

# Type-based Verification at Scale

## miTLS: a verified reference implementation of TLS

Antoine Delignat-Lavaud

with

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss,  
Alfredo Pironti, Pierre-Yves Strub, Santiago Zanella Beguelin

<https://www.miTLS.org>



Microsoft Research

Microsoft Research - Inria  
**JOINT CENTRE**



# Transport Layer Security (1994—)

The most widely deployed cryptographic protocol?

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

20 years of attacks, fixes, and extensions

1994	Netscape's Secure Sockets Layer
1995	SSL3
1999	TLS1.0 (RFC2246, ~SSL3)
2006	TLS1.1 (RFC4346)
2008	TLS1.2 (RFC5246)

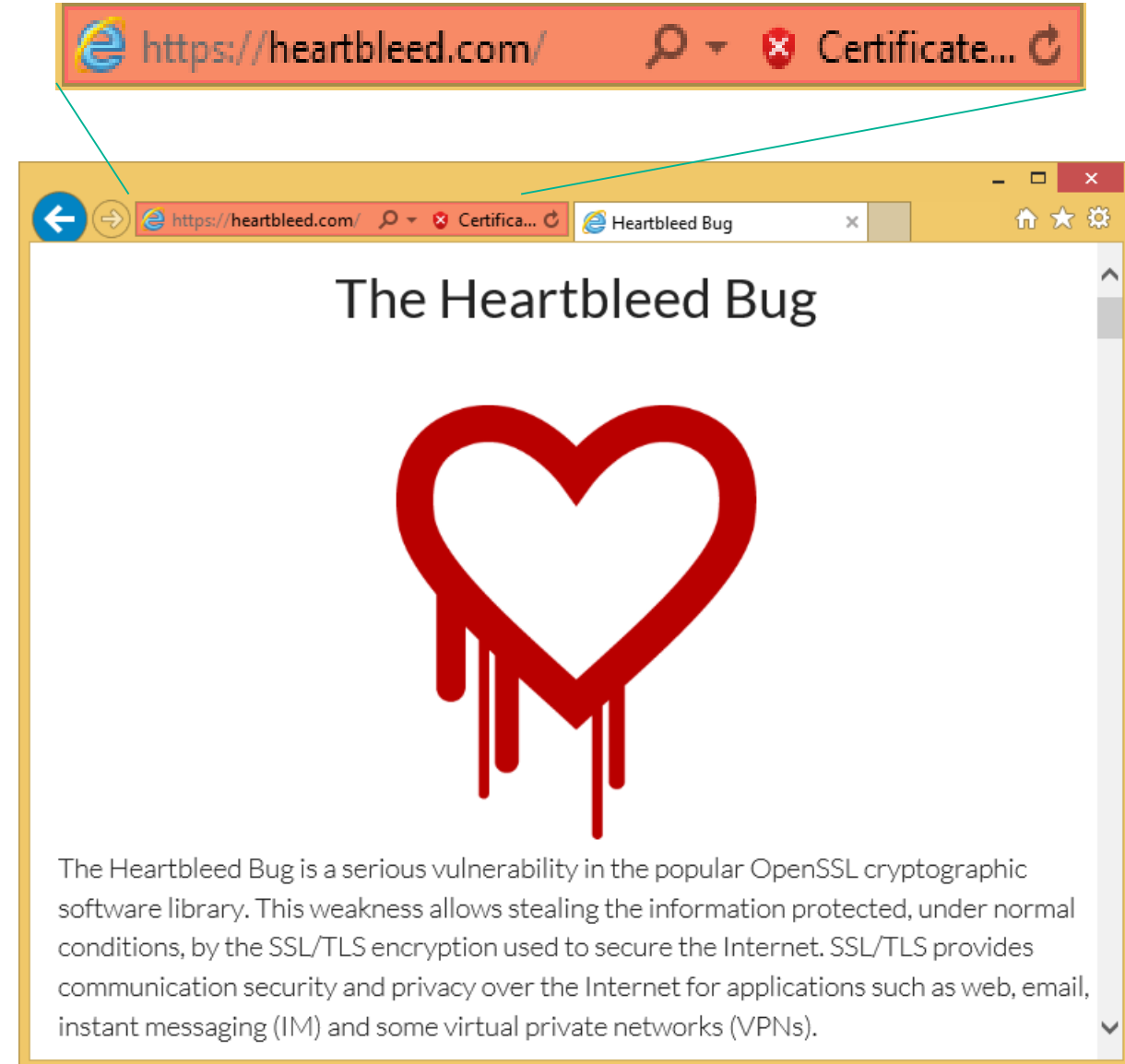
Many implementations

SChannel, OpenSSL, NSS, GnuTLS, JSSE, PolarSSL  
many patches every year; Snowden allegations

Many papers

Well-understood, detailed specs  
many security theorems...

mostly for small simplified models of TLS



What can still possibly go wrong?

## Infrastructure

certificate management (PKI)

### Protocol Logic

e.g. ambiguous messages

- cause clients and server to negotiate weak sessions

### TLS DESIGN

### Cryptography

e.g. not enough randomness

- write applet to realize adaptive attack (BEAST)

### Implementation Bugs

many critical errors

### Weak Algorithms

MD5, PKCS1, RC4, ...

## Application

HTTPS clients & servers

# ASN.1

Binary encoding  
standard

Ancient (1984)

<Tag, Length, Value>

Distinguished rules  
(DER): unique  
serialization

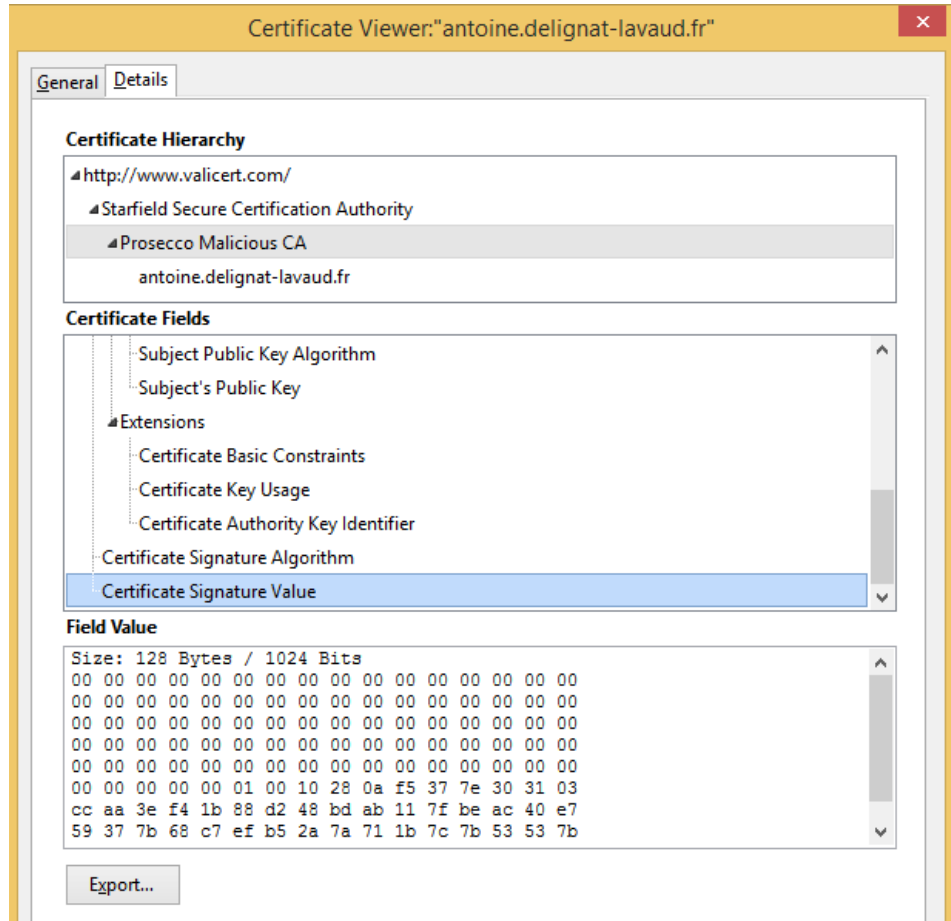
```
SEQUENCE (3 elem)
  SEQUENCE (8 elem)
    [0] (1 elem)
      INTEGER 2
    INTEGER (141 bit) 1492258819486064224988303096848576164759414
    SEQUENCE (2 elem)
      OBJECT IDENTIFIER 1.2.840.113549.1.1.5
      NULL
    SEQUENCE (2 elem)
      SET (1 elem)
        SEQUENCE (2 elem)
          OBJECT IDENTIFIER 2.5.4.10
          PrintableString AlphaSSL
        SET (1 elem)
          SEQUENCE (2 elem)
            OBJECT IDENTIFIER 2.5.4.3
            PrintableString AlphaSSL CA - G2
      SEQUENCE (2 elem)
        UTCTime 2013-06-02 17:27:55 UTC
        UTCTime 2017-06-02 17:27:55 UTC
    SEQUENCE (2 elem)
      SET (1 elem)
        SEQUENCE (2 elem)
          Offset: 132
          Length: 2+31
          (constructed) 1 elem
          Value:
            SEQUENCE (2 elem)
              OBJECT IDENTIFIER 2.5.4.11
              PrintableString Domain Control Validated
            SEQUENCE (2 elem)
              OBJECT IDENTIFIER 2.5.4.3
              PrintableString *.ht.vc
        SEQUENCE (2 elem)
          SEQUENCE (2 elem)
            OBJECT IDENTIFIER 1.2.840.113549.1.1.1
            NULL
          BIT STRING (1 elem)
            SEQUENCE (2 elem)
              INTEGER (2048 bit) 25070016126400689348179857701190619
              INTEGER 65537
      [3] (1 elem)
        SEQUENCE (9 elem)
          SEQUENCE (3 elem)
```

30	82	04	92	30	82	03	7A	A0	03	02	01	02	02	12	11
21	5A	C2	85	BD	0A	8C	58	07	4F	22	B4	89	04	29	87
76	30	0D	06	09	2A	86	48	86	F7	0D	01	01	05	05	00
30	2E	31	11	30	0F	06	03	55	04	0A	13	08	41	6C	70
68	61	53	53	4C	31	19	30	17	06	03	55	04	03	13	10
41	6C	70	68	61	53	53	4C	20	43	41	20	2D	20	47	32
30	1E	17	0D	31	33	30	36	30	32	31	37	32	37	35	35
5A	17	0D	31	37	30	36	30	32	31	37	32	37	35	35	5A
30	35	31	21	30	1F	06	03	55	04	0B	13	18	44	6F	6D
61	69	6E	20	43	6F	6E	74	72	6F	6C	20	56	61	6C	69
64	61	74	65	64	31	10	30	0E	06	03	55	04	03	14	07
2A	2E	68	74	2E	76	63	30	82	01	22	30	0D	06	09	2A
86	48	86	F7	0D	01	01	01	05	00	03	82	01	0F	00	30
82	01	0A	02	82	01	01	00	C6	97	C0	88	C6	30	A5	7A
0C	68	DA	22	F6	31	57	9C	9B	27	80	BB	CD	B9	D9	81
77	BF	6D	11	77	BE	9A	14	14	18	CB	BB	38	C4	90	74
0D	17	73	2C	DF	4E	34	F1	B4	C1	97	31	42	F5	DA	7E
ED	B6	76	B6	D1	9D	78	4F	D2	0F	31	27	AA	64	7E	B7
DC	88	63	BF	9F	00	02	BD	68	98	29	A8	36	B1	68	2B
9D	05	AF	A5	73	54	46	62	FE	7E	A0	D4	D8	AD	BF	F5
1A	CC	3F	B7	22	E5	4B	52	F9	38	26	98	5D	D6	07	F0
CB	0C	1E	FF	43	E2	A9	AD	BB	B1	CA	83	A0	33	4F	BA
76	4C	1E	CA	D9	A2	C4	86	F2	47	90	9B	98	92	57	76
F9	EE	5D	22	77	6F	EB	A3	EE	11	86	D2	13	C4	50	1C
90	09	62	D5	22	8E	DF	EB	51	8B	F7	3E	66	B9	45	BC
76	13	45	CE	92	59	AD	27	1B	B3	E3	25	1D	0A	13	FD
CA	94	BF	7D	60	37	03	00	20	87	D8	75	B2	49	03	5A
CF	96	17	79	C6	7C	46	6E	D1	C4	67	D9	E1	C9	64	7B
8A	72	0C	3A	2A	6E	C6	E4	45	6F	AD	A9	D7	28	ED	54
B3	F9	58	DB	21	B3	4D	D1	02	03	01	00	01	A3	82	01
A1	30	82	01	9D	30	0E	06	03	55	1D	0F	01	01	FF	04
04	03	02	05	A0	30	49	06	03	55	1D	20	04	42	30	40
30	3E	06	06	67	81	0C	01	02	01	30	34	30	32	06	08
2B	06	01	05	05	07	02	01	16	26	68	74	74	70	73	3A
2F	2F	77	77	77	2E	67	6C	6F	62	61	6C	73	69	67	6E
2E	63	6F	6D	2F	72	65	70	6F	73	69	74	6F	72	79	2F
30	19	06	03	55	1D	11	04	12	30	10	82	07	2A	2E	68
74	2E	76	63	82	05	68	74	2E	76	63	30	09	06	03	55

**Infrastructure**

Certificates are hard to check

# NSS Signature Forgery (August 2014)



## PKCS#1 Padding

```
01ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff000100307B300706052b0e03021a
04dc0146f9f544f3545f84977549d01efcf664cc4c1b603
```

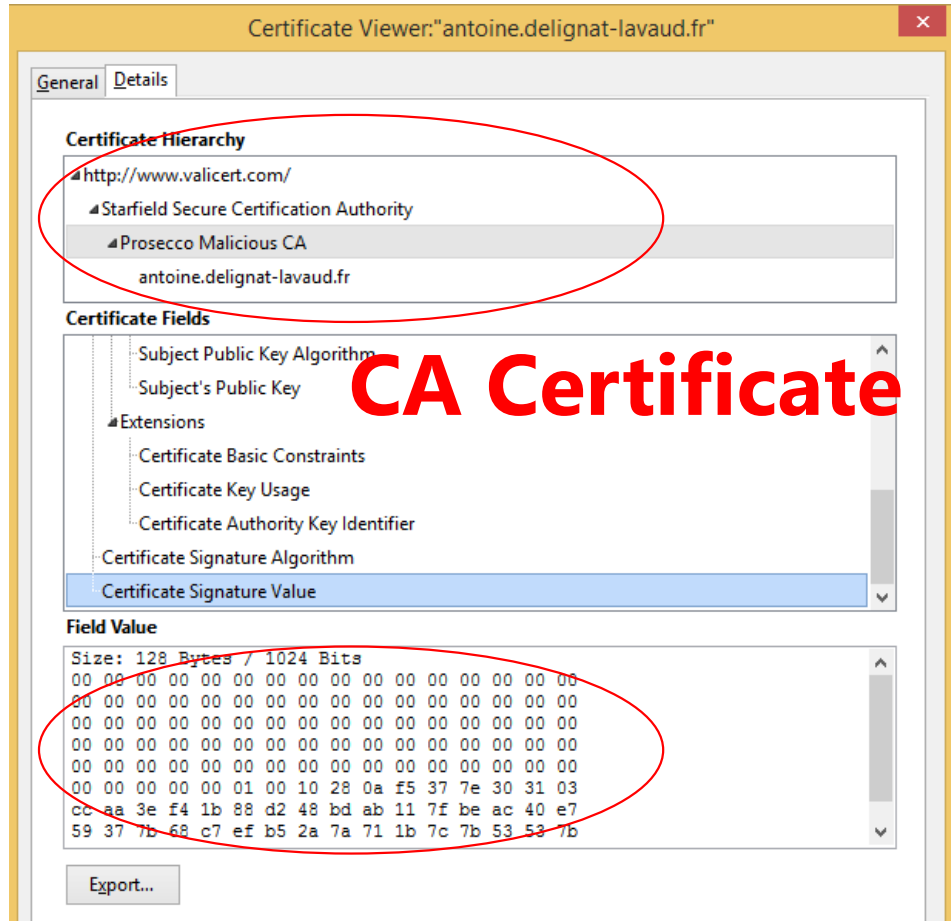
Signed hash

**Sign:**  $S = (\text{padding} || \text{oid} || h)^d \bmod N$   
**Verify:**  $S^e \bmod N$  (e.g.  $e=3$ )

## Infrastructure

Certificates are hard to check

# NSS Signature Forgery (August 2014)



## PKCS#1 Padding + hash algorithm OID

Injection of junk bytes  
Ignored by ASN.1 parser

```
000100307B300706052b0e03021a04dcxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxx0000000146f9f544f3545f84977549d01efcf664cc4c1b603
```

Signed hash

Bleichenbacher attack on low  
public exponents ( $e=3$ )  
Cubic root of padding + Fermat  
theorem for hash

## Infrastructure

Certificates are hard to check

gotofail bug  
iOS, Feb'14

then GnuTLS, Mar'14

then Heartbleed,  
OpenSSL, April'14

**Implementation Bugs**  
many critical errors

```
Solution1
gotofail.c  X

static OSStatus
SSLVerifySignedServerKeyExchange
(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
 uint8_t *signature, UInt16 signatureLen)
{
    OSStatus      err;
    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...

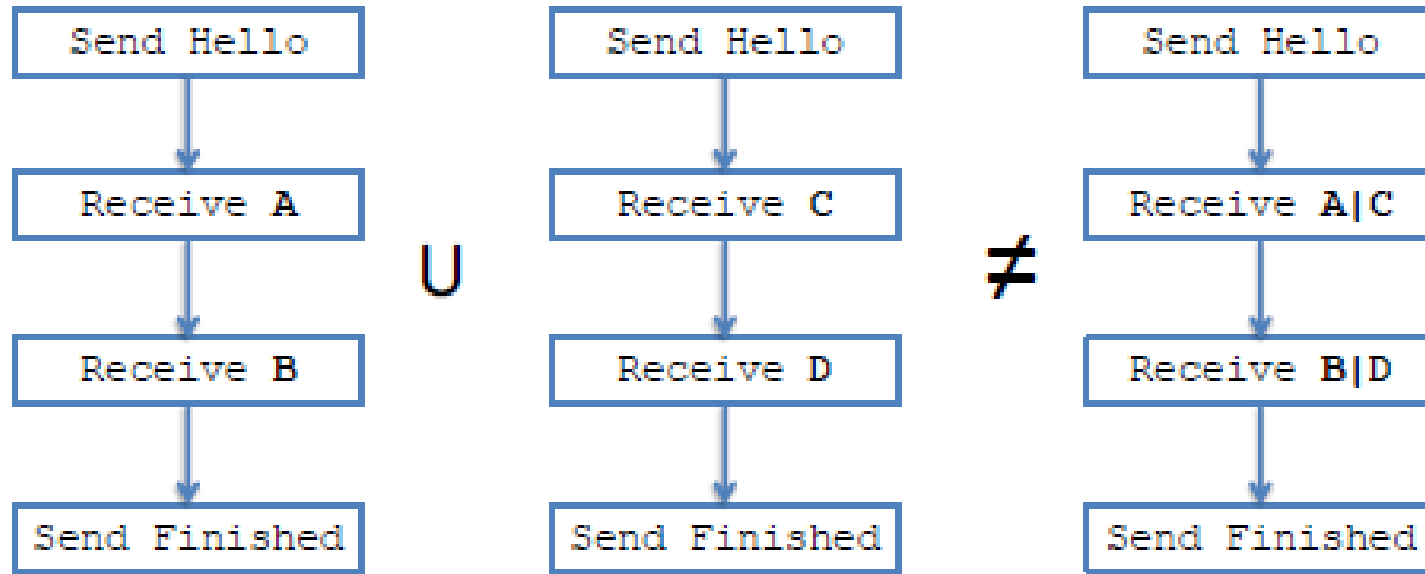
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

The duplicate goto always branches to the end of the function with err = 0

The key is not bound to the server signing-key certificate

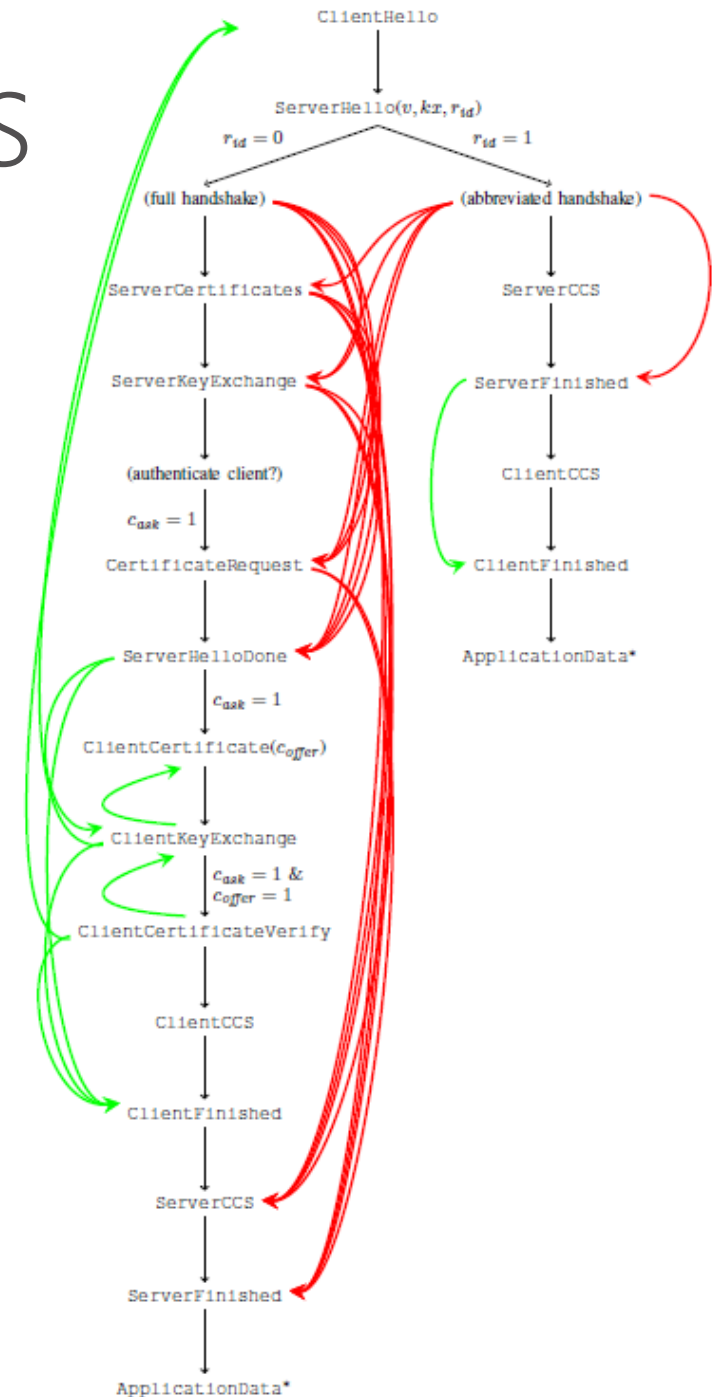


# SMACK: State Machine AttaCKs



## Implementation Bugs

## What gets really implemented?



# Application

## HTTPS clients & servers

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

Karthikeyan Bhargavan\*, Antoine Delignat-Lavaud\*, Cédric Fournet<sup>†</sup>, Alfredo Pironti\* and Pierre-Yves Strub<sup>‡</sup>

\*INRIA Paris-Rocquencourt <sup>†</sup>Microsoft Research <sup>‡</sup>IMDEA Software Institute

*Abstract*—TLS was designed as a transparent channel abstraction to allow developers with no cryptographic expertise to protect their application against attackers that may control some clients, some servers, and may have the capability to tamper with network connections. However, the security guarantees of TLS fall short of those of a secure channel, leading to a variety of attacks.

We show how some widespread false beliefs about these guarantees can be exploited to attack popular applications and defeat several standard authentication methods that rely too naively on TLS. We present new client impersonation attacks against TLS renegotiations, wireless networks, challenge-response protocols, and channel-bound cookies. Our attacks exploit combinations of RSA and Diffie-Hellman key exchange, session resumption, and renegotiation to bypass many recent countermeasures. We also demonstrate new ways to exploit known weaknesses of HTTP over TLS. We investigate the root causes for these attacks and propose new countermeasures. At the protocol level, we design and implement two new TLS extensions that strengthen the authentication guarantees of the handshake. At the application level, we develop an exemplary HTTPS client library that implements several mitigations, on top of a previously verified TLS implementation, and verify that their composition provides strong, simple application security.

sessions, validating certificates, etc. Meanwhile, TLS applications continue to rely on URLs, passwords, and cookies; they mix secure and insecure transports; and they often ignore lower-level signals such as handshake completion, session resumption, and truncated connections.

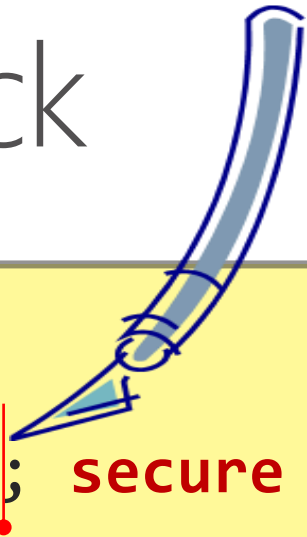
Many persistent problems can be blamed on a mismatch between the authentication guarantees expected by the application and those actually provided by TLS. To illustrate our point, we list below a few myths about those guarantees, which we debunk in this paper. Once a connection is established:

- 1) the principal at the other end cannot change;
- 2) the master secret is shared only between the two peers, so it can be used to derive fresh application-level keys;
- 3) the `tls-unique` channel binding [6] uniquely identifies the connection;
- 4) the connection authenticates the whole data stream, so it is safe to start processing application data as it arrives.

The first is widely believed to be ensured by the TLS renegotiation extension [49]. The second and third are used for man-in-the-middle protections in tunneled protocols like PEAP and some authentication modes in SASL and GSS-API. The fourth

new attacks  
found while  
studying  
HTTPS

# Cookie Cutter Attack



```
HTTP/1.1 302 Redirect
Location: https://x.com/P
Set-Cookie: SID=[SessionToken]; secure
Content-Length: 0
```

*Protected by TLS*

Many web services rely on session tokens to authenticate their users

The **secure** cookie attribute tells the client browser that the cookie is HTTPS-only

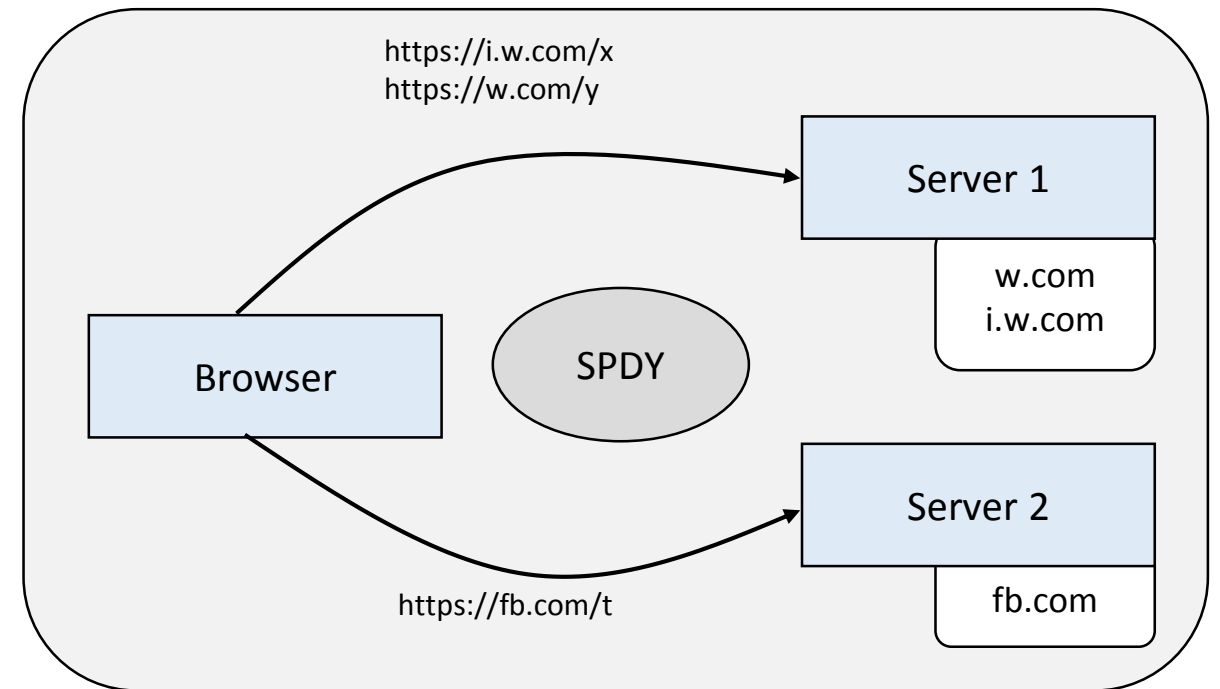
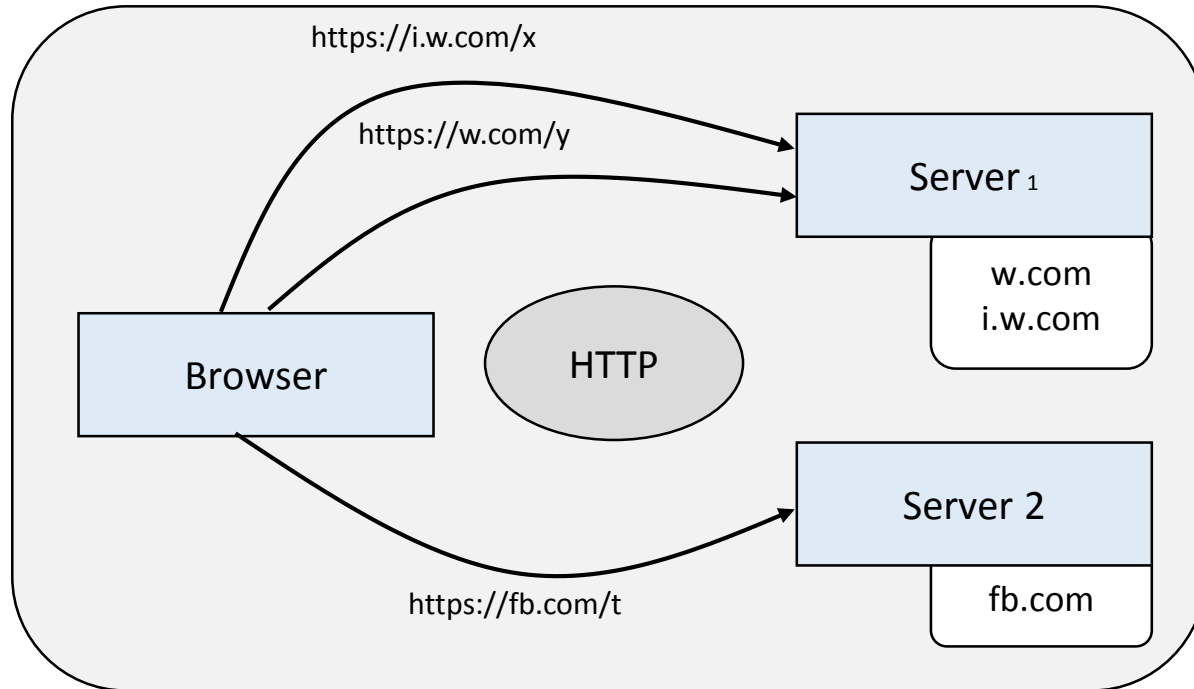
Many browsers silently process truncated HTTP (e.g. images)

After truncation, any fake HTTP query leaks the authentication token

Browser vulnerable to truncations?	Header	Body (Length)	Body (Chunked)
Android 4.2.2	YES	YES	YES
Chrome 27	YES	YES	YES
Chrome 28	NO	NO	YES
Firefox 24	NO	YES	YES
Safari Mobile 7.0.2	YES	YES	YES
Opera Mini 7.5	YES	YES	YES
Opera Classic 12.1	YES	YES	YES
Internet Explorer 10	NO	YES	YES

**Application**  
HTTPS clients & servers

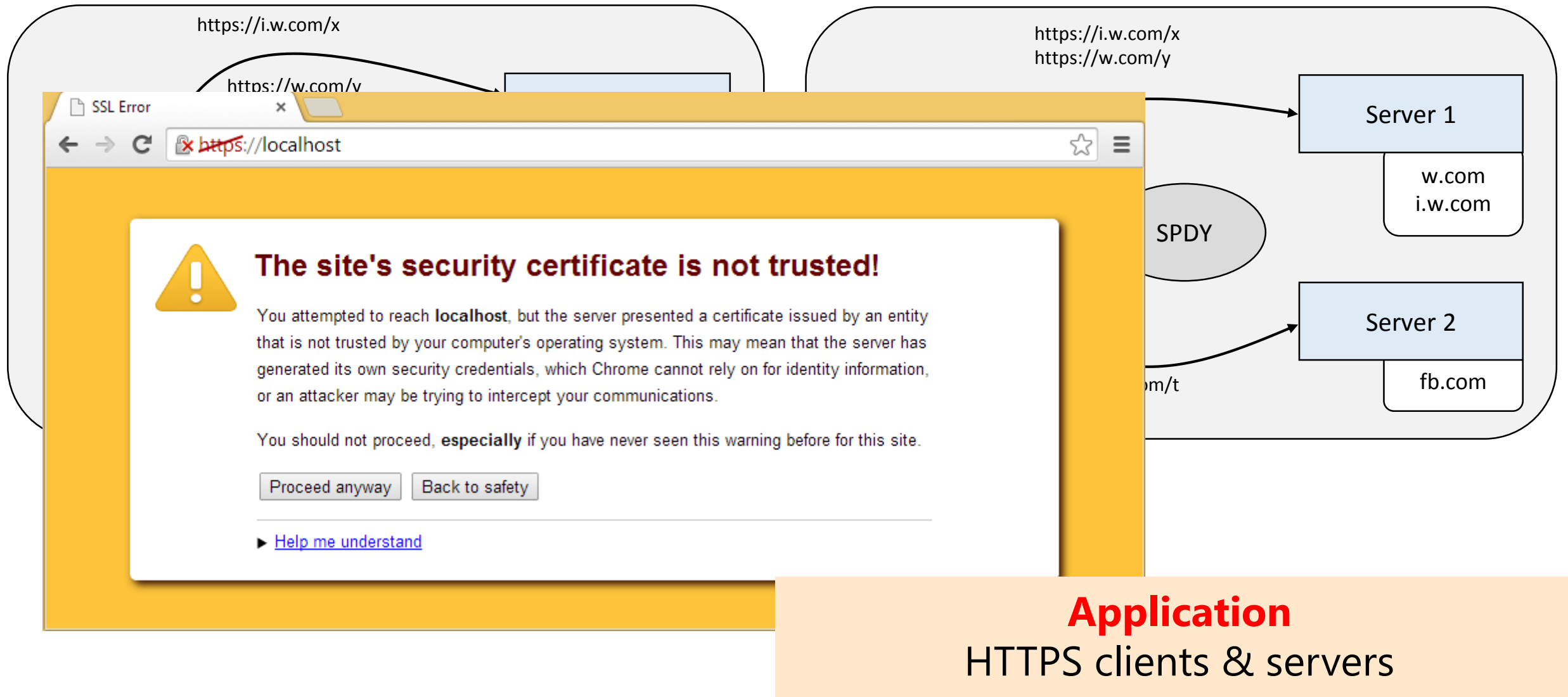
# SPDY Connection Pooling Attack



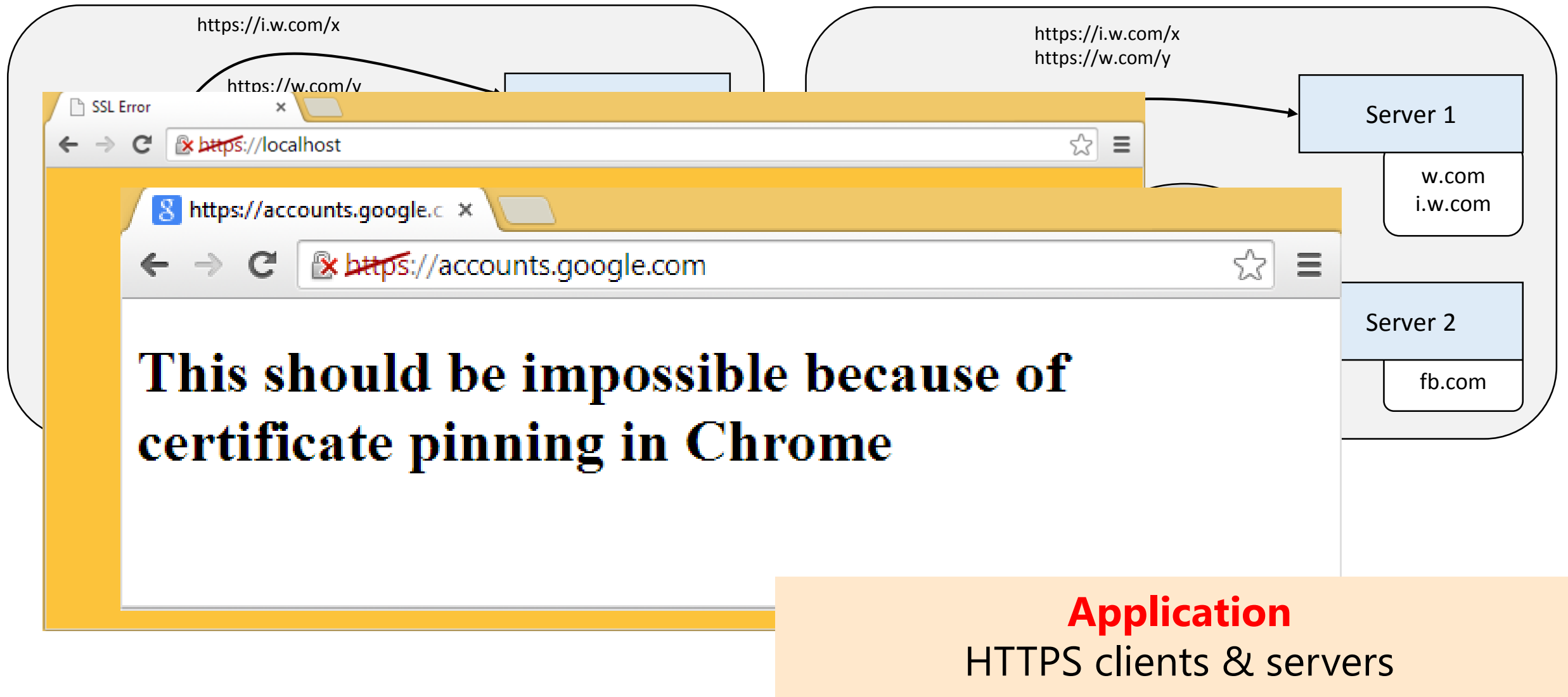
**Application**

HTTPS clients & servers

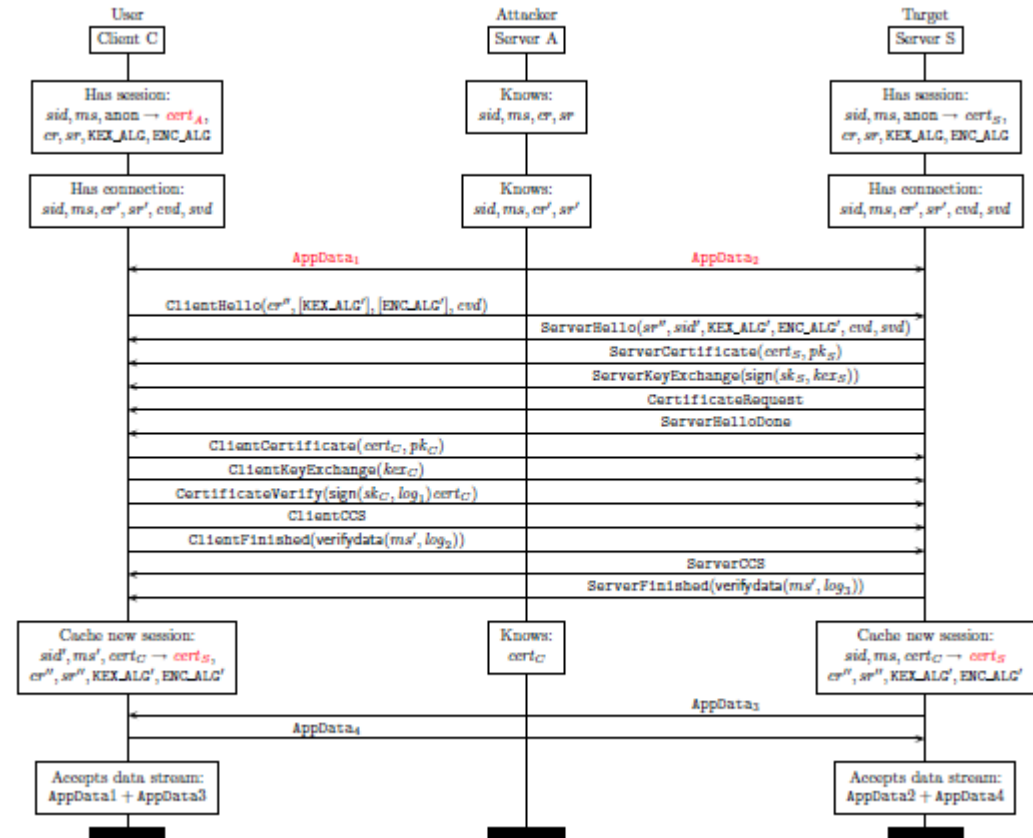
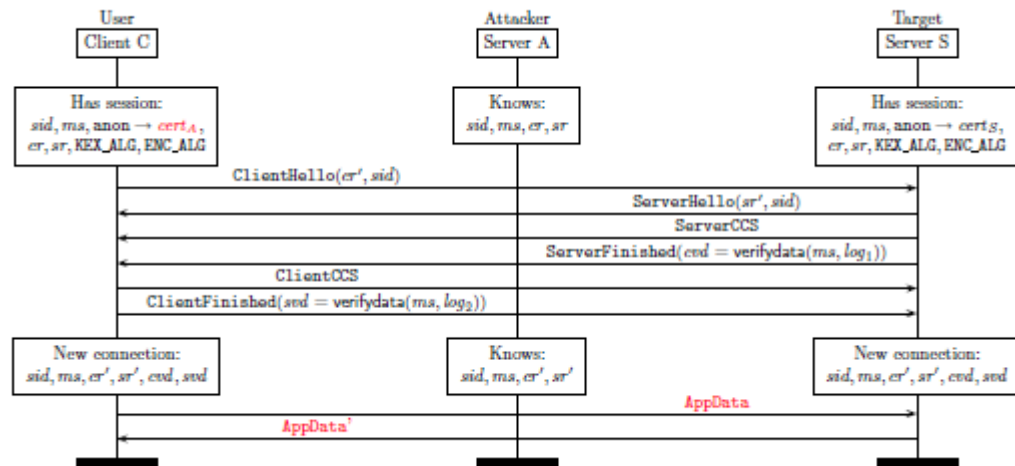
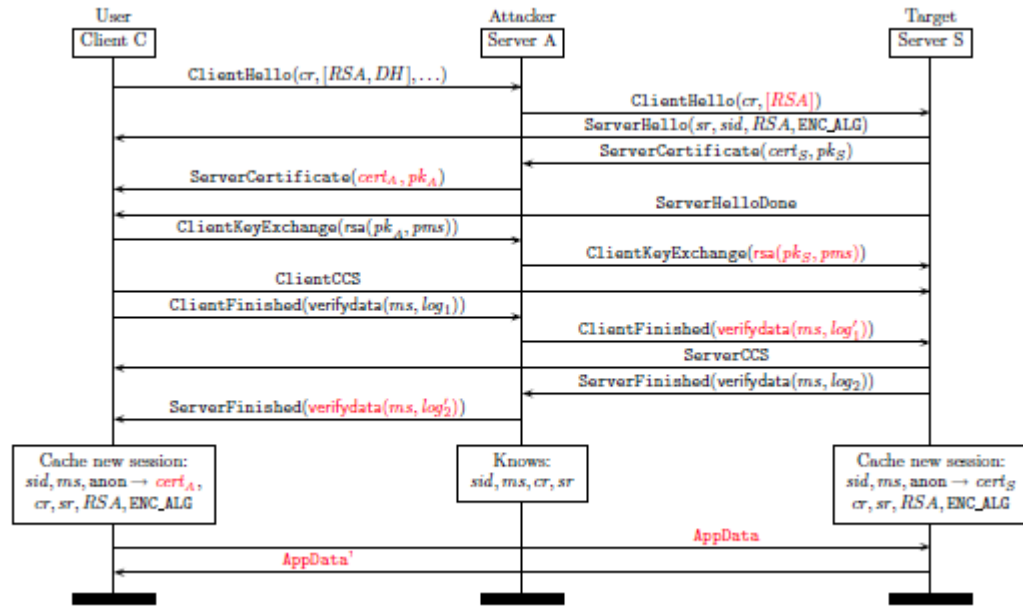
# SPDY Connection Pooling Attack



# SPDY Connection Pooling Attack



# Triple Handshake Attack



**Protocol Logic**  
Bad compositions of protocol features



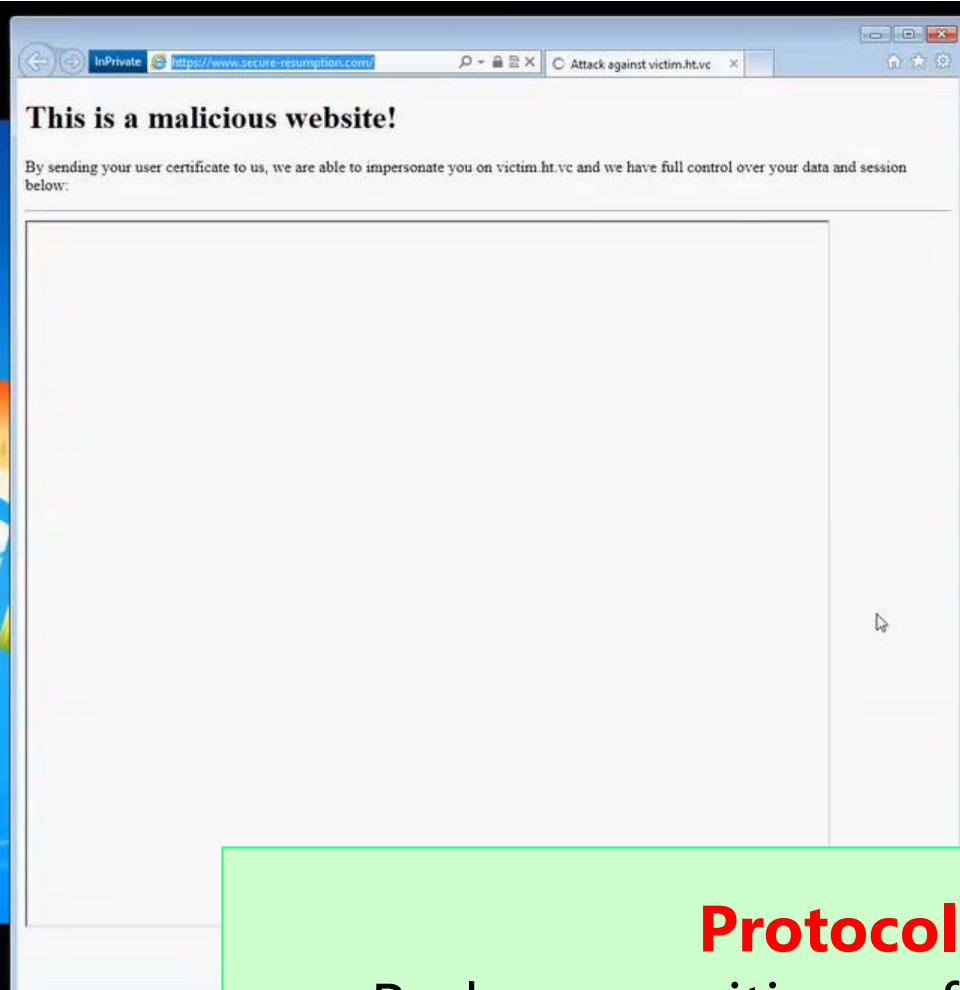
# Triple Handshake Attack

a server-in-the-middle, using 3 related handshakes

Client	TLS library
Chromium Opera 15+	NSS
Internet Explorer	SChannel
Safari & Apple mail	Secure Transport
Apple Mail	Secure Transport
CURL	OpenSSL
CURL	GnuTLS
Wget	OpenSSL
NodeJS HTTPS	OpenSSL
PHP SSL Transport	OpenSSL
Apache HttpClient	JSSE 1.7
SVN / Neon	OpenSSL
Qt / CORE	OpenSSL
Qt / QT	OpenSSL

```
max@argon: /var/prosecco/projects/tls-sessions/3rdparty/gnutls-3.2.4/doc/examples
Sending server hello...
Sending server certificate...
Sending client certificate request...
Sending server hello done...
Sending server certificate...
Client sent his encrypted PMS!
Checking client certificate...
Client/Server finished.
Client/Server finished.
+++ Storing session (32-1018) in cache[0]
*** Rehandshake was completed
- Key Exchange: RSA
Peer provided 1 certificates.
Certificate info:
  subject 'C=CC,ST=State,O=Organization,CN=alfredo@pironti.eu,EMAIL=root@mitls.org', issuer 'C=CC,ST=State,L=Location,O=Organization,CN=rsa.mtls.org,EMAIL=root@mitls.org', RSA key 1024 bits, signed using R
SA-SHA1, activated '2013-10-17 10:34:04 UTC', expires '2022-01-03 10:34:04 UTC', SHA-1 fingerprint 'elab93
91e4abd9fab63be6d7861f1102041afa'
Certificate is valid since: Thu Oct 17 12:34:04 2013
Certificate expires: Mon Jan  3 11:34:04 2022
Certificate serial number: 03
Certificate public key: RSA Certificate version: #3
DN: C=CC,ST=State,O=Organization,CN=alfredo@pironti.eu,EMAIL=root@mitls.org
Issuer's DN: C=CC,ST=State,L=Location,O=Organization,CN=rsa.mtls.org,EMAIL=root@mitls.org
- Protocol: TLS1.2
- Certificate Type: X.509
- Compression: NULL
- Cipher: AES-128-CBC
- MAC: SHA1
- Rehandshaked Resumed Session: no
- Sending HTTP response...
*** Child process killed
```

```
C:\Users\alfredo\Work\prosecco-git\projects\tls-sessions\mtls-attack\attack\bin\Debug\attack.exe
Starting the handshake tampering phase
[Thread 1] Client killed connection before sending certificate
[Thread 1] Client killed connection before sending certificate
[Thread 1] Successful server handshake
[Thread 1] Successful client handshake
[Thread 1] Serving local home page, then closing
Starting one passthrough phase
[Thread 5] Starting passthrough
```



**Protocol Logic**

Bad compositions of protocol features



## Infrastructure

certificate management (PKI)

### Protocol Logic

e.g. ambiguous messages

- cause clients and server to negotiate weak sessions

### TLS DESIGN

### Cryptography

e.g. not enough randomness

- write applet to realize adaptive attack (BEAST)

### Implementation Bugs

many critical errors

### Weak Algorithms

MD5, PKCS1, RC4, ...

## Application

HTTPS clients & servers

To get application security,  
we must capture **all** these aspects  
within the same model

- We build a verified reference implementation
- We use automated proof tools to scale up

# A cryptographically verified reference implementation of TLS

# <https://www.miTLS.org>

We develop and verify a reference implementation for SSL 3.0—TLS 1.2

1. **Standard compliance:** we closely follow the RFCs
  - concrete message formats
  - support for multiple ciphersuites, sessions and connections, re-handshakes and resumptions, alerts, message fragmentation,...
  - interop with other implementations such as web browsers and servers
2. **Verified security:** we structure our code to enable its modular verification, from its main API down to concrete assumptions on its base cryptography (e.g. RSA)
  - probabilistic computational security theorems for a 7000-line functionality (automation required)
3. **Experimental platform:** for testing corner cases, trying out attacks, studying application-level protocols, analysing new extensions and patches, ...

# Ciphersuites & Crypto Agility

TLS negotiates its use of cryptography

Not all algorithms are equal!

Cautionary tale: ECDHE considered safest, open to attack for 2 years due to bug in elliptic curve fast multiplication

Clients and servers should get security for the ciphersuite they prefer, not the weakest they support

Circular dependency: TLS relies on the ciphersuites being negotiated

We verify TLS **generically**, for multiple ciphersuites & algorithms

This requires new cryptographic models

```
TLS_NULL_WITH_NULL_NULL
TLS_RSA_WITH_NULL_MD5
TLS_RSA_WITH_NULL_SHA
TLS_RSA_WITH_NULL_SHA256
TLS_RSA_WITH_RC4_128_MD5
TLS_RSA_WITH_RC4_128_SHA
TLS_RSA_WITH_3DES_EDE_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA
TLS_RSA_WITH_AES_128_CBC_SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
TLS_DH_anon_WITH_RC4_128_MD5
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA
TLS_DH_RSA_WITH_AES_128_CBC_SHA
TLS_DHE_DSS_WITH_AES_128_CBC_SHA
TLS_DHE_RSA_WITH_AES_128_CBC_SHA
TLS_DH_anon_WITH_AES_128_CBC_SHA
TLS_DH_DSS_WITH_AES_256_CBC_SHA
TLS_DH_RSA_WITH_AES_256_CBC_SHA
TLS_DHE_DSS_WITH_AES_256_CBC_SHA
TLS_DHE_RSA_WITH_AES_256_CBC_SHA
TLS_DH_anon_WITH_AES_256_CBC_SHA
TLS_DH_DSS_WITH_AES_128_CBC_SHA256
TLS_DH_RSA_WITH_AES_128_CBC_SHA256
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
TLS_DH_anon_WITH_AES_128_CBC_SHA256
TLS_DH_DSS_WITH_AES_256_CBC_SHA256
TLS_DH_RSA_WITH_AES_256_CBC_SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
TLS_DH_anon_WITH_AES_256_CBC_SHA256
```

# Verification Method: Type-Based Cryptography

Cryptographic algorithms

types express cryptographic assumptions

Cryptographic constructions

types express security guarantees

Security protocols

types express attacker models

Applications & Adversaries

