Verifying Real-World Security Protocols

from finding attacks to proving security theorems

Karthik Bhargavan

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)**

- **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
| • 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**
<table>
<thead>
<tr>
<th>Year</th>
<th>Issue</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sep'11</td>
<td>BEAST</td>
<td>CBC predictable IVs</td>
</tr>
<tr>
<td>Sep'12</td>
<td>CRIME</td>
<td>Compression before Encryption</td>
</tr>
<tr>
<td>Mar'13</td>
<td>RC4</td>
<td>Keystream biases</td>
</tr>
<tr>
<td>May'13</td>
<td>Lucky 13</td>
<td>MAC-Encode-Encrypt CBC</td>
</tr>
<tr>
<td></td>
<td>HeartBleed</td>
<td>Memory safety bug</td>
</tr>
<tr>
<td></td>
<td>3Shake</td>
<td>Insecure resumption</td>
</tr>
<tr>
<td>Apr'14</td>
<td>POODLE</td>
<td>SSLv3 MAC-Encode-Encrypt</td>
</tr>
<tr>
<td></td>
<td>SMACK</td>
<td>State machine attacks</td>
</tr>
<tr>
<td>Apr'14</td>
<td>FREAK</td>
<td>Export-grade 512-bit RSA</td>
</tr>
<tr>
<td>May'15</td>
<td>LOGJAM</td>
<td>Export-grade 512-bit DH</td>
</tr>
<tr>
<td>Jan'15</td>
<td>SMACK</td>
<td>State machine attacks</td>
</tr>
<tr>
<td>Mar'15</td>
<td>FREAK</td>
<td>Export-grade 512-bit RSA</td>
</tr>
<tr>
<td>May'15</td>
<td>LOGJAM</td>
<td>Export-grade 512-bit DH</td>
</tr>
<tr>
<td>Jan'16</td>
<td>SLOTH</td>
<td>RSA-MD5 signatures</td>
</tr>
<tr>
<td>Mar'16</td>
<td>DROWN</td>
<td>SSLv2 RSA-PKCS#1v1.5</td>
</tr>
<tr>
<td>Aug'16</td>
<td>Sweet32</td>
<td>3DES and Blowfish</td>
</tr>
</tbody>
</table>

**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
Example: Diffie-Hellman key exchange

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

$A$

Knows $G = (g, p)$

$k_c = \text{kdf}(g^{xy'} \mod p)$

$g^x \mod p$

$g^{xy'} \mod p$

$B$

Knows $G = (g, p)$

$k_s = \text{kdf}(g^{x'y} \mod p)$

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

A

PKI

B

\[ G = (g, p) \]

\[ m_1 = g^x \mod p \]

\[ m_2 = g^y \mod p \]

\[ k = \text{kdf}(g^{xy} \mod p) \]

\[ \text{sign}(sk_A, \text{hash}(m_1 | m_2)), \text{mac}(k, A) \]

\[ \text{sign}(sk_B, \text{hash}(m_1 | m_2)), \text{mac}(k, B) \]

SIGNATURE + MAC prevents MitM attacks
Crypto Proof: Diffie-Hellman assumption

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

$g^{xy} \mod p$

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

Why a 512-bit group? backwards compatibility, export regulations,...
Protocol Flaw: group downgrade attack

The Logjam Attack [2015]: Crypto Weakness + Protocol Flaw
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)
• 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?
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
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),h,aaa,offer in
   let (early_secret:bitstring,kb:mac_key) = nego in
   let zoffer = nego(TLS13,DHE_13(g,gx),h,aaa,offer in
   if m = hmac(StrongHash,kb,ms2bytes(CH,cr,zoffer)) then
   let (kc0:ae_key,ems0:bitstring) =
     kdf_k0(early_secret,ms2bytes(ch)) in
   insert serverSession0(cr,psk,offer,kc0,ems0);

   new sr:random;
   in(io,SH(xxx,mode));
   let nego=TLS13,DHE_13(g,gx),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));
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
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 \( S \)

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

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

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

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

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

query attacker($\text{msg}(\text{conn}, S)$) $\implies$
    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?

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

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

• Strongest secrecy query that can be proved in our model

\[
\text{query} \ attacker(\text{msg}(\text{conn},S)) \implies \\
\text{event}(\text{WeakOr}\text{CompromisedKey}(S)) \lor \\
\text{event}(\text{ServerChoosesAE}(\text{conn},S,\text{WeakAE})) \lor \\
\text{event}(\text{ServerChoosesKEX}(\text{conn},S,\text{WeakDH})) \lor \\
\text{event}(\text{ServerChoosesKEX}(\text{conn'},S,\text{WeakRSADecryption})) \lor \\
\text{event}(\text{ServerChoosesHash}(\text{conn'},S,\text{WeakHash}))
\]

• 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

1. "Login with Facebook."
2. User authentication
3. Redirect to rp.com with Access Token \( \text{AT} \) in URI fragment (\# \( \text{AT} \) ...)
4. Retrieve URI (w/o token)
5. Send \( \text{AT} \)
6. Retrieve data using \( \text{AT} \)
7. Logged in

- 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)
IdP Mix-Up Attack in Implicit Mode

1.a "Login with Attacker"

1.b OK, Authenticate at Facebook

"User will now log in using Attacker as IdP"

2. user authentication

3. Redirect to rp.com with Access Token AT in URI fragment

4. retrieve URI

5. send AT

6. use AT
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://crypto-verif.inria.fr
• EasyCrypt: https://www.easycrypt.info