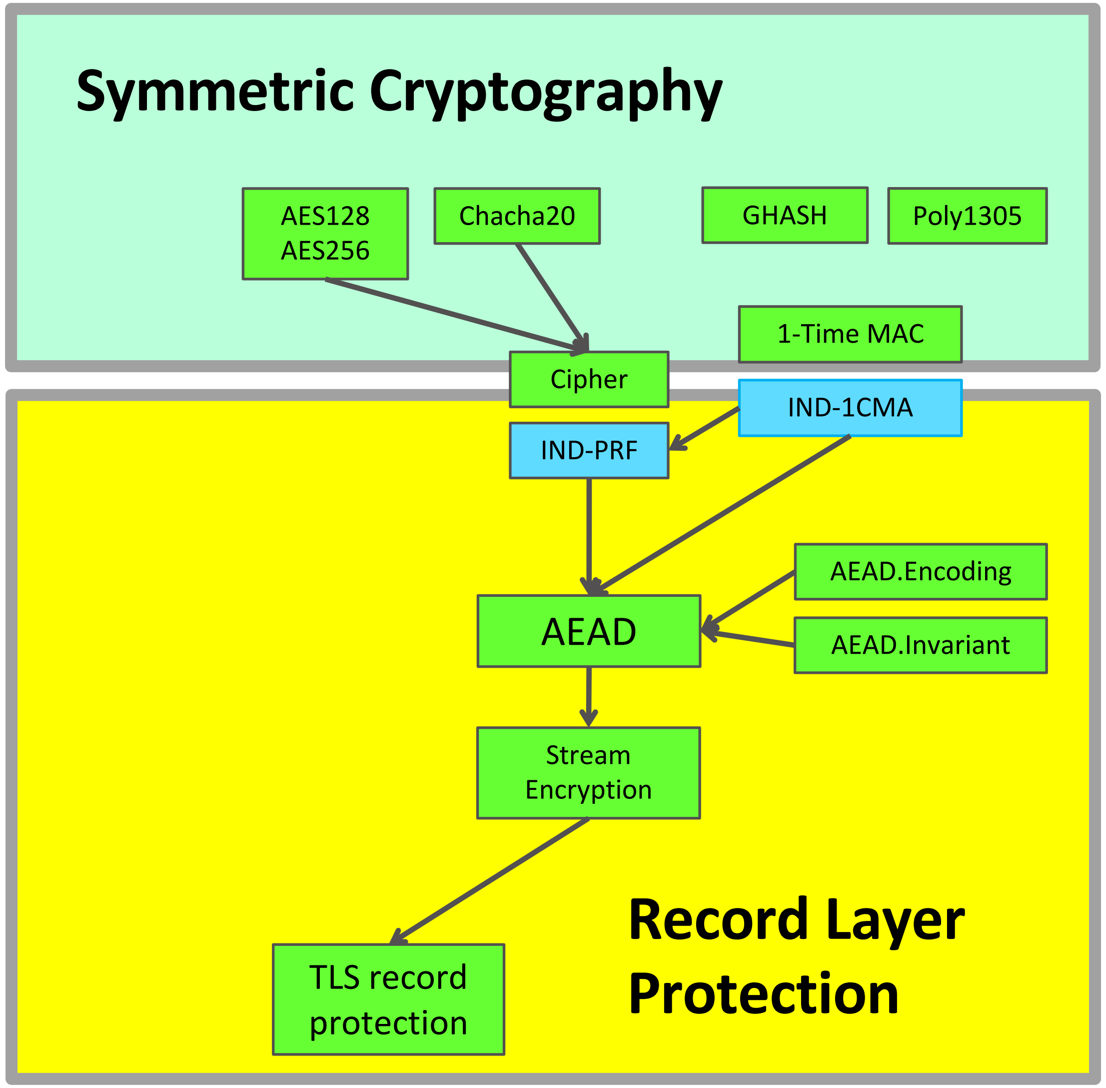
interop with rest of
TLS/HTTPS ecosystem

production code

Verified security for Agile TLS Record Layer

[Delignat-Lavaud et al., IEEE S&P'17]

1. **Agile Security definition**
2. **TLS constructions (AEAD)**
3. **Concrete security bounds**
4. **Verification**
5. **Performance**



HACL*: A Verified Crypto Library for TLS

[Zinzindohoue et al., ACM CCS'17]

Crypto library verified in F* and compiled to C

- Verified memory safety, functional correctness, and secret independence (timing side-channel resistance)
- Performance comparable with hand-coded C libraries
- Currently used in Firefox for Curve25519/Chacha20/Poly1305

Crypto algorithms used in TLS 1.3

- SHA-2*, P-256, **Curve25519**, RSA-PSS, ECDSA, EdDSA HMAC, HKDF, AES-GCM, CHACHA20-POLY1305

Conclusion

Many new issues when deploying a protocol like TLS 1.3

- Downgrade attacks, Implementation bugs, …
- Fixes proposed by academics are now built into TLS 1.3

Formal verification tools can help gain confidence in both protocol design and implementation

- **Download and use:** Tamarin, ProVerif, CryptoVerif, EasyCrypt, F*

Questions?

- ProVerif: http://proverif.inria.fr
- Tamarin: https://tamarin-prover.github.io/
- Cryptoverif: http://cryptoverif.inria.fr
- EasyCrypt: https://www.easycrypt.info
- F*: http://www.fstar-lang.org/
- Project Everest: https://project-everest.github.io/