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
informatiques mathématiques inria
institute IMdea
madrid institute for advanced studies

Transport Layer Security (1994—)

The most widely deployed cryptographic protocol?

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

20 years of attacks, fixes, and extensions

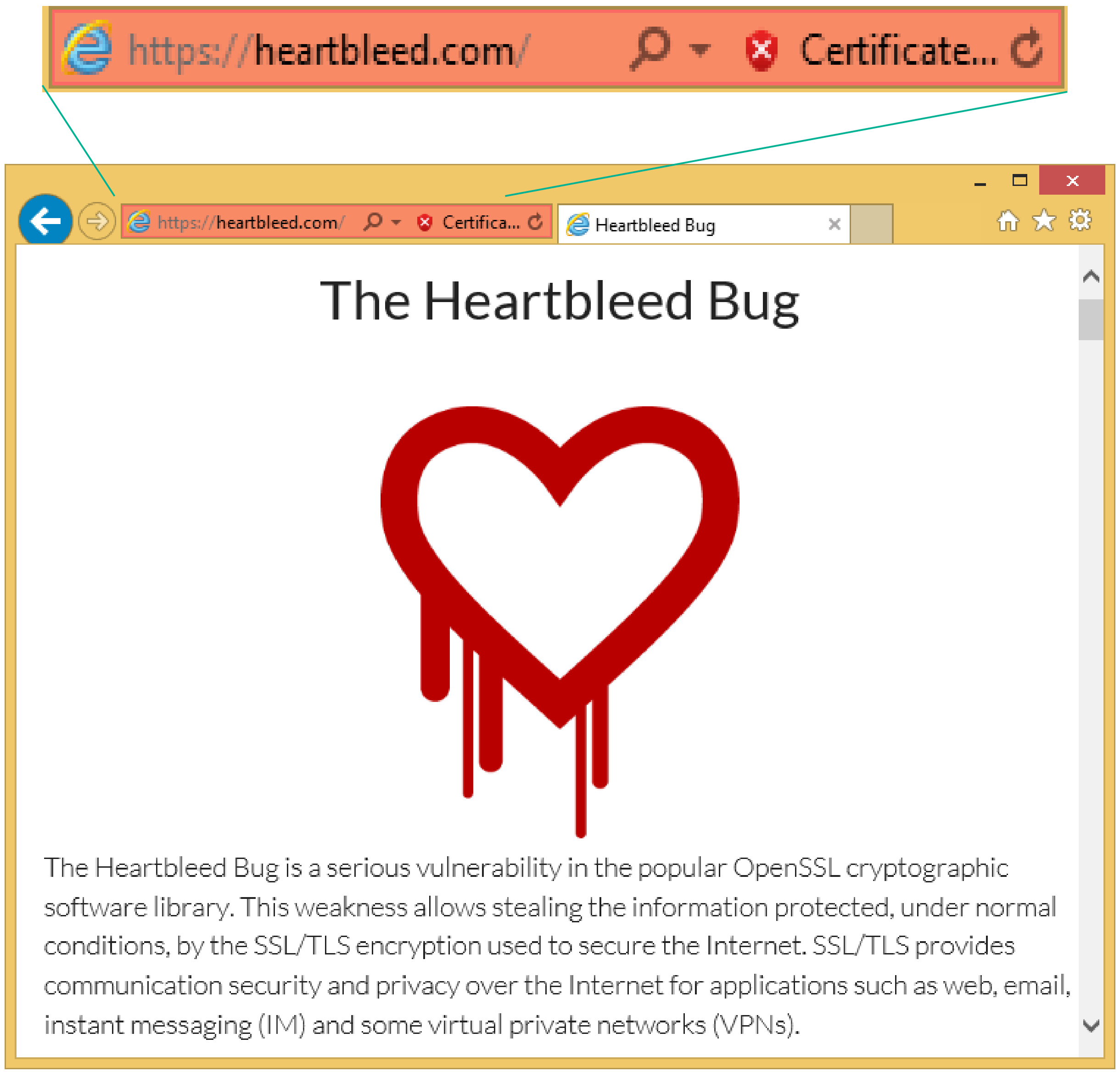
1994	Netscape's Secure Sockets Layer
1995	SSL3
1999	TLS1.0 (RFC2246, ≈SSL3)
2006	TLS1.1 (RFC4346)
2008	TLS1.2 (RFC5246)

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)

TLS DESIGN

Protocol Logic
e.g. ambiguous messages

- cause clients and server to negotiate weak sessions

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

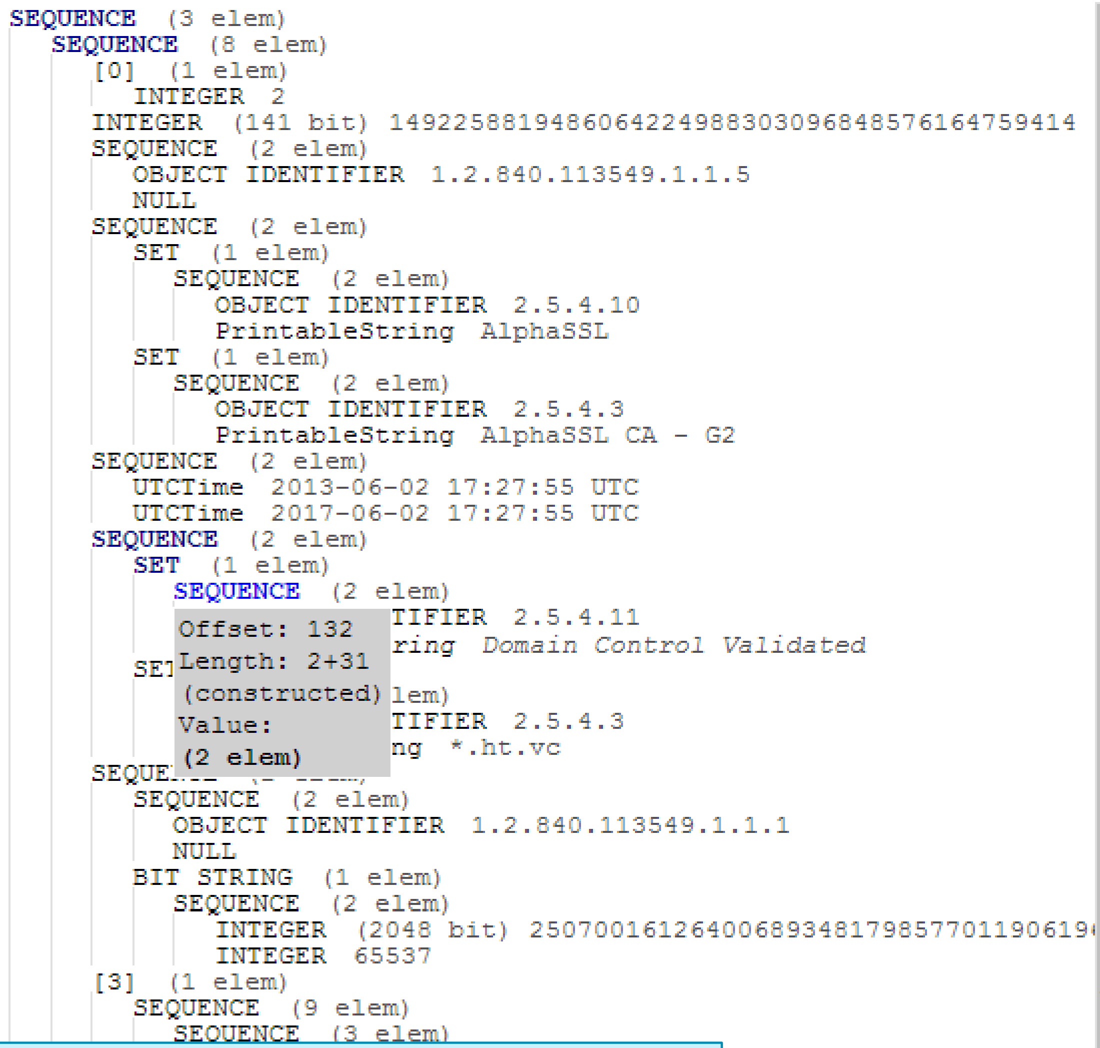
ASN.1

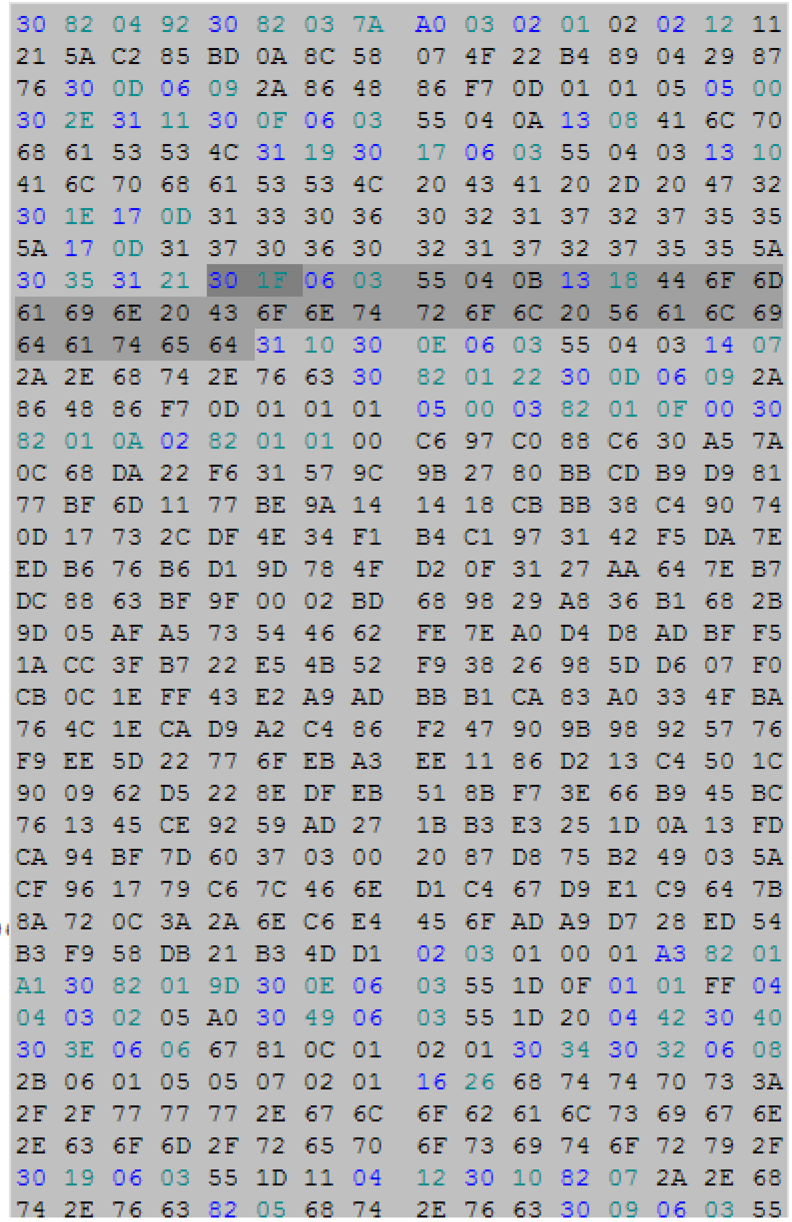
Binary encoding standard

Ancient (1984)

<Tag, Length, Value>

Distinguished rules (DER): unique serialization

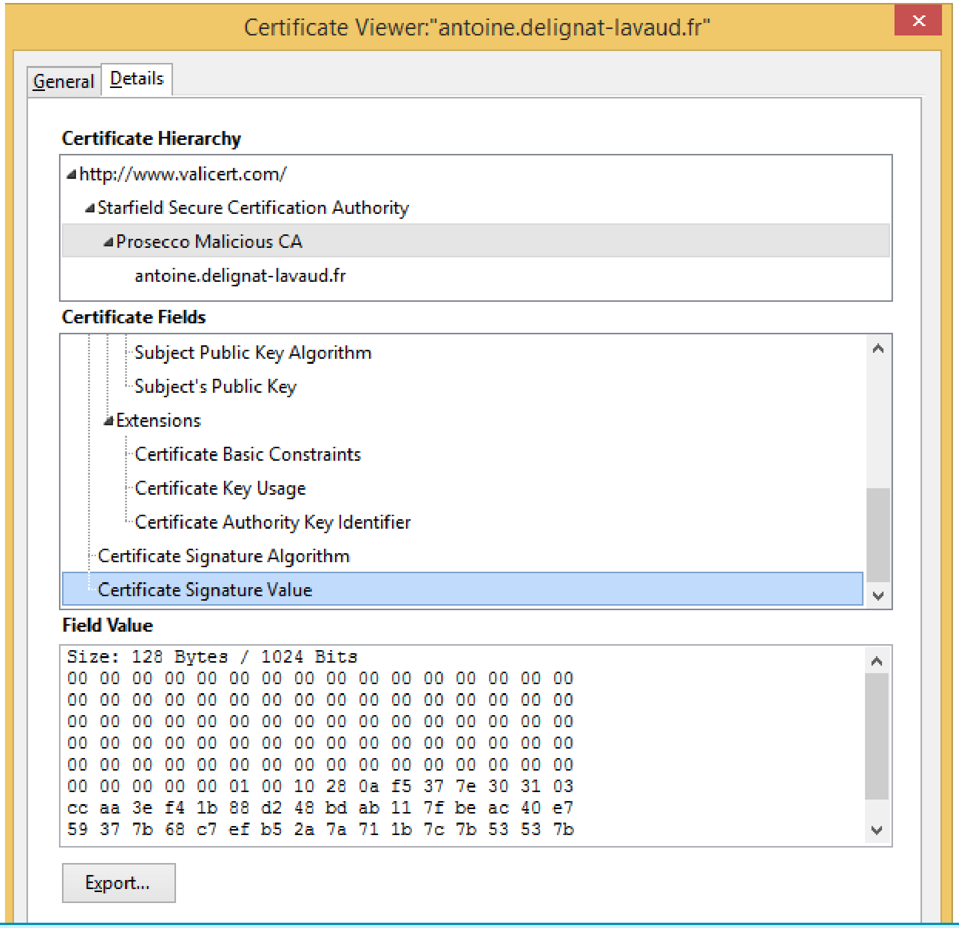




Infrastructure

Certificates are hard to check

NSS Signature Forgery (August 2014)



PKCS#1 Padding

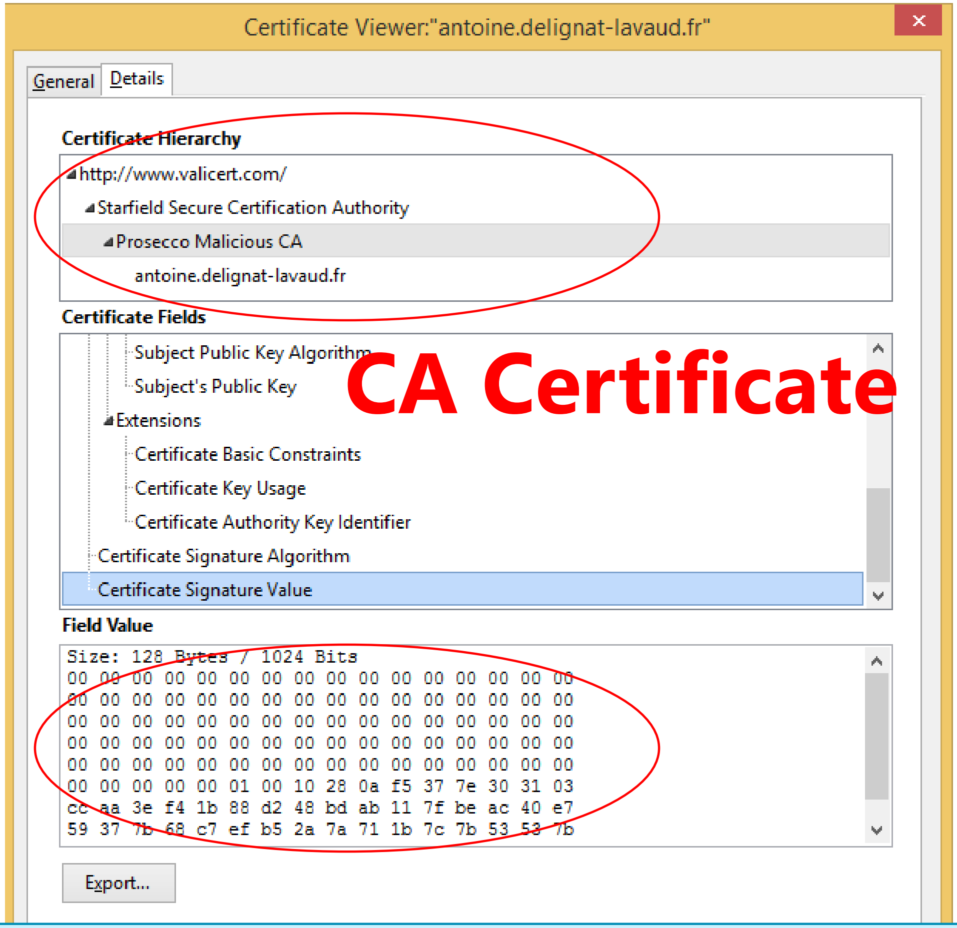
01ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffff000100307B300706052b0e03021a
04dc0146f9f544f3545f84977549d01efcf664cc4c1b603

Signed hash

Sign: S = (padding||oid||h)^d mod N
Verify: S^e mod 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

000100307B300706052b0e03021a04dcxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxx0000000146f9f544f3545f84977549d01efcf664cc4c1b603

Signed hash

Infrastructure
Certificates are hard to check

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

gotofail bug

iOS, Feb'14

then GnuTLS, Mar'14

then Heartbleed, OpenSSL, April'14

Implementation Bugs

many critical errors

Solution1

gotofail.c

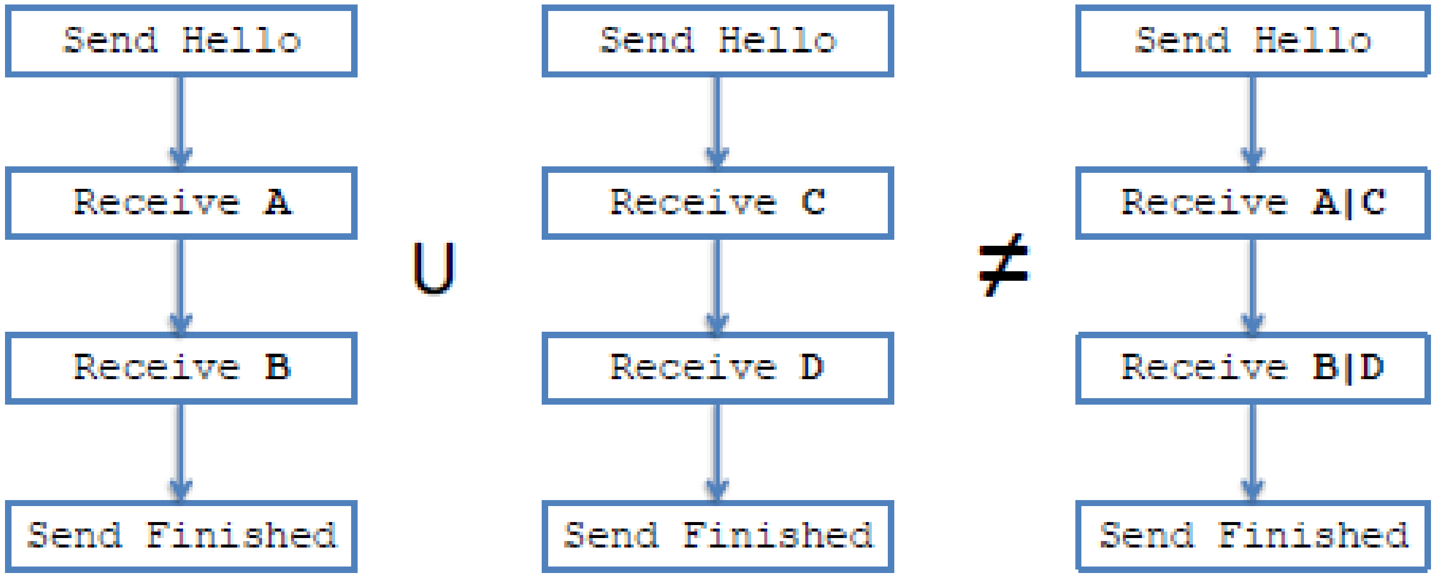
```c
static OSStatus
SSLVerifySignedServerKeyExchange
    (SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
     uint8_t *signature, UInt16 signatureLen)
{
    OSStatus        err;
    ...

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

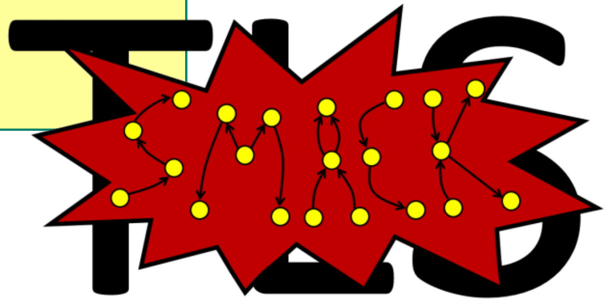
The key is not bound to the server signing-key certificate

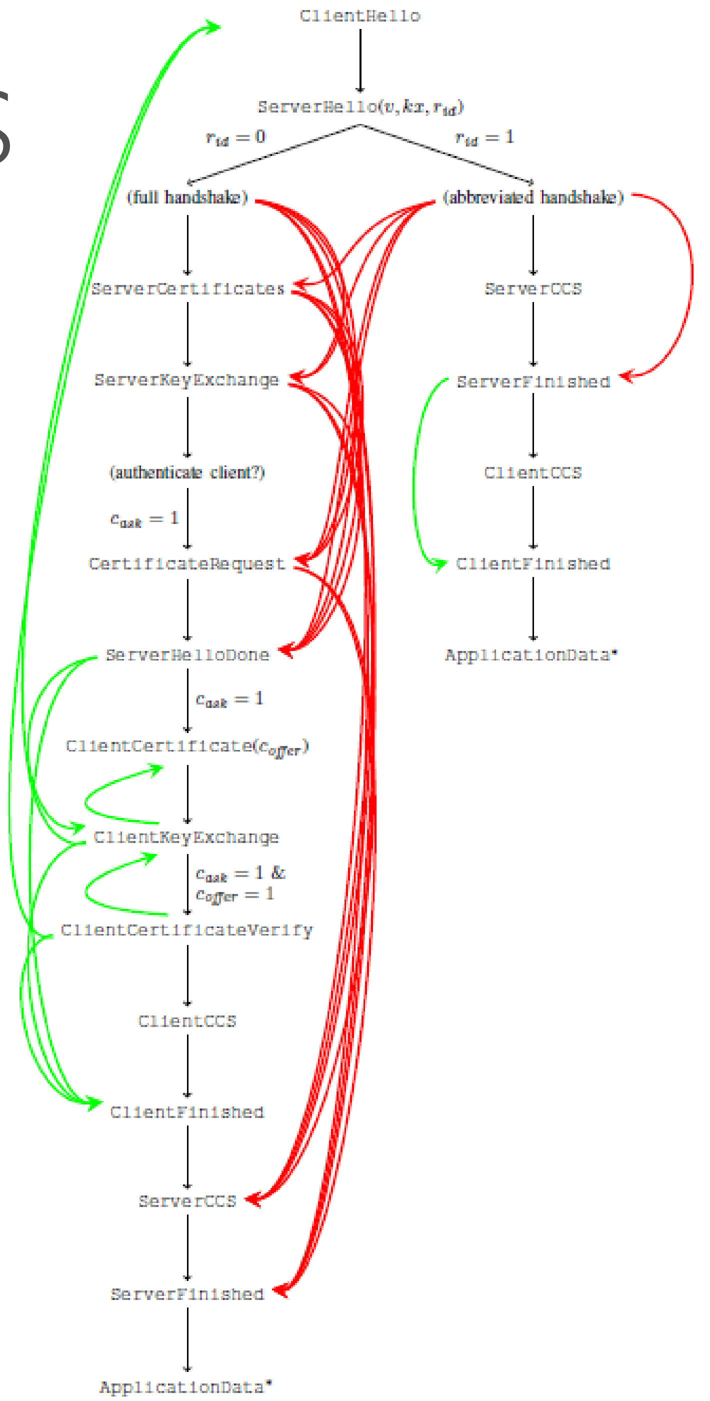
140 %

SMACK: State Machine AttaCKs



Implementation Bugs
What gets really implemented?





Application

HTTPS clients & servers

new attacks found while studying HTTPS

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

Karthikeyan Bhargavan*, Antoine Delignat-Lavaud*, Cédric Fournet†, Alfredo Pironti* and Pierre-Yves Strub‡
*INRIA Paris-Rocquencourt †Microsoft Research ‡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.

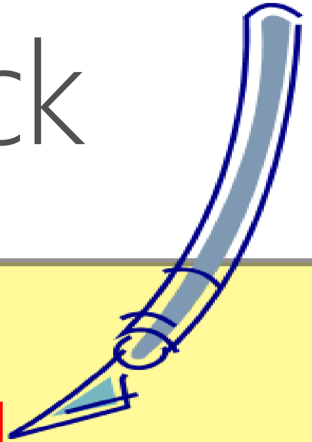
Many persistent problems can be blamed on a mismatch between the authentication guarantees expected by the application and those actually provided by TLS. To illustrate our point, we list below a few myths about those guarantees, which we debunk in this paper. Once a connection is established:

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

IEEE Security & Privacy 2014

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

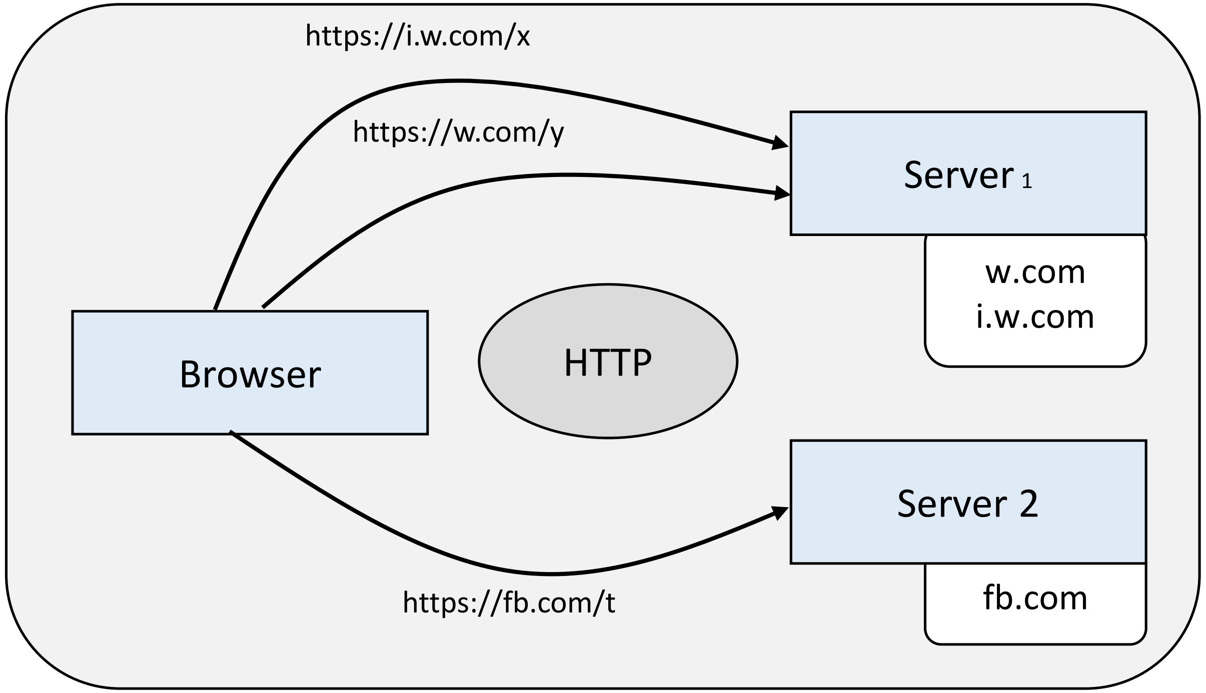
Many browsers silently process truncated HTTP (e.g. images)

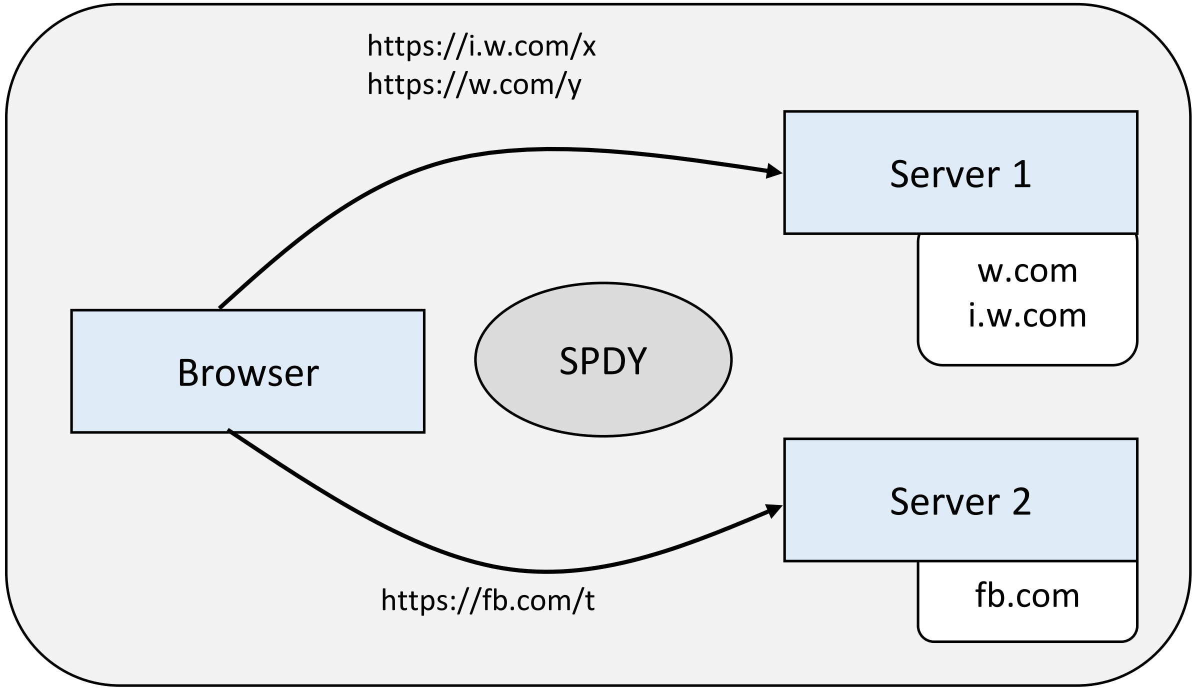
After truncation, any fake HTTP query leaks the authentication token

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

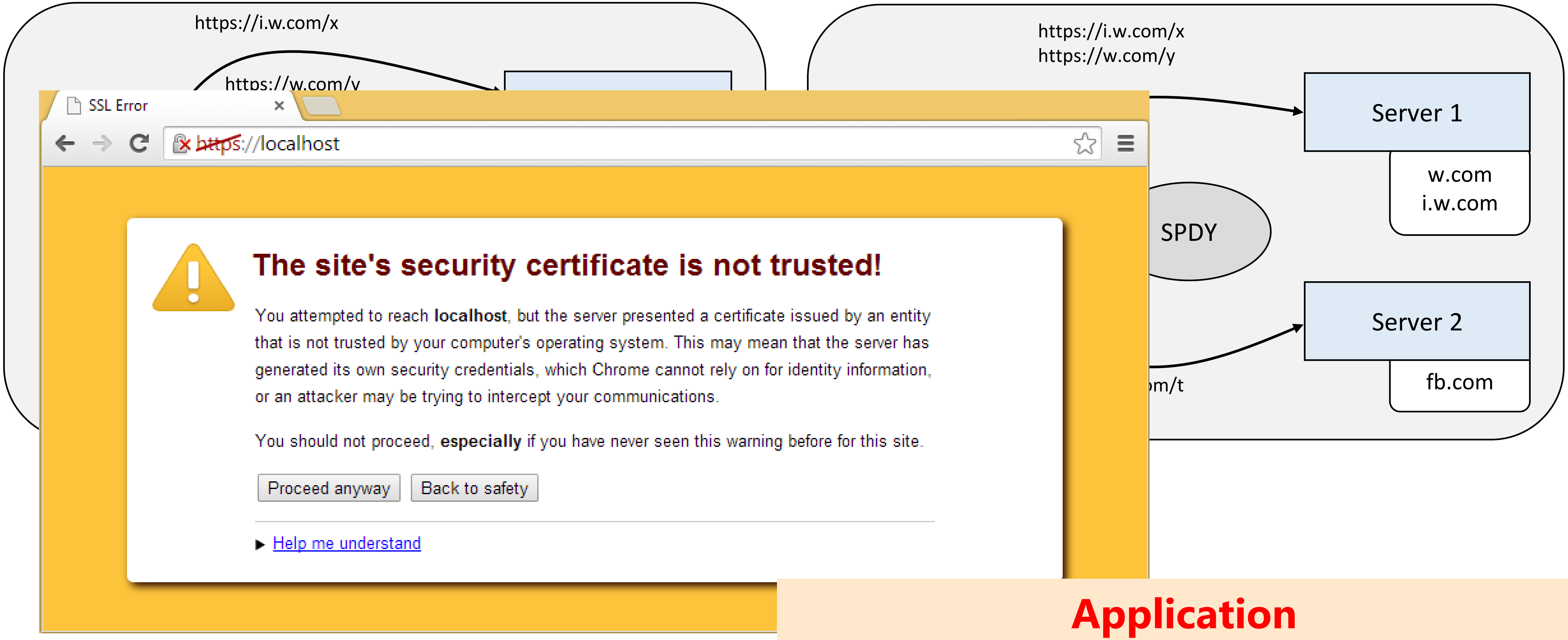
SPDY Connection Pooling Attack





Application
HTTPS clients & servers

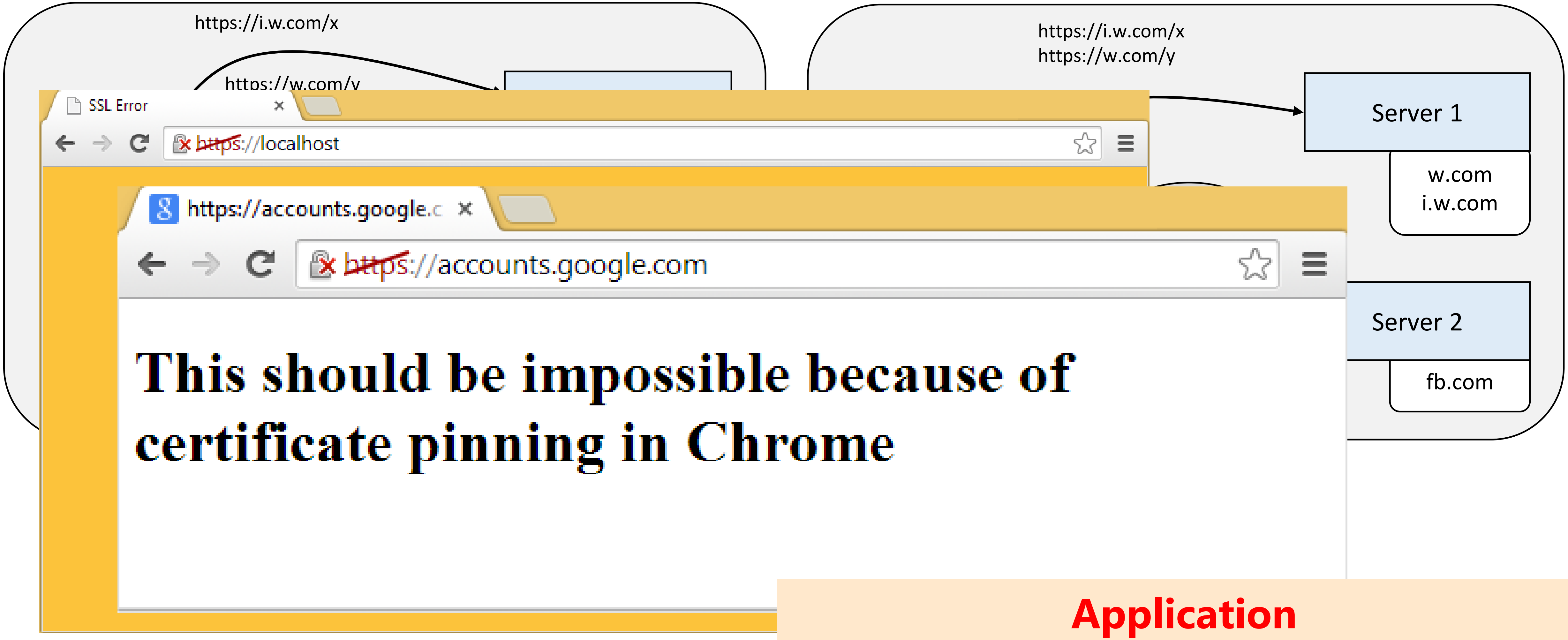
SPDY Connection Pooling Attack



Application

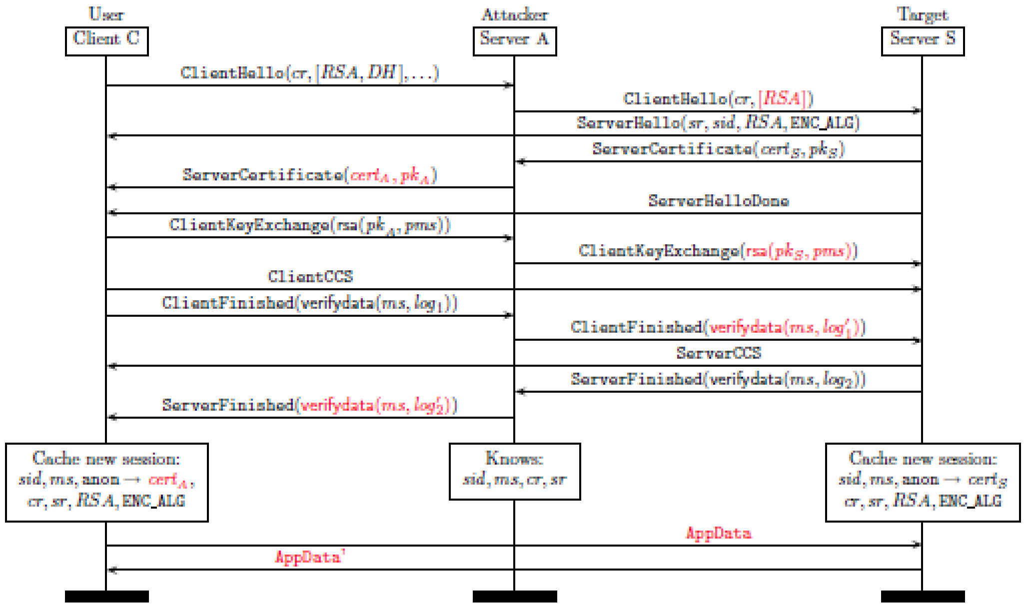
HTTPS clients & servers

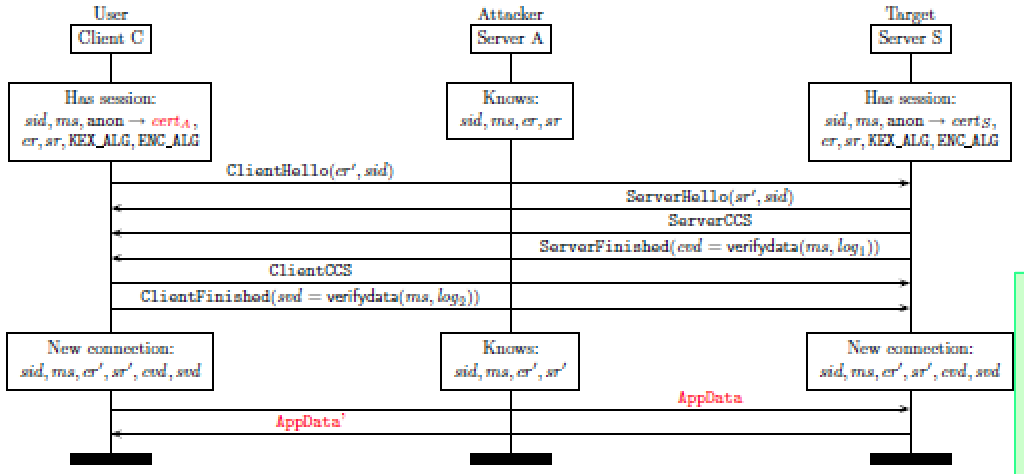
SPDY Connection Pooling Attack

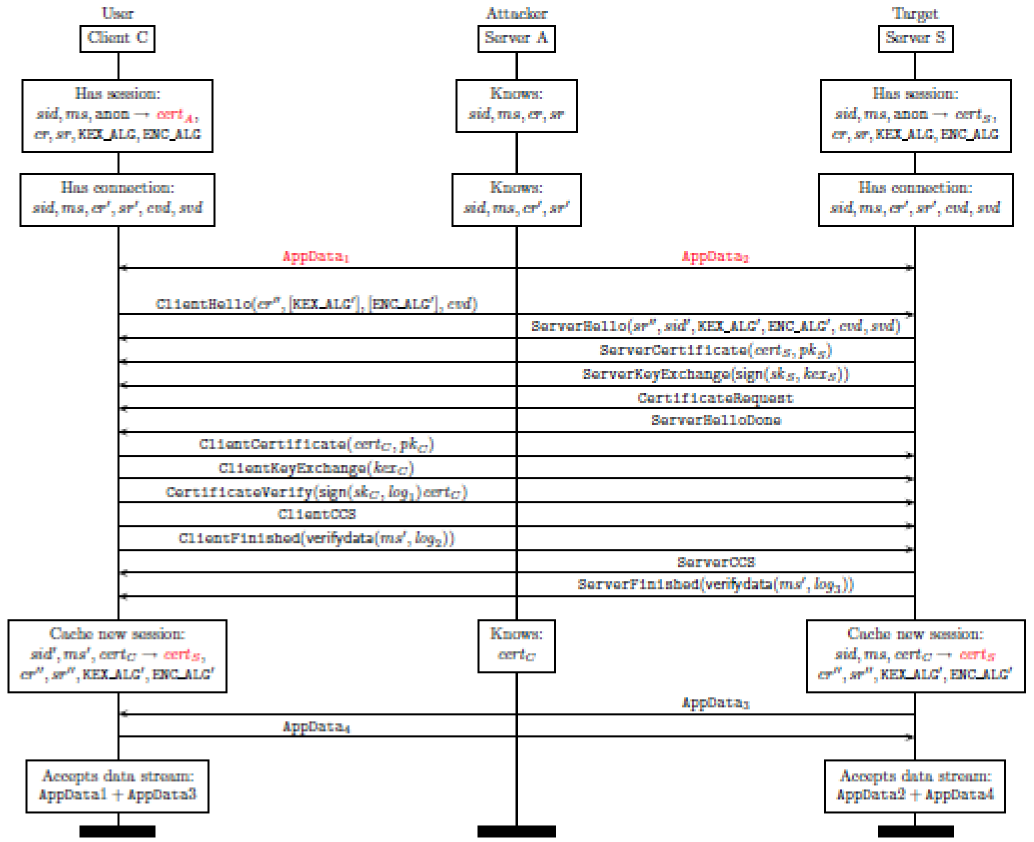


Application
HTTPS clients & servers

Triple Handshake Attack





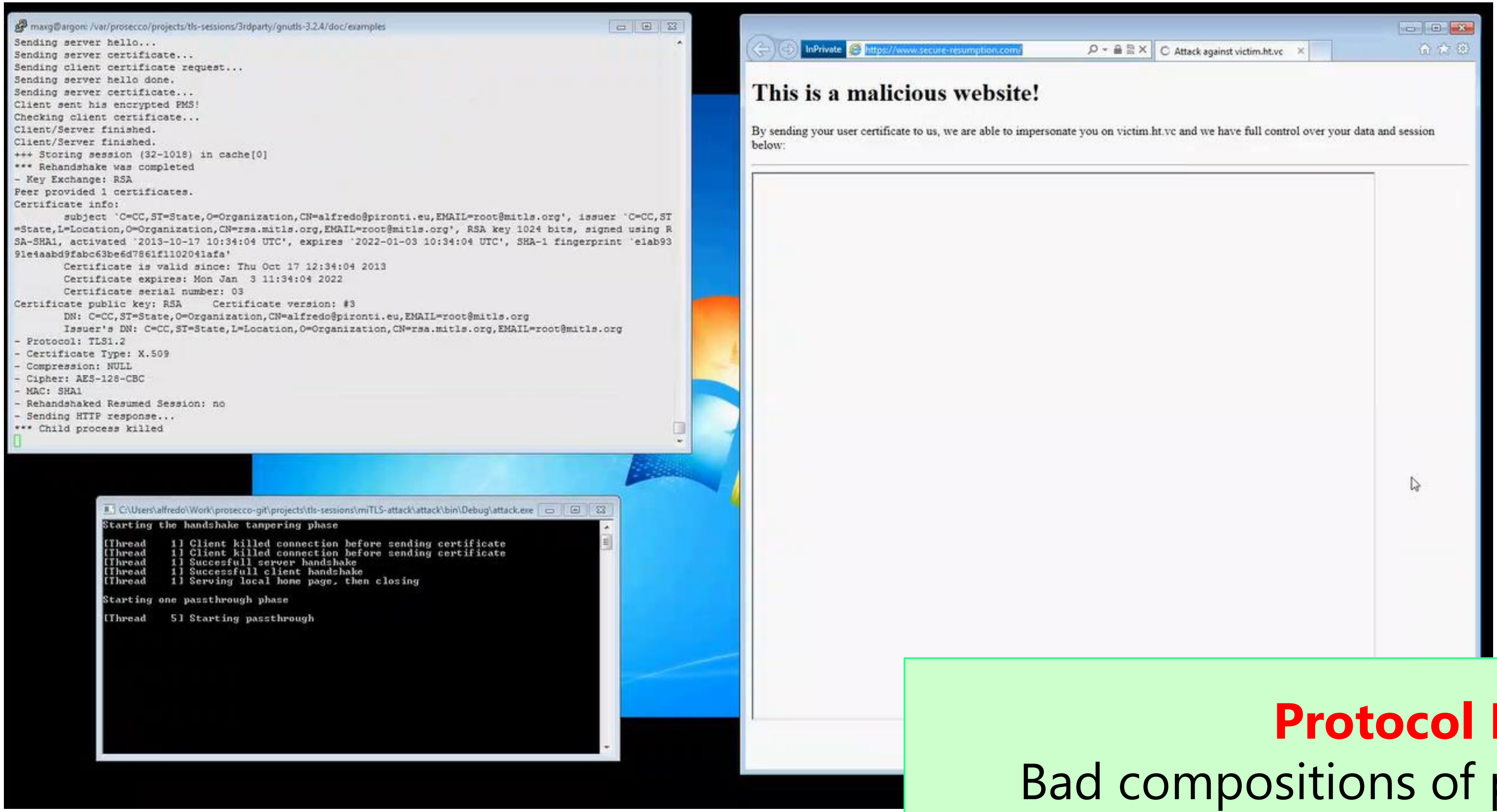


Protocol Logic

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
Git / CURL	GnuTLS

Protocol Logic

Bad compositions of protocol features

Infrastructure
certificate management (PKI)

TLS DESIGN

Protocol Logic
e.g. ambiguous messages
- cause clients and server to negotiate weak sessions

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

IEEE Security & Privacy 2013

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