Verifying Real-World Security Protocols

from finding attacks to proving security theorems

Karthik Bhargavan

Inria

http://prosecco.inria.fr

+

many co-authors at INRIA, Microsoft Research, …

The TLS 1.3 experiment

Formal security analysis hand-in-hand with standardization

- **Cryptographic proofs (of drafts 5,9,10)**
  [Dowling et al. CCS'15, Krawczyk et al. Euro S&P'16, Li et al. S&P'16]
- **Mechanized cryptographic proofs (of draft 18)**
  [Bhargavan et al. S&P'17]
- **Automated symbolic protocol analysis (of draft 10, 18, 20)**
  [Cremers et al. Oakland'16 and CCS'17, Bhargavan et al. S&P'17]
- **Verified implementations (of draft 18)**
  [Bhargavan et al. S&P'17 and S&P'17]

What did all these papers prove? How much effort does it take?

Can we formally analyze your shiny new crypto protocol?

Why bother with formal security analysis?

• **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]
• **HeartBleed**	Memory safety bug	[Apr’14]
• **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]
• **Sweet32**	3DES and Blowfish	[Aug’16]

BEAST	CBC predictable IVs
CRIME	Compression before Encryption
RC4	Keystream biases
Lucky 13	MAC-Encode-Encrypt CBC
HeartBleed	Memory safety bug
3Shake	Insecure resumption
POODLE	SSLv3 MAC-Encode-Encrypt
SMACK	State machine attacks
FREAK	Export-grade 512-bit RSA
LOGJAM	Export-grade 512-bit DH
SLOTH	RSA-MD5 signatures
DROWN	SSLv2 RSA-PKCS#1v1.5
Sweet32	3DES and Blowfish

CRYPTOGRAPHIC WEAKNESSES

BEAST	CBC predictable IVs
CRIME	Compression before Encryption
RC4	Keystream biases
Lucky 13	MAC-Encode-Encrypt CBC
HeartBleed	Memory safety bug
3Shake	Insecure resumption
POODLE	SSLv3 MAC-Encode-Encrypt
SMACK	State machine attacks
FREAK	Export-grade 512-bit RSA
LOGJAM	Export-grade 512-bit DH
SLOTH	RSA-MD5 signatures
DROWN	SSLv2 RSA-PKCS#1v1.5
Sweet32	3DES and Blowfish

PROTOCOL LOGIC FLAWS

BEAST	CBC predictable IVs
CRIME	Compression before Encryption
RC4	Keystream biases
Lucky 13	MAC-Encode-Encrypt CBC
HeartBleed	Memory safety bug
3Shake	Insecure resumption
POODLE	SSLv3 MAC-Encode-Encrypt
SMACK	State machine attacks
FREAK	Export-grade 512-bit RSA
LOGJAM	Export-grade 512-bit DH
SLOTH	RSA-MD5 signatures
DROWN	SSLv2 RSA-PKCS#1v1.5
Sweet32	3DES and Blowfish

IMPLEMENTATION BUGS

BEAST	CBC predictable IVs
CRIME	Compression before Encryption
RC4	Keystream biases
Lucky 13	MAC-Encode-Encrypt CBC
HeartBleed	Memory safety bug
3Shake	Insecure resumption
POODLE	SSLv3 MAC-Encode-Encrypt
SMACK	State machine attacks
FREAK	Export-grade 512-bit RSA
LOGJAM	Export-grade 512-bit DH
SLOTH	RSA-MD5 signatures
DROWN	SSLv2 RSA-PKCS#1v1.5
Sweet32	3DES and Blowfish

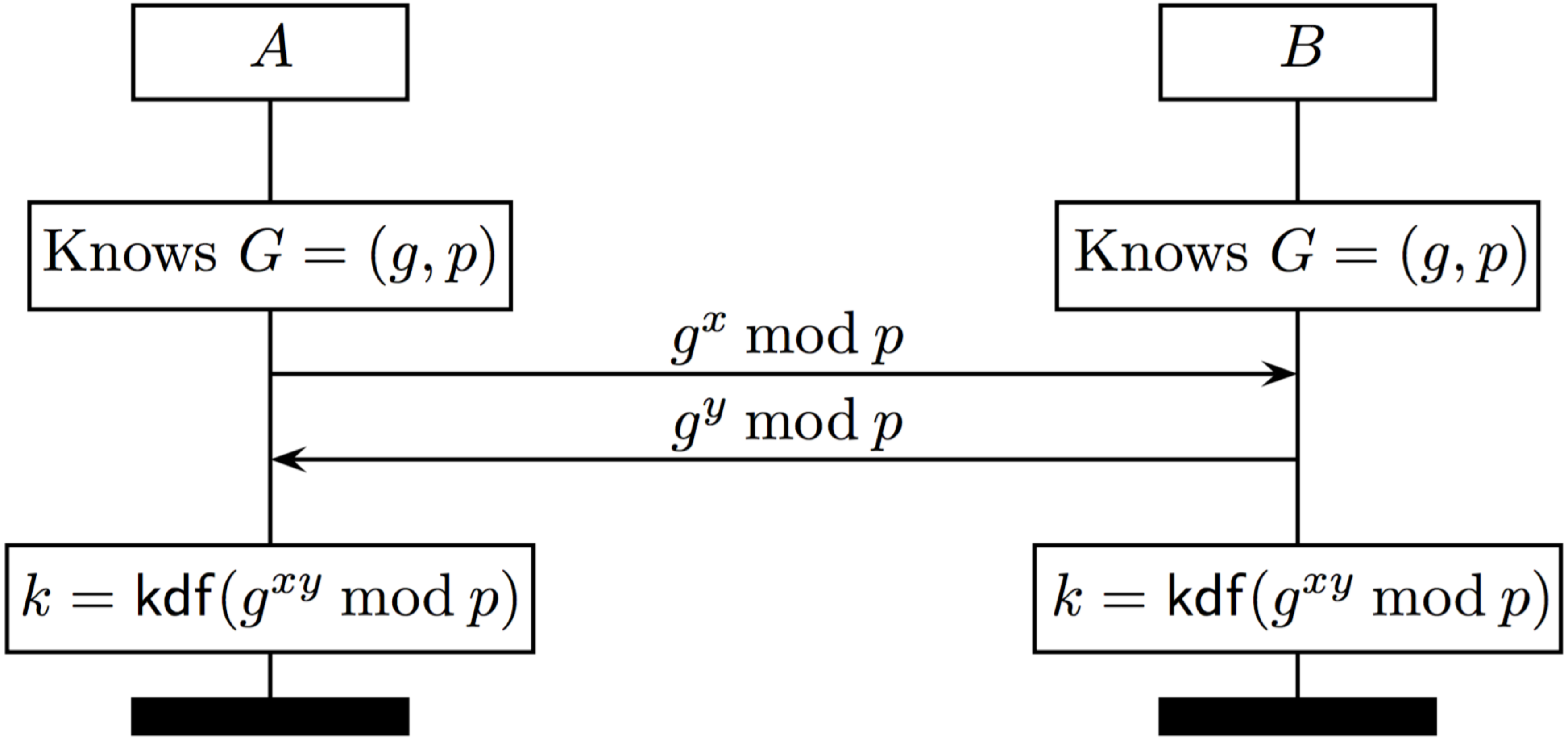
CRYPTOGRAPHIC WEAKNESSES

PROTOCOL LOGIC FLAWS

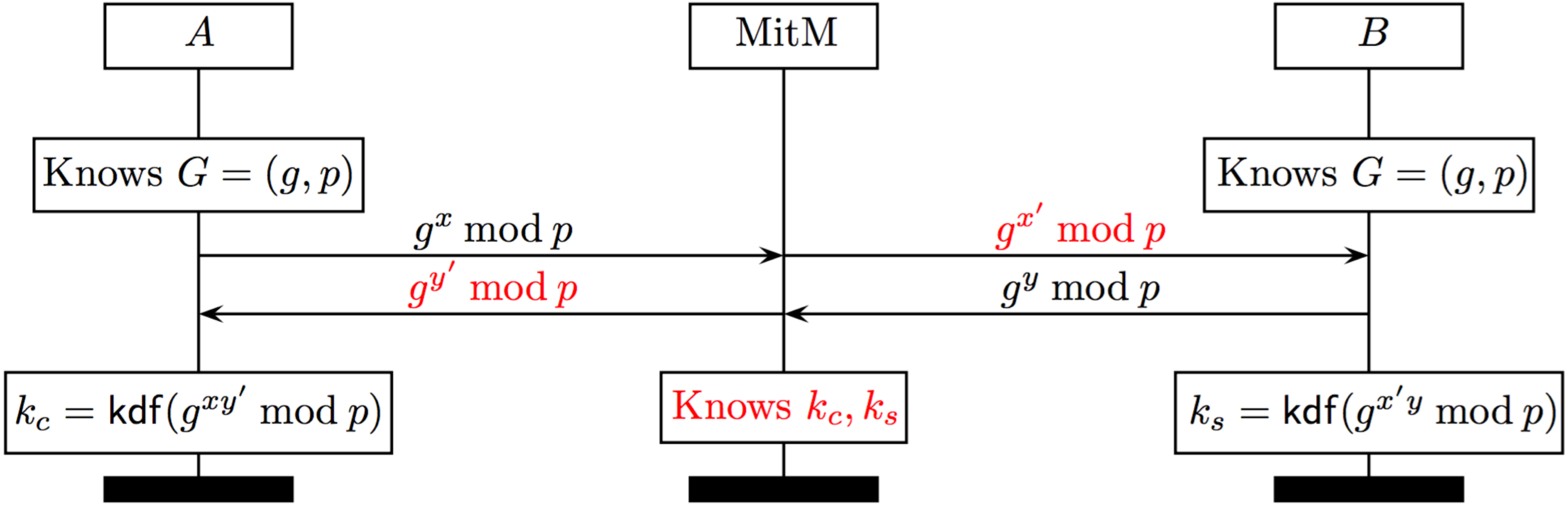
IMPLEMENTATION BUGS

Often, a combination of all of the above

Example: Diffie-Hellman key exchange

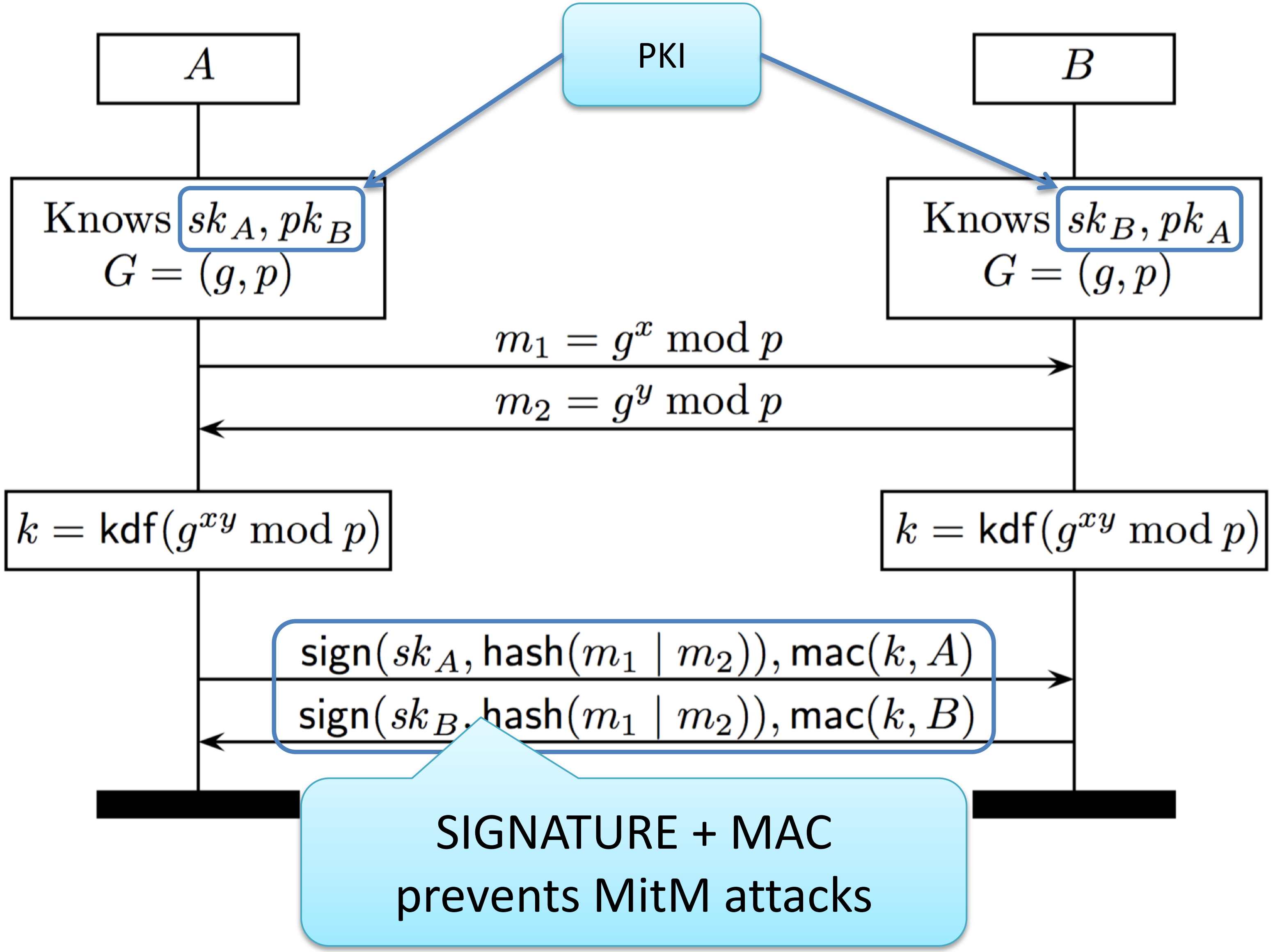


Classic man-in-the-middle attack



Active Network Attacker
or Malicious Peer

SIGMA: authenticated Diffie-Hellman



Crypto Proof: Diffie-Hellman assumption

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

$0 < x,y < p$

g fixed (e.g. 2)

Large Prime

(e.g. $2^{255} - 19$)

PROTOCOL SECURITY RELIES ON DH HARDNESS ASSUMPTION:

An attacker who does not know x or y cannot compute g^{xy} mod p

Crypto Weakness: small prime groups

If the prime p is too small,
an attacker can compute the discrete log:

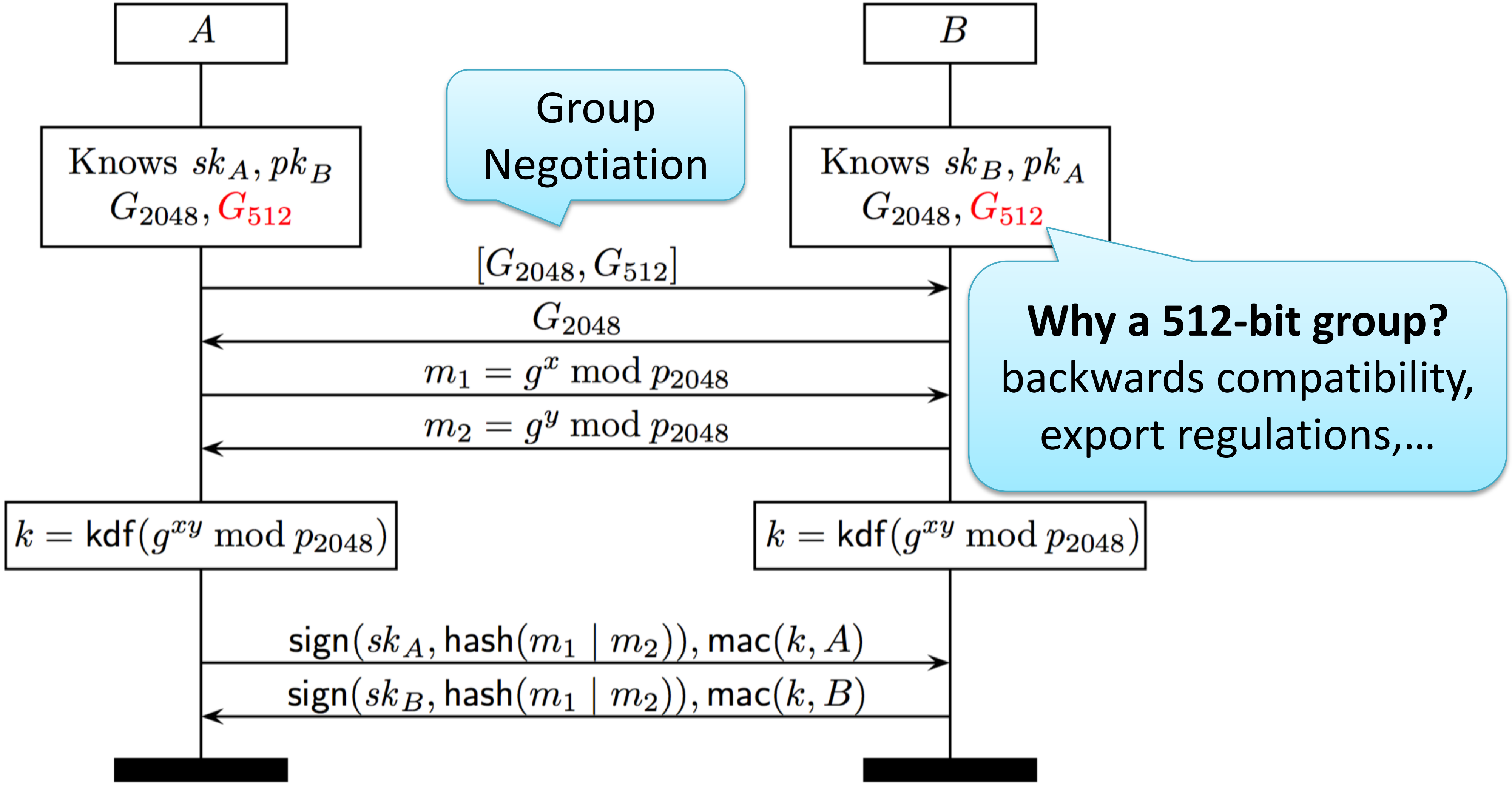
$$y = \log(g^y \ mod\ p)$$

and hence compute the session key: $g^{xy} \mod p$

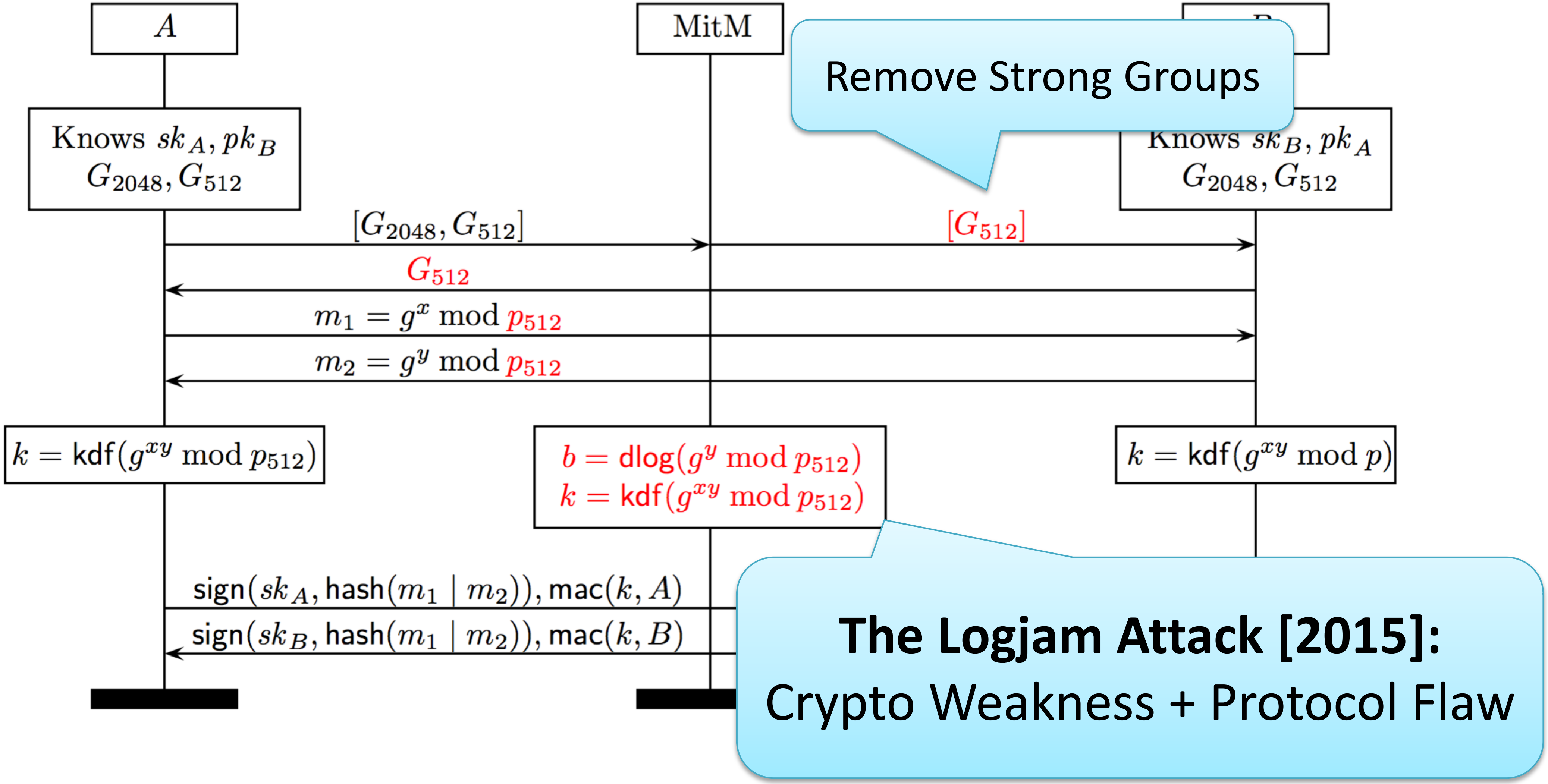
Current discrete log computation records:

- [Joux et al. 2005] **431-bit prime**
- [Kleinjung et al. 2007] **530-bit prime**
- [Bouvier et al. 2014] **596-bit prime**
- [Kleinjung et al. 2017] **768-bit prime**

Negotiating the strongest available group



Protocol Flaw: group downgrade attack



Implementation Bugs

Negotiation flaws re-enable disabled ciphersuites

- e.g. FREAK, Logjam, DROWN

Functional correctness bugs in DH computation

- e.g. Carry propagation errors in Curve25519

Side-channel attacks on signature algorithm

- e.g. Timing attacks on ECDSA/RSA

Identifying and preventing such attacks

Prove cryptographic security of the protocol core

- Hire a cryptographer to do the proof (~ months)
- Use **mechanized provers**: EasyCrypt, CryptoVerif, …

Analyze full protocol for MitM attacks like downgrades

- Model and verify full protocol automatically (~ weeks)
- Use **protocol verification tools**: ProVerif, Tamarin,…

Verify implementation to find coding bugs

- Insert verification into development workflow (~ years)
- Use software **verification tools**: hacspec, F*, Frama-C, …

Identifying and preventing such attacks

Prove cryptographic security of the protocol core

- Hire a cryptographer to do the proof (~ months)
- Use **mechanized provers**: EasyCrypt, CryptoVerif, …

Analyze full protocol for MitM attacks like downgrades

- Model and verify full protocol automatically (~ weeks)
- Use **protocol verification tools**: ProVerif, Tamarin,…

Verify implementation to find coding bugs

- Insert verification into development workflow (~ years)
- Use software **verification tools**: hacspec, F*, Frama-C, …

Designing protocols to be verifiable

1. Precisely define the threat model and security goals

2. Use standard, well-understood crypto constructions

3. Break protocol into composable sub-protocols

4. Remove or limit key reuse between different modes

5. Specify state machines and necessary data structures

The TLS 1.3 experiment

Protocol re-designed to enable easier cryptographic analysis

- Sometimes security won over performance, sometimes not

Formal security analysis hand-in-hand with standardization

- **Cryptographic proofs (of drafts 5,9,10)**
  [Dowling et al. CCS'15, Krawczyk et al. Euro S&P'16, Li et al. S&P'16]
- **Mechanized cryptographic proofs (of draft 18)**
  [Bhargavan et al. S&P'17]
- **Automated symbolic protocol analysis (of draft 10, 18, 20)**
  [Cremers et al. Oakland'16 and CCS'17, Bhargavan et al. S&P'17]
- **Verified implementations (of draft 18)**
  [Bhargavan et al. S&P'17 and S&P'17]

Some modes of TLS 1.2 are broken.

All modes of TLS 1.3 are provably secure.

Can a man-in-the-middle downgrade TLS 1.3 connections to use broken TLS 1.2 modes?

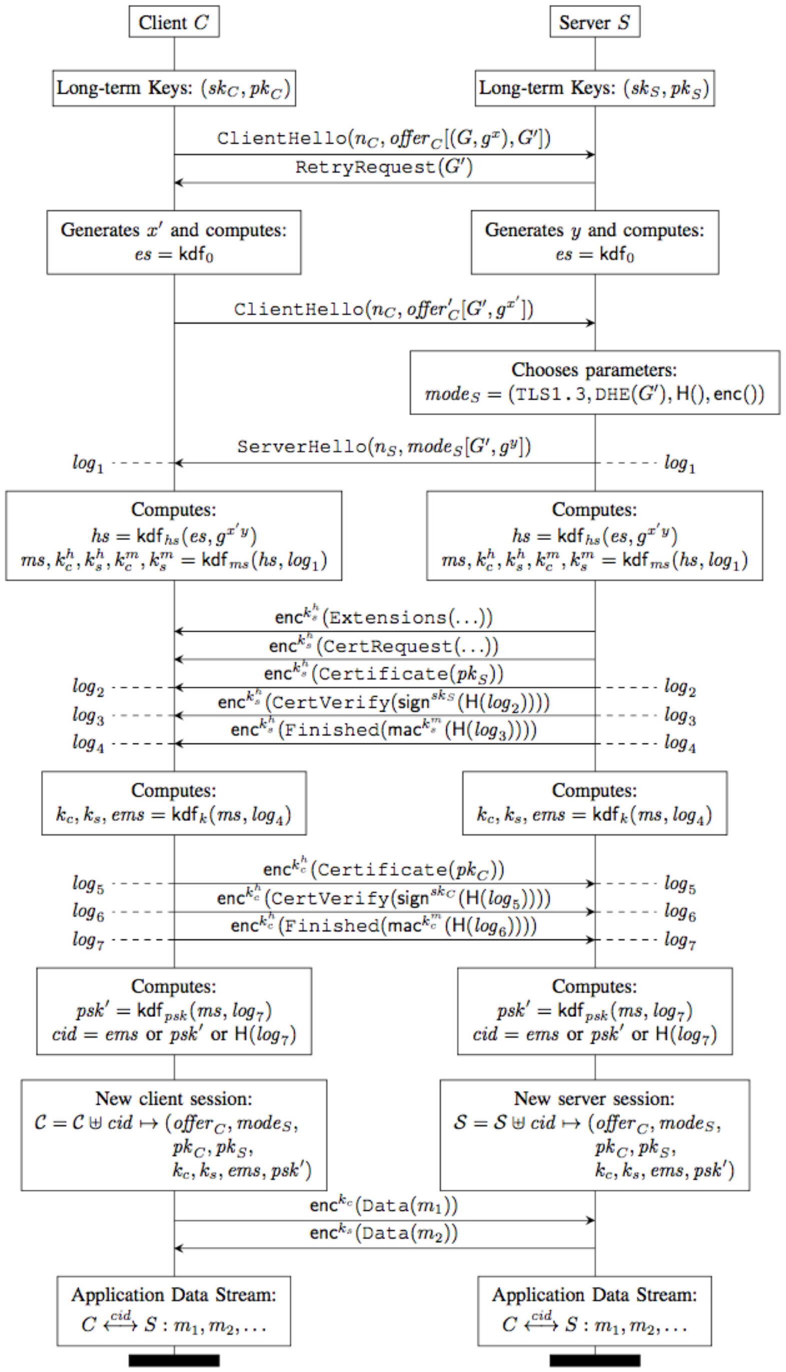
Modeling TLS 1.3 in ProVerif

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:

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

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

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

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 1.2 + TLS 1.3

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

Not just TLS: Analyses for Other Protocols

Attacks and proofs for OAuth 2.0

- Symbolic analysis [Fett, Kuesters, Schmitz, CCS'16], ….

Attacks and proofs for ACME

- ProVerif [Bhargavan, Delignat-Lavaud, Kobeissi, FC'17]

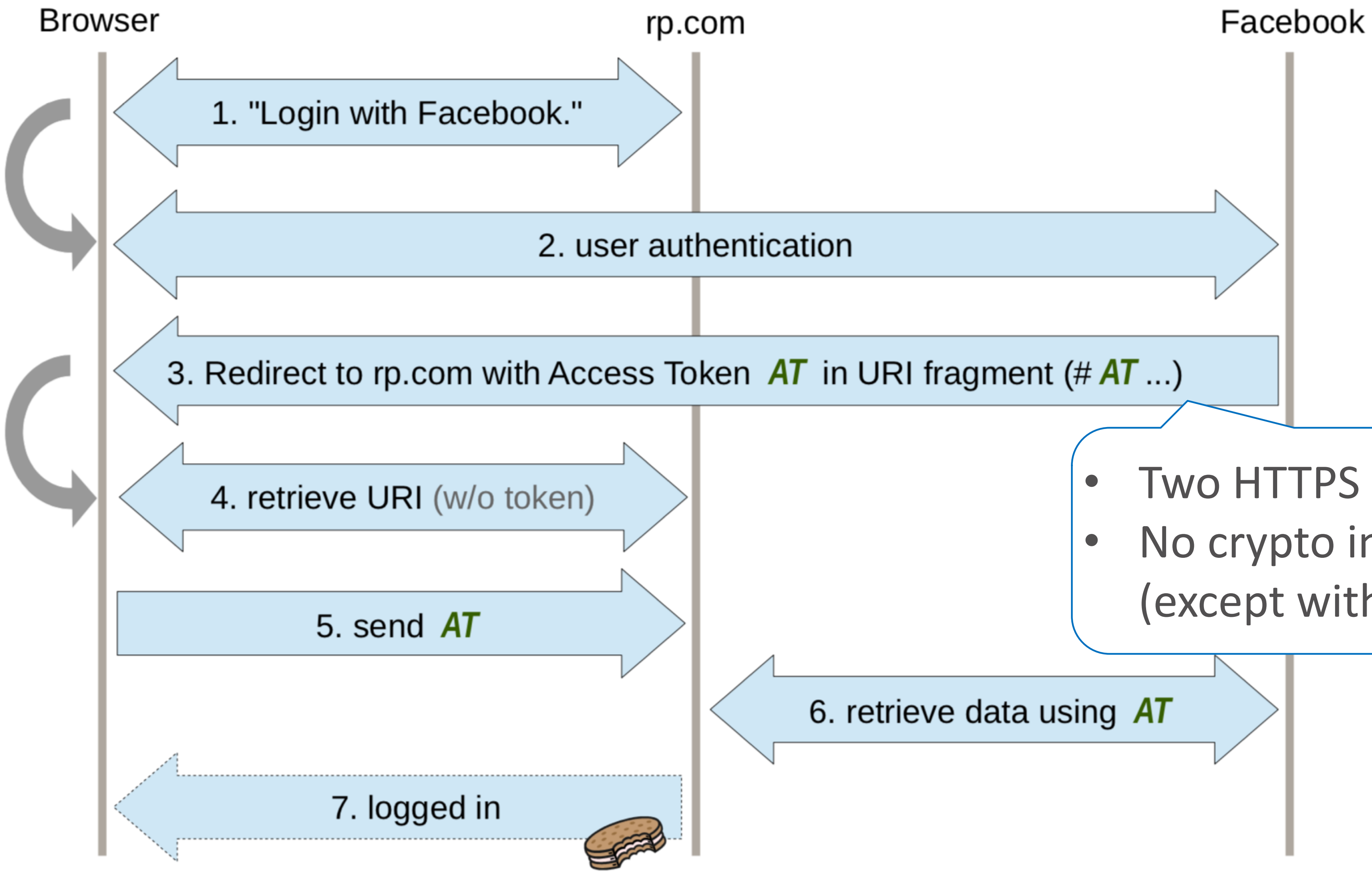
Attacks on 5G AKA

- Tamarin [Dehnel-Wild, Cremers, 2017]

NEW: A call for design and analysis of MLS

- Tamarin [Cohn-Gordon et al], ProVerif, CryptoVerif, …

OAuth 2.0 Web Authorization Protocol



- Two HTTPS redirects
- No crypto in protocol (except within AT)

What is the Web threat model?

OAuth 2.0 needs to protect against web attackers

- Significantly more powerful than symbolic network attackers
- OAuth 2.0 RFC: 76 pages
- OAuth 2.0 security considerations: 71 pages

Analysis needs a new threat model for Web attackers

- Detailed browser model
- Hand proofs of security (automation ongoing)

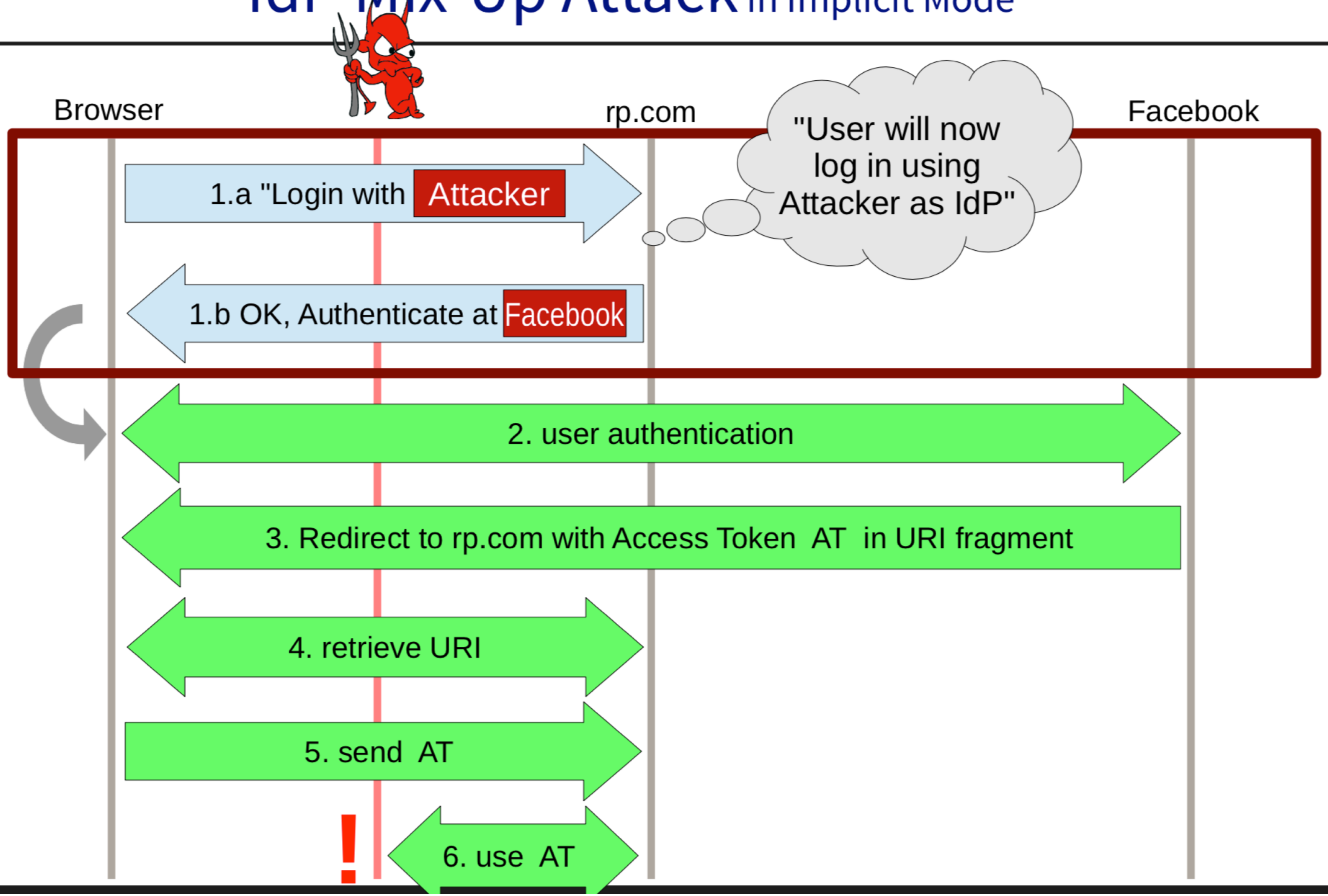
A Comprehensive Formal Security Analysis of OAuth 2.0*

Daniel Fett
University of Trier, Germany
fett@uni-trier.de

Ralf Küsters
University of Trier, Germany
kuesters@uni-trier.de

Guido Schmitz
University of Trier, Germany
schmitzg@uni-trier.de

IdP Mix-Up Attack in Implicit Mode



Conclusion

Formal security analyses can find protocol flaws, and provide strong cryptographic security guarantees

- Requires some expertise, tools are improving
- Designing protocols to ease analysis provides good trade-offs

The first step is to write a formal specification

- Threat model, security goals, protocol model
- Often, modeling the protocol already exposes bugs
- Maybe you can also include the formal spec in the RFC?
- **Do it:** hacspec, ProVerif, Tamarin, EasyCrypt, CryptoVerif,…

Questions?

- hacspec: https://github.com/HACS-workshop/hacspec
- ProVerif: http://proverif.inria.fr
- Tamarin: https://tamarin-prover.github.io/
- Cryptoverif: http://cryptoverif.inria.fr
- EasyCrypt: https://www.easycrypt.info