Some Simple Protocols: Attacks and Proofs
Designing an RPC Protocol

**Goal:** protect the secrecy and authenticity of *req* and *resp*
Symmetric-Key RPC

Is this secure?
Reflection Attack on Symmetric-Key RPC

Initiator $I$

Initially Knows: $k_{ir}$

Exchange: $I \leftrightarrow R : req; req$

Attacker $O$

Initially Knows: $k_{ir}$

Attacker reflects request back to $I$

Responder $R$
Fixing Symmetric-Key RPC

Fix: Add message tags (or use different keys)
Public-Key Encrypt-then-Sign RPC

Initiator $I$

Initially Knows:
\[ sk_i, pk_r \]

Exchange:
\[ I \leftrightarrow R : req; resp \]

Responder $R$

Initially Knows:
\[ sk_r, pk_i \]

Exchange:
\[ I \leftrightarrow R : req; resp \]

\[ \text{sign}(sk_i, \text{penc}(pk_r, 0\|req)) \]

\[ \text{sign}(sk_r, \text{penc}(pk_i, 1\|resp)) \]

Is this secure?
Re-signing Attack on Encrypt-then-Sign

Initiator \( I \)
- Initially Knows: \( sk_i, pk_r \)
- \( \text{sign}(sk_i, \text{penc}(pk_r, 0||req)) \)

Attacker \( O \)
- Initially Knows: \( pk_i, pk_r \)
- \( \text{sign}(sk_o, \text{penc}(pk_r, 0||req)) \)
- \( \text{sign}(sk_r, \text{penc}(pk_o, 1||resp)) \)

Responder \( R \)
- Initially Knows: \( sk_r, pk_i \)

Knows Secret: \( resp \)

Attacker replaces I’s signature with his own, to obtain the secret \( resp \)
Fixing Encrypt-then-Sign

Fix: Add signer’s identity to inner encryption (secure composition)
Using Sign-then-Encrypt

Initiator $I$

Initially Knows:
\[ sk_i, pk_r \]

Exchange:
\[ I \leftrightarrow R : req; resp \]

Responder $R$

Initially Knows:
\[ sk_r, pk_i \]

\[ \text{penc}(pk_r, \text{sign}(sk_i, 0||req)) \]

\[ \text{penc}(pk_i, \text{sign}(sk_r, 1||resp)) \]

Is this secure?
Identity Misbinding Attack on Sign-then-Encrypt

**Attacker** acting as a valid responder for I, re-encrypts request to R, causing an *identity mis-binding attack*.
Response Correlation Attack on RPC

Attacker reorders responses. Works on all our RPC protocols.
Fixing Sign-then-Encrypt

**Initiator** $I$

**Initially Knows:**

$sk_i, pk_r$

**Exchange:**

$I \leftrightarrow R : req; resp$

**Responder** $R$

**Initially Knows:**

$sk_r, pk_i$

**Exchange:**

$I \leftrightarrow R : req; resp$

$penc(pk_r, sign(sk_i, 0||pk_r||req))$

$penc(pk_i, sign(sk_r, 1||pk_i||req||resp))$

**Fix:** add recipient identity and request to inner sig *(secure composition)*
Authenticated Key Exchange: STS

\[ A \xrightarrow{g^x} B \]

\[ E \xleftarrow{g^y, B, \text{SIG}_B(g^x, g^y)} E, \text{SIG}_E(g^y, g^x) \]

Session Key = KDF(g^{xy})
Authenticated Key Exchange: ISO

\[ A, g^x \]

\[ B, g^y, \text{SIG}_B(g^x, g^y, A) \]

\[ \text{SIG}_A(g^y, g^x, B) \]

Session Key = KDF(g^{xy})
Authenticated Key Exchange: ENC-STS

\[ A, \{ \text{SIG}_A(g^y, g^x) \}^{K_s} \]

\[ g^x, g^y, B, \{ \text{SIG}_B(g^x, g^y) \}^{K_s} \]

\[ A \]

Session Key \( K_s = \text{KDF}(g^{xy}) \)
Secure Messaging:
towards verified standards and high-assurance implementations

Karthik Bhargavan
http://prosecco.inria.fr

+ joint work many many many many others
Secure Messaging Today

**Common Goal:** End-to-End Security

**Relatively New Cryptographic Protocols**
- Signal, iMessage, Telegram MTProto, ...
- Use modern cryptographic constructions
- Subtly different security guarantees
- Implemented on desktops and phones

**Are they safe to use?**
- Often relied upon by high-risk populations
Messaging Server

Compromised Server

Untrusted Network

Compromised Devices

Alice

Bob

m
Basic Secure Messaging Goals

Confidentiality: Is $m$ kept secret from the adversary?
Authenticity: Can Bob be sure that Alice sent $m$?
Solution: End-to-End encryption from Alice to Bob

Asynchronous Secure Channel Protocol
• Alice and Bob may not be online at the same time
• Corrupt server or network attacker may steal/tamper with messages
Advanced Secure Messaging Goals

Forward Secrecy
Suppose the attacker records everything sent on the network. Alice and Bob exchange and delete $m$, then their phones are stolen. Attacker still cannot obtain $m$.

Post-Compromise Secrecy
Suppose the attacker copies Alice’s phone and returns it. Alice then sends $m$ to Bob. Attacker still cannot obtain $m$. 
Example: OTR [2004-]

- Off-The-Record protocol

- Goals:
  - Encryption (Secrecy)
  - Authentication
  - Deniability
  - Perfect Forward Secrecy

- Used in Pidgin, Adium, etc.
OTR Key Exchange

- Version 1 used signed DH → an identity misbinding attack
- Versions 2 & 3 use SIGMA
  - Anonymous DH followed by encrypted authentication phase
  - Commitment scheme to prevent brute-force attacks on DH key
- Requires both Alice & Bob to be online for key exchange

Figure 1: OTR authenticated key exchange protocol
OTR Data Protocol

- Authenticated encryption
  - Encrypt-then-MAC
  - AES-CTR, HMAC-SHA256
- Rekeying after (at least) one message sent in each direction
  - Fresh ephemerals per message
  - Rekey when both sides have acknowledged new ephemerals
- Deniability: publish old MAC keys

![Diagram of OTR data exchange protocol](image-url)
OTR Defects

- Cannot be used if peer is offline
- Publishing MAC keys too early leads to forgery attacks
  - Must release MAC keys after two generations of key refresh
- Slow and cumbersome
  - Diffie-Hellman (1536-bit group)
  - DSA Signatures
- Well-regarded but unpopular
Example: Telegram

Telegram Secret Chats [2013-]
• 100M active users

MTProto Protocol (2.0)
• Home-grown design
• Old-ish Crypto primitives: SHA-1, SHA-256, AES-IGE
• Unusual AE construction: Hash-then-Encrypt

Is MTProto Secure?
Interlude:

Diffie-Hellman Key Exchange
Anonymous Diffie-Hellman key exchange

\[ g^x \mod p \]

\[ g^y \mod p \]

\[ k = \text{kdf}(g^{xy} \mod p) \]
Classic man-in-the-middle attack

Active Network Attacker or Malicious Peer
SIGMA: authenticated Diffie-Hellman

**SIGMA**

**Authenticated Diffie-Hellman with PKI**

- User A knows $sk_A, pk_B$ and $G = (g, p)$
- User B knows $sk_B, pk_A$ and $G = (g, p)$

1. User A computes $m_1 = g^x \mod p$
2. User B computes $m_2 = g^y \mod p$
3. $k = \text{kdf}(g^{xy} \mod p)$
4. User A signs $\text{hash}(m_1 | m_2)$ with $sk_A$ and MACs $k$ and $A$
5. User B signs $\text{hash}(m_1 | m_2)$ with $sk_B$ and MACs $k$ and $B$

**Signature + MAC prevents MitM attacks**
Security Proof: Diffie-Hellman assumption

PROTOCOL SECURITY RELIES ON DH HARDNESS ASSUMPTION:
An attacker who does not know $x$ or $y$ cannot compute $g^{xy} \mod p$
Crypto Weakness: bad primes

If the prime $p$ is too small or if $p$ is not a prime, 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
End Interlude:
Back to Telegram
Telegram MTProto Key Exchange

1. Server chooses a Diffie-Hellman (DH) group
2. Alice, Bob do an anonymous DH exchange
3. Alice, Bob authenticated by comparing identicons
   1. Compute Hash($g^{ab}$)
   2. Truncate to 128/160 bits
   3. Convert to identicon image
   4. Alice and Bob compare identicons when they meet
1. Malicious server can choose bad Diffie-Hellman group
   • **BUG:** Some clients forget to check group for primality
   • Server can hijack connection

2. Network attacker can exploit small subgroups
   • **BUG:** Some clients forget to validate public keys
   • Attacker sends $g^a = g^b = 1$
   • $g^{ab} = 1$, authentication breaks
Other Attacks on MTProto Crypto

- Brute-force ($2^{64} / 2^{80}$) collision attack on identicons
  - hash truncation weakens authentication

- Message encryption scheme not secure against tampering
  - chosen ciphertext attack breaks message confidentiality
Better Example: Signal

Protocol used by WhatsApp, Skype, Facebook Messenger, Signal, ...

- $\gg 1B$ Users

Axolotl ratchet $\rightarrow$ TextSecure v2 $\rightarrow$ TextSecure v3 $\rightarrow$ Signal Protocol

- First asynchronous messaging protocol
- Innovative state-of-the-art design
- Modern Crypto: X25519, AES-256, HMAC
- Standard AE construction: Encrypt-then-MAC
Recall: Secure Messaging Goals

Confidentiality: Is $m$ kept secret from the adversary?

Authenticity: Can Bob be sure that Alice sent $m$?

Forward Secrecy: Confidentiality for messages before compromise

Post-Compromise Security: Confidentiality after compromise
Signal Protocol

B → (g^b, g^s, g^o)

Four Diffie-Hellman Key Exchanges

- \( g^a \) (actively) authenticates Alice
- \( g^b \) (passively) authenticates Bob
- \( g^x, g^o, g^s \) provide freshness

Diffie-Hellman Ratchet

- Adds fresh key material to session
- Provides post-compromise security

Hash (KDF) Ratchet

- Updates session key
- Provides forward secrecy

\( m_3 + \text{keys} \)

"it's at Oakland"
Signal Protocol

A fairly complicated cryptographic protocol

• Composition of X3DH (initial key exchange)
  + DH Ratchet (post-compromise security)
  + Hash Ratchet (forward security)
  + Message Authenticated Encryption

• Confidentiality and authenticity guarantee for each message depends upon a long chain of prior messages

How can we be sure that this protocol is secure?

• How do we ensure that its implementation is bug-free?
Formalizing Signal

Initiator I

Prior Knowledge:

\[(i, g^i)\]

Initiate\((i, g^r, g^s[, g^o]) \rightarrow (rk_0):\)

generate \((e, g^e)\)

\[dh_0 = 0xFF \mid g^{si} \mid g^{re} \mid g^{se} \mid g^{oe}\]

\[rk_0 = HKDF(dh_0, 0x00^{32}, "WhisperText")\]

Responder R

Prior Knowledge:

\[(r, g^r), (s, g^s[, (o, g^o)])\]
Signal Ratchet: Key Refresh

- Every message has a new ephemeral value $g^{x'}$

- New keys derived from old keys + $g^{yx'}$ (new x, old y)
  - Each ephemeral used twice
  - Old MAC keys may be published after new $g^{y'}$ received

- Key separation via multiple HKDFs
Signal: 0-RTT Weaknesses

• Binding keys to wrong identity
  – B may have used C’sephemerals
    (Unknown key share attack)
• Attacker can exhaust B’s one-time keys
• First message can be replayed
  – No replay cache
• Key compromise impersonation
  – If both a and s are compromised
• We find all attacks in ProVerif
Formally Analyzing Signal

**Goals** | **Messages** | **Parties** | **Roles** | **Time**
--- | --- | --- | --- | ---
Secrecy | 1 | A, B | One | 00h.04m.07s.
Secrecy | 1 | A, B | Two | 00h.11m.17s.
Indist. | 1 | A, B | One | 02h.06m.15s.
Authen. | 1 | A, B, M | One | 00h.58m.19s.
Authen. | 1 | A, B, M | Two | 29h.17m.39s.
Fo. Se. | 1 | A, B | One | 00h.04m.14s.
KCI | 1 | A, B | One | 00h.19m.20s.

**Figure 3:** Verification times for SP ProVerif models.

**Goals** | **Parties** | **Running Time**
--- | --- | ---
Forward Secrecy | A, B, M | 3 min. 58 sec.
Forward Secrecy | A, M | 7 min. 04 sec.
KCI | A, B, M | 3 min. 15 sec.
Others | A, B, M | 4 min. 15 sec.
Others | A, M | 3 min. 35 sec.

**Figure 4:** Verification times for SP CryptoVerif models, without anti-replay countermeasure. The runtimes with the anti-replay countermeasure are of the same order of magnitude.
Are we done?

Signal is a provably secure two-party messaging protocol
• Some weaknesses in the initial message: replays, KCI attack
• Tradeoff between using signatures and just Diffie-Hellman

What about group messaging? [Next Part of Talk]
• Most of us use WhatsApp groups
• Do they provide the same level of security as two-party Signal?

What about implementation bugs? [Last two lectures]
• Dozens of implementations of Signal and its crypto
• Can we prove that this code is correct? side-channel free?
Messaging Layer Security: A new standard for secure group messaging
Group Messaging Server

Compromised Server

Untrusted Network

Alice

Charlie (malicious insider)

Debbie

Eve (malicious outsider)

Bob
Adding a New Group Member
Removing a Group Member

Group Messaging Server

Remove Charlie

Alice

Charlie

Debbie

Eve

Bob
Basic Group Messaging Goals

Confidentiality: Is $m$ kept secret from non members?

Authenticity: Can Bob be sure that a member Alice sent $m$?

Group Agreement: Do Bob and Alice agree on group membership?

Add Security: Debbie cannot read messages from before she joined

Remove Security: Charlie cannot read messages after he left
Advanced Group Messaging Goals

Group Forward Secrecy
If all members delete \( m \), then attacker cannot obtain \( m \), even if he compromises all their phones.

Group Post-Compromise Secrecy
If the attacker copies Alice’s phone data and returns it, then if Alice updates its keys before sending or receiving \( m \), then the attacker cannot obtain \( m \).

We know how to do this for 2-party messaging (Signal)
How do we design a protocol that works for large dynamic groups?
Signal Sender Keys

Use 1-1 Signal Channels to send Group Messaging Keys
Requires $n^2$ channels to establish a group of $n$ members

Authenticate each user separately
No group agreement, since no notion of global group

Add, Remove: send a message to all members
$n$ messages for each group change (typically $n$ DH operations)

Does not scale to large groups
Corporate groups can have thousands of members
IETF Messaging Layer Security

A new working group to design efficient, provably secure group messaging protocol
Should efficiently handle groups with 10K+ members

Assume a trusted authentication service
Each group member has a verifiable authentication credential

Under active development [2018-]
Draft 1: Asynchronous Ratcheting Trees
Draft 2: TreeKEM
Draft 3-7: TreeKEM with Blanking
Chained mKEM: linear scaling

Encrypt group key to all members in encryptions

At each group change, create new group key in encryptions

Satisfies all our goals, but scales linearly with group size
Tree-Based Key Establishment
[ART, Cohn-Gordon et al CCS 2017]

MLS draft 0-1 asynchronous ratcheting trees

A tree of subgroups
n leaves for group members
n nodes for subgroups
each subgroup has its own key

A subgroup key is known only to its members
encrypted for left+right child

Members = \{a, b, c, d, e\}  (max_size = 8)
Tree-Based Key Establishment
[ART, Cohn-Gordon et al CCS 2017]

Creating a Group
create and encrypt subgroup keys for all nodes in tree
$n$ encryptions at sender
$log(n)$ decryptions at receiver

Modifying a Leaf
create and encrypt subgroup keys for path from leaf to root
$log(n)$ encryptions at sender
$log(n)$ decryptions at receiver

Members = \{a, b, c, d, e\} (max_size = 8)
TreeKEM: Reducing Receiver Cost

Subgroup keys are derived from one child, and encrypted for other asymmetry between children yields a faster protocol.

Modifying a Leaf
hash and encrypt subgroup keys for path from leaf to root
log(n) encryptions at sender
1 dec + log(n) hashes at receiver
Double Join Attack on ART and TreeKEM

Malicious Insider Alice with key $ek_a$
Double Join Attack on ART and TreeKEM

Malicious Insider Alice poisons keys at 4, 45, and 47

c3 removes a0
Double Join Attack on ART and TreeKEM

Malicious Insider Alice has been removed but it still knows the group key, violating Remove Security
## Analyzing MLS Candidates

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Create</th>
<th>Add</th>
<th>Remove</th>
<th>Update</th>
<th>Group Agreement</th>
<th>Update PPCS</th>
<th>Remove PACS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Send</td>
<td>Recv</td>
<td>Send</td>
<td>Recv</td>
<td>Send</td>
<td>Recv</td>
<td>Send</td>
</tr>
<tr>
<td>Sender Keys [18]</td>
<td>$N^2$</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chained mKEM$^+$</td>
<td>$N$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$N$</td>
<td>1</td>
<td>$N$</td>
</tr>
<tr>
<td>2-KEM Trees$^+$</td>
<td>$N$ log($N$)</td>
<td>$log(N)$</td>
<td>$log(N)$</td>
<td>$log(N)$</td>
<td>$log(N)$</td>
<td>$log(N)$</td>
<td>$log(N)$</td>
</tr>
<tr>
<td>ART [7]</td>
<td>$N$ log($N$)</td>
<td>$log(N)$</td>
<td>$log(N)$</td>
<td>$log(N)$</td>
<td>-</td>
<td>-</td>
<td>$log(N)$</td>
</tr>
<tr>
<td>TreeKEM$^+$</td>
<td>$N$ log($N$)</td>
<td>$log(N)$</td>
<td>1</td>
<td>1</td>
<td>$log(N)$</td>
<td>1</td>
<td>$log(N)$</td>
</tr>
<tr>
<td>TreeKEM$_B^+$</td>
<td>$N$ 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$log(N)$..$N$</td>
<td>1</td>
<td>$log(N)$..$N$</td>
</tr>
<tr>
<td>TreeKEM$_{B+S}^+$</td>
<td>$N$ 1</td>
<td>1</td>
<td>1</td>
<td>$N$</td>
<td>$log(N)$..$N$</td>
<td>1</td>
<td>$log(N)$..$N$</td>
</tr>
</tbody>
</table>

### Formally analyzing MLS Drafts 0-7

Many protocol have been proposed, some broken, then fixed
We want to **find attacks**, and **prove security**

**Automated analysis is hard**: unbounded members, recursive structures
We use **F***: machine-checked proofs by induction for any size group
Back to TLS: Finding Practical Attacks and Building Comprehensive Proofs
Recall: Many recent attacks on TLS

- 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]
The TLS 1.3 experiment

Multi-year effort to redesign IETF Transport Layer Security
• 4 years, 28 drafts, 12 IETF meetings

Major contributions from academic security researchers
• **Cryptographic proofs (of drafts 5,9,10)**
• **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]
A Completely New Design for TLS

1. Disable all legacy crypto, and mandate new standards
   - No RSA, CBC, MD5, weak DH
   - New downgrade protection mechanism

2. Faster handshakes with 0-RTT, 0.5-RTT, 1-RTT
   - 0-RTT/0.5-RTT has lower security, 1-RTT has full security

3. Cleaner composition between TLS and its environment
   - PSK-ECDHE both for external PSKs and resumed sessions
   - Post-handshake authentication, exporter keys for applications
TLS 1.3 Core Key Exchange

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

+ 0-RTT + TLS 1.2
Is TLS 1.3 provably secure?
The Three Lives of TLS 1.3

**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+]
- **RFC 8846** (2018): Ready for deployment?
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]
TLS Security Goals

CONFIDENTIALITY
Nobody other than the bank can read what I type

AUTHENTICATION
Nobody other than me can access my account page

TLS Protocol
Classic Secure Channel.
Provable Security for Secure Channel Protocols

**Protocol:** Suppose client A sends a message M to server B (over TLS)
- A and B are *honest*: their keys are *secret*
- **Symbolic Model:** A, B are high-level programs over *terms*
- **Computational Model:** A, B are probabilistic polynomial Turing machines (PPTM) over *bitstrings*

**Threats:** Suppose the attacker fully controls the network
- All other clients, servers are *dishonest*: their keys are *public*
- **Symbolic/Computational Model:** Attacker as program/PPTM

**Security Goals:** The attacker does not obtain M
- Even if M has only 1 bit of entropy
- Even if all other TLS connections are fully broken
Proving Security for TLS 1.3

**Protocol:** Write a formal model of TLS 1.3 clients and servers
- *Symbolic:* Applied Pi Calculus (*ProVerif*), Rewriting Rules (*Tamarin*)
- *Computational:* Applied Pi Calculus (*CryptoVerif*), While Programs (*EasyCrypt*), Functional Programs (*F* *)*

**Threats:** Write precise cryptographic assumptions
- *Symbolic:* equational theory modeling crypto using rewrite rules
- *Computational:* probabilistic game-based assumptions
- *Attacker* is any process/program that obeys these assumptions

**Security Goals:** Ask security queries of the protocol + attacker
- *Symbolic:* Is a bad state reachable? Is secret M leaked?
- *Computational:* What is the probability of attack?
TLS 1.3 Symbolic Model
- 1-RTT Server Process
- Written in applied pi calculus
- Verifiable using ProVerif

Full model of TLS 1.3+1.2
- Protocol: 500 lines
- Assumptions: 400 lines
- Security goals: 200 lines
Proving Downgrade Resilience for TLS 1.3

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

Client $I$

$\text{CH}(n_I, \max I, [a_1, \ldots, a_n], [(G_1, g^{x_1})])$

$\text{Retry}(G_2)$

$\text{log}_1$

Server $R$

$\text{log}_1$

$\text{SH}(n_R, v, a_R, (G_2, g^y))$

$\text{CH}(n_I, \max I, [a_1, \ldots, a_n], [(G_1, g^{x_1}), (G_2, g^{x_2})])$
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:

\[ \text{hkd-extract}(k, s) = \text{HMAC-H}^b(s) \]
\[ \text{hkd-expand-label}_1(s, l, h) = \text{HMAC-H}^b(\text{len}_1 || "TLS 1.3" || l || h || 0x01) \]
\[ \text{derive-secret}(s, l, m) = \text{hkd-expand-label}_2(s, l, H(m)) \]

1-RTT Key Schedule:

\[ \text{kdf}_0 = \text{hkd-extract}(\text{len}_1, 0) \]
\[ \text{kdf}_h(es, e) = \text{hkd-extract}(es, e) \]

\[ \text{kdf}_h(s, \text{log}_1) = ms, k_c, k_s, k_m, k_a \text{ where} \]
\[ ms = \text{hkd-extract}(h, 0) \]
\[ hts_c = \text{derive-secret}(hs, hts_c, \text{log}_1) \]
\[ hts_s = \text{derive-secret}(hs, hts_s, \text{log}_1) \]
\[ k_c^b = \text{hkd-expand-label}(hts_c, key, \text{sym}) \]
\[ k_m^b = \text{hkd-expand-label}(hts_c, finished, \text{sym}) \]
\[ k_h^b = \text{hkd-expand-label}(hts_s, key, \text{sym}) \]
\[ k_a^m = \text{hkd-expand-label}(hts_s, finished, \text{sym}) \]

\[ \text{kdf}(ms, \text{log}_1) = k_c, k_s, \text{ems} \text{ where} \]
\[ ats_c = \text{derive-secret}(ms, ats_c, \text{log}_4) \]
\[ ats_s = \text{derive-secret}(ms, ats_s, \text{log}_4) \]
\[ ems = \text{derive-secret}(ms, \text{ems}, \text{log}_4) \]
\[ k_c = \text{hkd-expect-label}(ats_c, key, \text{sym}) \]
\[ k_s = \text{hkd-expand-label}(ats_s, key, \text{sym}) \]

PSK-based Key Schedule:

\[ \text{kdf}(psk) = es, k_b^b \text{ where} \]
\[ es = \text{hkd-extract}(1^b || psk) \]
\[ k^b = \text{derive-secret}(es, psk, \text{sym}) \]

\[ \text{kdf}_{\text{RTT}}(es, \text{log}_1) = k_c \text{ where} \]
\[ ets_c = \text{derive-secret}(es, ets_c, \text{log}_1) \]
\[ k_c = \text{hkd-expand-label}(ets_c, key, \text{sym}) \]
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,sp,TLS13);
   event ServerChoosesKEX(cr,sp,TLS13,DHE_13(g,gy));
   event ServerChoosesAE(cr,sp,TLS13,1);
   event ServerChoosesHash(cr,sp,TLS13,1);

   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:psk) =
  let es = hkdf_extract(zero,psk2b(psk)) in
  let kb = derive_secret(es,tl13_resumption_psk_binder_key,zero) in
  (es,b2mk(kb)).

letfun kdf_k0(es:bitstring,log:bitstring) =
  let atsc0 = derive_secret(es,tl13_client_early_traffic_secret,log) in
  let kc0 = hkdfexpand_label(atsc0,tl13_key,zero) in
  let ems0 = derive_secret(es,tl13_early_exporter_master_secret,log) in
  (b2ae(kc0),ems0).

letfun kdf_hs(es:bitstring,e:bitstring) =
  let extra = derive_secret(es,tl13_derived,hash(StrongHash,zero)) in
  hkdf Extract(extra,e).

letfun kdf_ms(es:bitstring,log:bitstring) =
  let extra = derive_secret(es,tl13_derived,hash(StrongHash,zero)) in
  let ms = hkdf_extract(hs,zero) in
  let htsc = derive_secret(hs,tl13_client_handshake_traffic_secret,log) in
  let htss = derive_secret(hs,tl13_server_handshake_traffic_secret,log) in
  let kch = hkdfexpand_label(htsc,tl13_key,zero) in
  let kcm = hkdfexpand_label(htsc,tl13_finished,zero) in
  let ksh = hkdfexpand_label(htss,tl13_key,zero) in
  let ksm = hkdfexpand_label(htss,tl13_finished,zero) in
  (ms,b2ae(kch),b2ae(ksh),b2mk(kcm),b2mk(ksm)).
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

**Downgradeable:** 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 \(\text{msg}(\text{conn}, S)\) sent from client C to server S

\[ \text{query not attacker}(\text{msg}(\text{conn}, S)) \]
Refining Security Queries

• **QUERY:** Is $\text{msg(\text{conn}, S)}$ secret?

  query not attacker($\text{msg(\text{conn}, S)}$)

• **FALSE:** ProVerif finds a counterexample if S’s private key is compromised
Refining Security Queries

• QUERY: Is \text{msg(conn,S)} secret as long as \text{S} is uncompromised?

\text{query attacker(msg(conn,S))} \implies \text{event(WeakOrCompromisedKey(S))}

• FALSE: ProVerif finds a counterexample if the AE algorithm is weak
Refining Security Queries

• **QUERY:** Is \( \text{msg} \text{(conn,S)} \) secret as long as \( S \) is uncompromised and only strong AE algorithms are used?

\[
\text{query attacker}(\text{msg}(\text{conn},S)) \implies \text{event(WeakOrCompromisedKey(S)) || event(ServerChoosesAE(conn,WeakAE))}
\]

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

*Logjam Attack on TLS 1.3 + 1.2*
Refining Security Queries

• Strongest secrecy query that can be proved in our model

```
query attacker(msg(conn,S)) ==>
  event(WeakOr\textbf{CompromisedKey}(S)) ||
  event(ServerChoosesAE(conn,S,WeakAE)) ||
  event(ServerChoosesKEX(conn,S,WeakDH)) ||
  event(ServerChoosesKEX(conn’,S,WeakRSAEncryption)) ||
  event(ServerChoosesHash(conn’,S,WeakHash))
```

• **TRUE**: ProVerif finds no counterexample
Symbolic Security Theorem for 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 yields preconditions for downgrade resilience
• Identifies all weak TLS 1.2 algorithms that can harm TLS 1.3 security
• The combined TLS 1.2+1.3 model is too large to reason about manually.
• Automated formal verification tools make this kind of analysis possible.
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
(EC)DHE group satisfies the Gap Diffie-Hellman assumption

- TLS 1.3 mandates strong MODP DH groups and elliptic curves
- We may be able to redo our proof with other assumptions like PRF-ODH

Signatures are Unforgeable under Chosen Message Attack (UF-CMA)

- TLS 1.3 mandates RSA-PSS, EdDSA, and ECDSA, all with strong keys.

Hash functions are Collision Resistant

- TLS 1.3 mandates SHA-256 and stronger hash functions.

Authenticated encryption is IND-CPA and INT-CTXT

- TLS 1.3 mandates AES-GCM and Chacha20-Poly1305.
(EC)DHE group shared secret cannot be all 0s
• Avoids ambiguity between PSK and PSK-ECDHE handshakes
• May already be needed to avoid small subgroups

(EC)DHE group shared secret cannot be of the form:
\text{len} \ | \ “\text{TLS 1.3, ”} \ | \ l \ | \ h \ | \ 0x01
• Avoids collision between HKDF-Extract(es, e) and Derive-Secret(es, pbk, “”’ ) or Derive-Secret(es, ets, log )
• Independently discovered and discussed on TLS mailing list
• Led to change in Draft-19, which makes this assumption unnecessary

HMAC with different keys yields independent random oracles
• Needed to disambiguate various uses of HMAC in the protocol
Manual Proof of Composition for full TLS 1.3

Handshake without pre-shared key

Handshake with pre-shared key

Record protocol

\( ats_c \)

\( ats_s \)

\( psk' \)

\( ats_c \)

\( ats_s \)

\( ets_c \)

updated \( ts \)
The TLS 1.3 experiment

Multi-year effort to redesign IETF Transport Layer Security

• 4 years, 28 drafts, 12 IETF meetings

Major contributions from academic security researchers

• **Cryptographic proofs (of drafts 5,9,10)**

• **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]
Concluding Thoughts

Secure messaging is an exciting and important area of research
- Improved security definitions and protocols for two party messaging
- New requirements and protocols for group messaging
- Potential impact on WhatsApp, IETF MLS standard

Formal methods are effective on real-world messaging protocols
- Symbolic analyses in ProVerif or Tamarin can find attacks
- Mechanized security proofs in F* or CryptoVerif or EasyCrypt
- Many research challenges remain: Privacy, zero-knowledge proofs, ...

Verification of high-performance crypto software now feasible
- Verified code already deployed in Firefox, Linux, Tezos, Microsoft, ...
- Challenges, opportunities for impactful research:
  Blockchains, Privacy-Preserving Machine Learning, Post-Quantum Crypto
We are Hiring!

Are you interested in protocol verification? Designing new secure messaging systems? Implementing advanced crypto algorithms? Privacy-Preserving Machine Learning?

- Talk to us about PhDs, Post-Docs, visiting researcher positions
- Publications and more information: http://prosecco.inria.fr