Skip, Freak, and Logjam:

Finding and Preventing attacks on TLS

http://smacktls.com
http://weakdh.org
http://mitls.org

Karthikeyan Bhargavan
+ many, many others.
(INRIA, LORIA, Microsoft Research, IMDEA,
Univ of Pennsylvania, Univ of Michigan, JHU)

INVENTORS FOR THE DIGITAL WORLD

Microsoft Research - Inria
JOINT CENTRE

INRIA Prosecco

Our goal is to verify implementations of mainstream cryptographic protocols

- Computational model of cryptography
- Semi-automated verification tools
- Account for messy details of protocol in practice

This talk: new proofs and attacks on TLS

- miTLS: formal security theorems [Crypto'14]
- Skip, Freak: state machine attacks [Oakland'15]
- Logjam: imperfect forward secrecy (submitted)
- How to reduce the gap between formal theorems and concrete attacks?

Transport Layer Security (1994—)

The default secure channel protocol?

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

20 years of attacks, and fixes

1994 Netscape's Secure Sockets Layer
1996 SSL3
1999 TLS1.0 (RFC2246)
2006 TLS1.1 (RFC4346)
2008 TLS1.2 (RFC5246)

2015 TLS1.3?

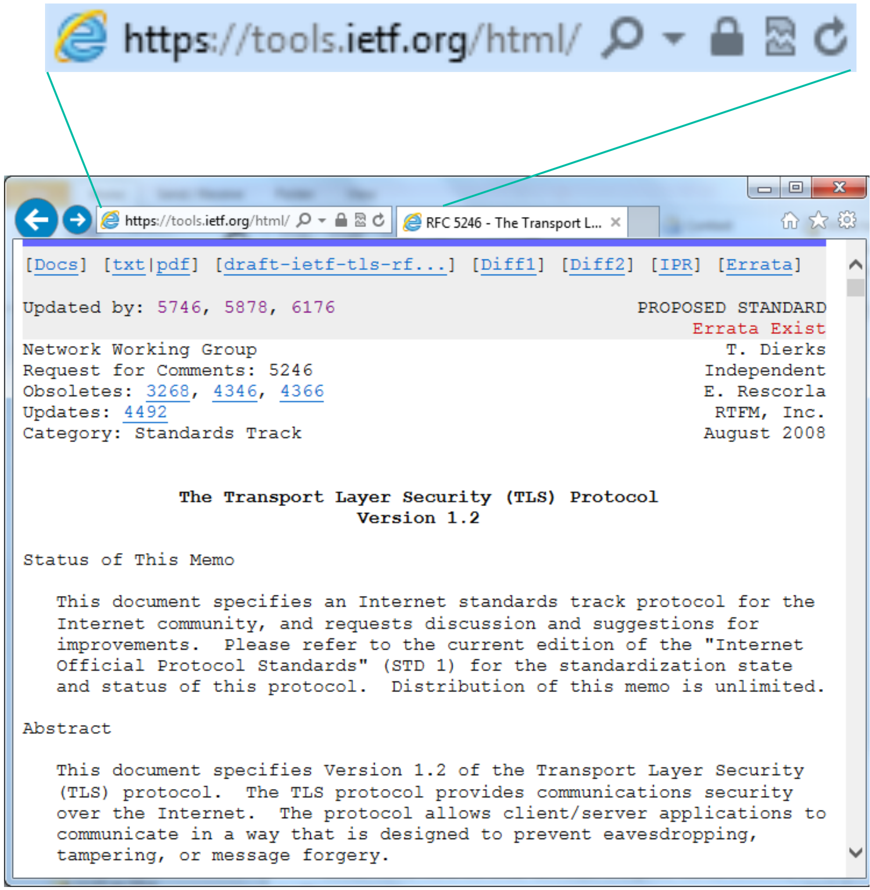
Many implementations

OpenSSL, SecureTransport, NSS, SChannel, GnuTLS, JSSE, PolarSSL, …

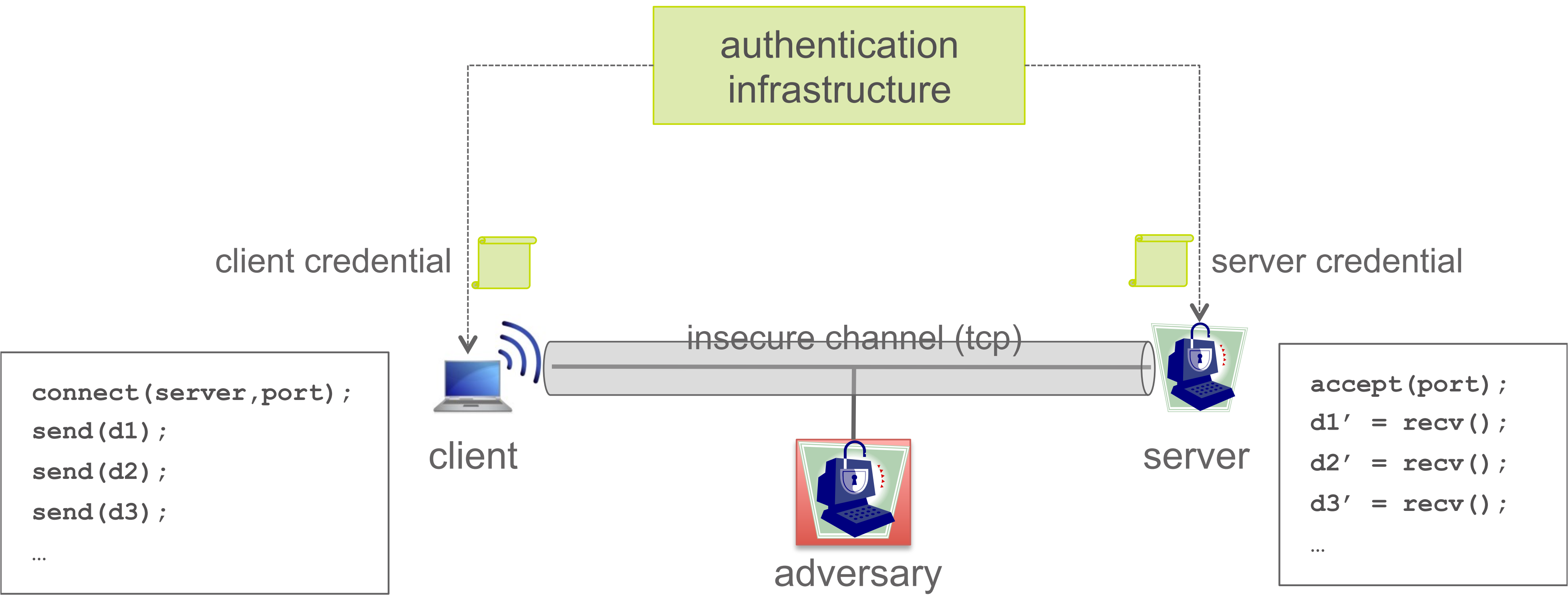
many bugs, attacks, patches every year

Many security theorems

mostly for small simplified models of TLS



Goal: a secure channel

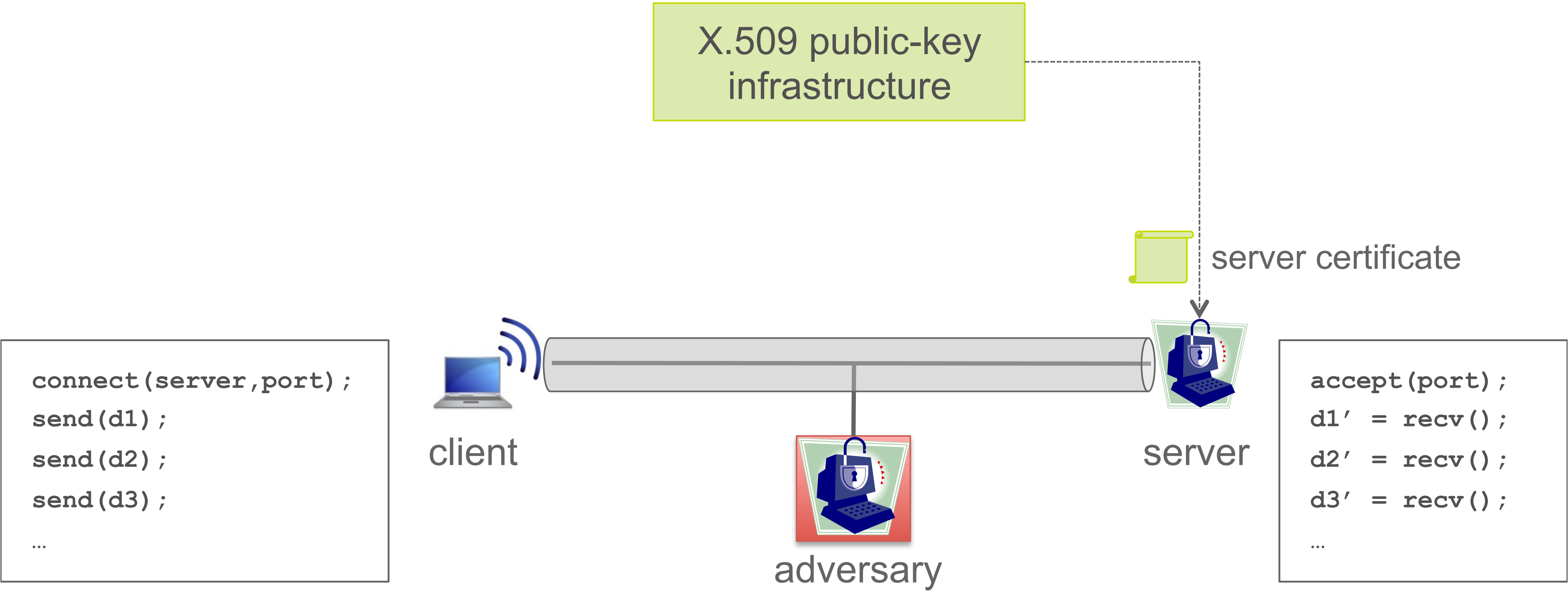


Security Goal: A network attacker cannot

- Impersonate the client or the server or inject data (authenticity)
- Distinguish the data stream from random bytes (confidentiality)

More formally: ACCE [Jager et al. ’11] based on sLHAE [Paterson et al ’11]

Secure channels for the Web

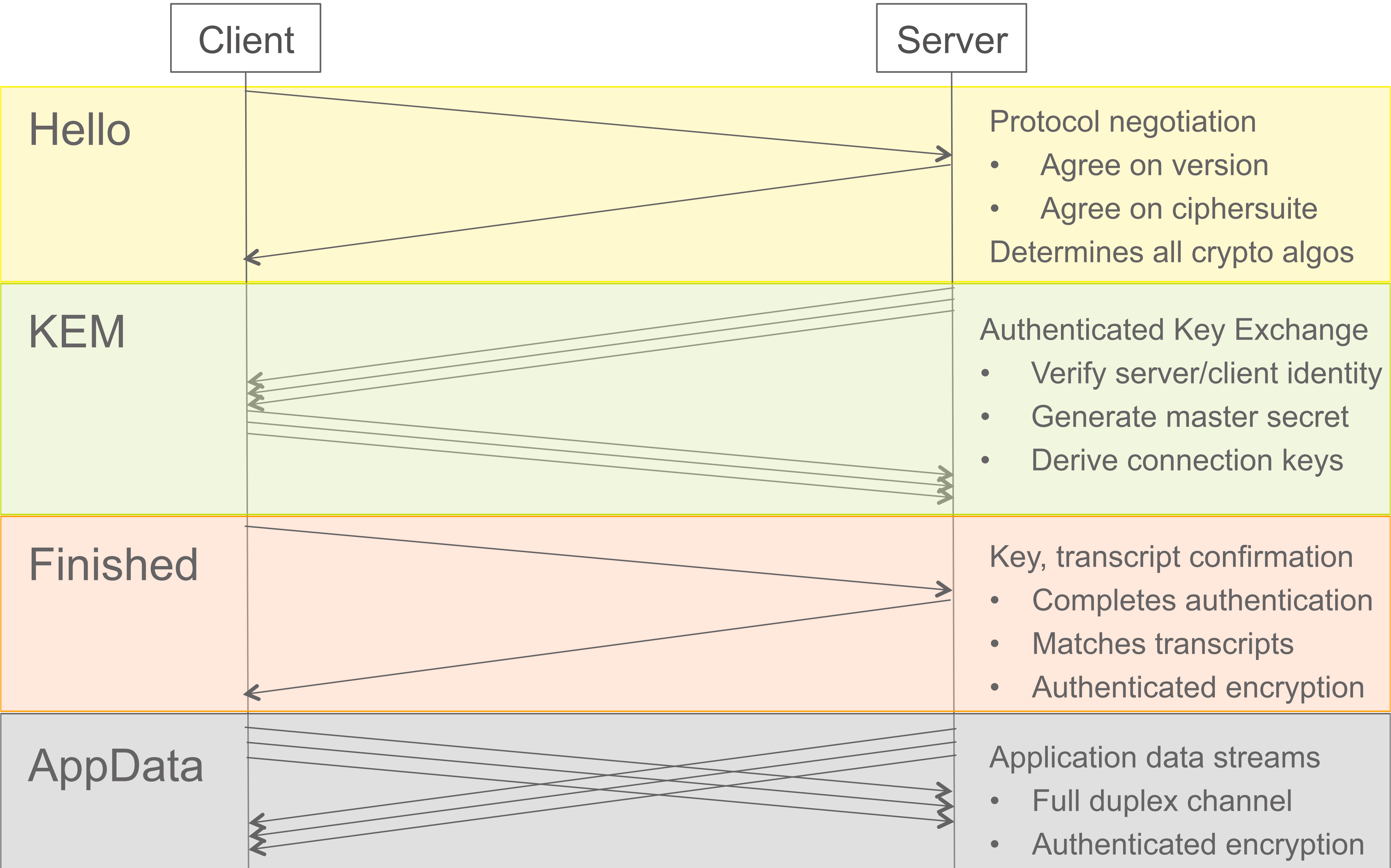


Security Goal: A network attacker cannot

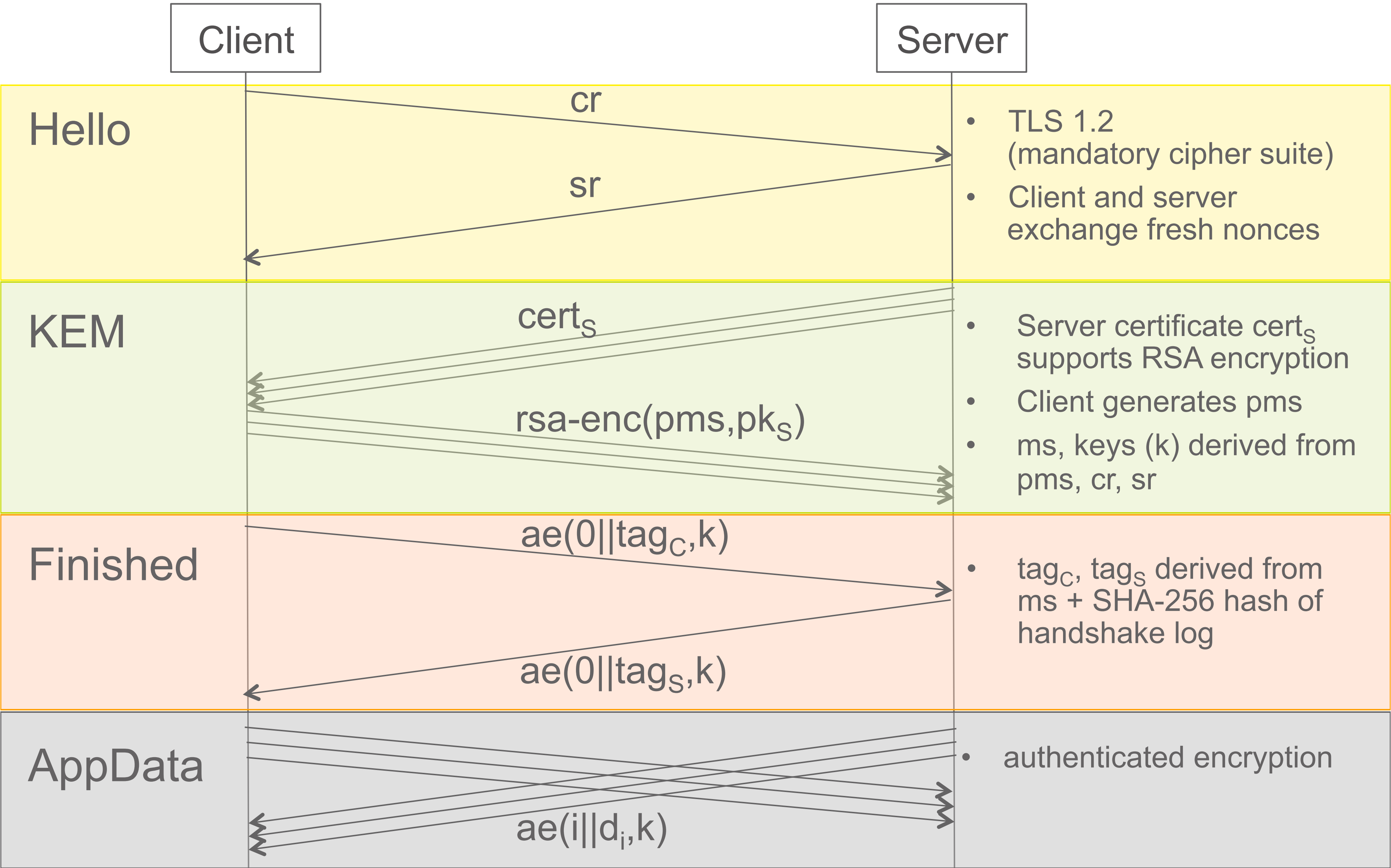
- Impersonate the server or inject server data (authenticity)
- Distinguish user data from random bytes (confidentiality)

More formally: SACCE [Krawczyk et al. '13] + sLHAE

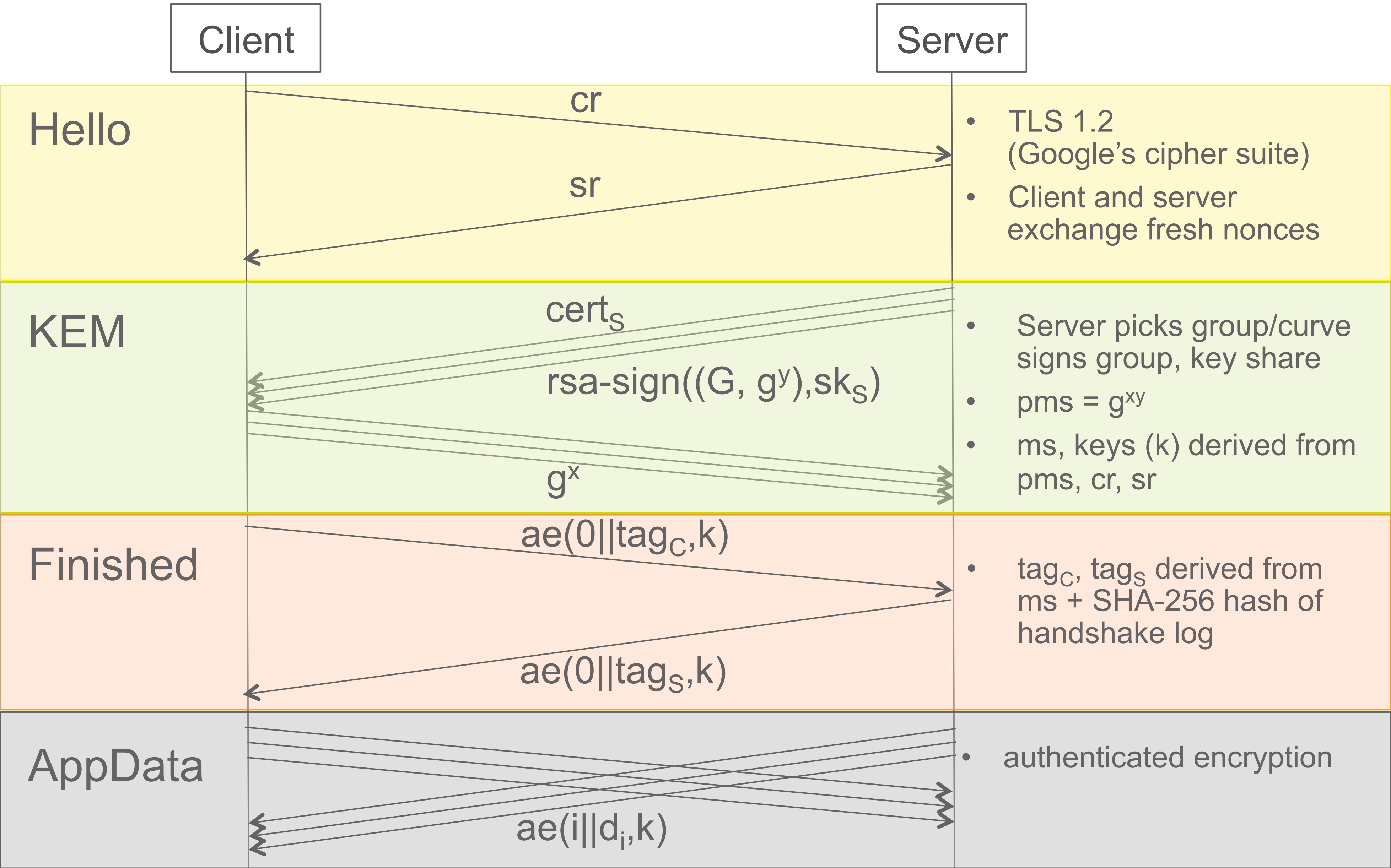
TLS protocol overview



RSA Key Transport



(EC)DHE Key Exchange



Cryptographic weaknesses

Many obsolete crypto constructions

- RSA encryption with PKCS#1 v1.5 padding *(Bleichenbacher)*
- MAC-then-Pad-then-Encrypt with AES-CBC *(Padding oracle)*
- Compress-then-MAC-then-Pad-then-Encrypt *(CRIME)*
- Chained IVs in TLS 1.0 AES-CBC *(BEAST)*
- RC4 key biases

Countermeasures

- Disable these features: SSL3, compression, RC4
- Implement ad-hoc mitigations very very carefully:
  - empty fragment to initialize IV for TLS 1.0 AES-CBC
  - constant time mitigation for Bleichenbacher attacks
  - constant-time plaintext length-hiding HMAC to prevent Lucky 13

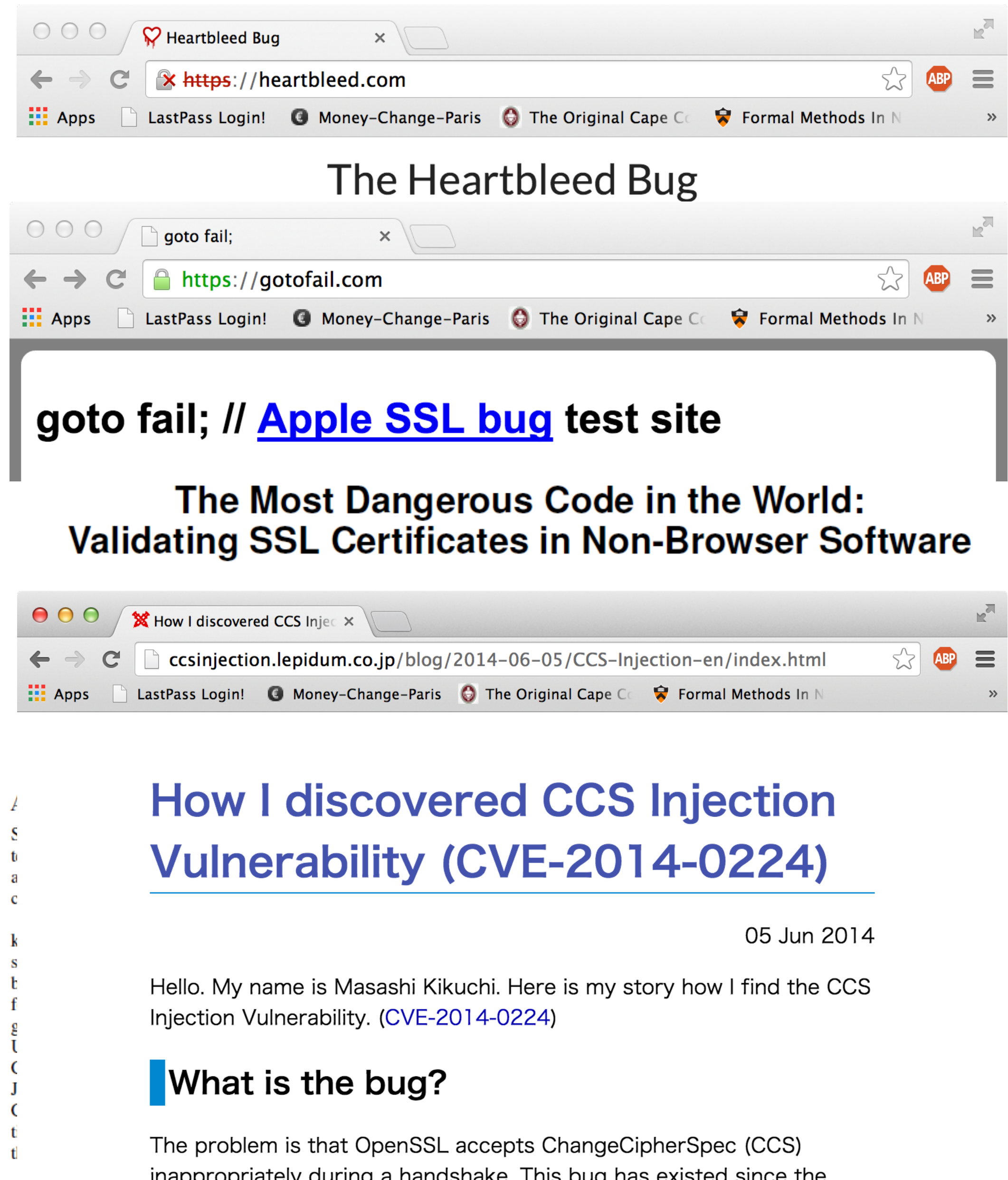
Other implementation challenges

Memory safety
Buffer overruns leak secrets

Missing checks
Forgetting to verify signature/MAC/certificate bypasses crypto guarantees

Certificate validation
ASN.1 parsing, wildcard certificates

State machine bugs
Most TLS implementations don't conform to spec
Unexpected transitions break protocol (badly)

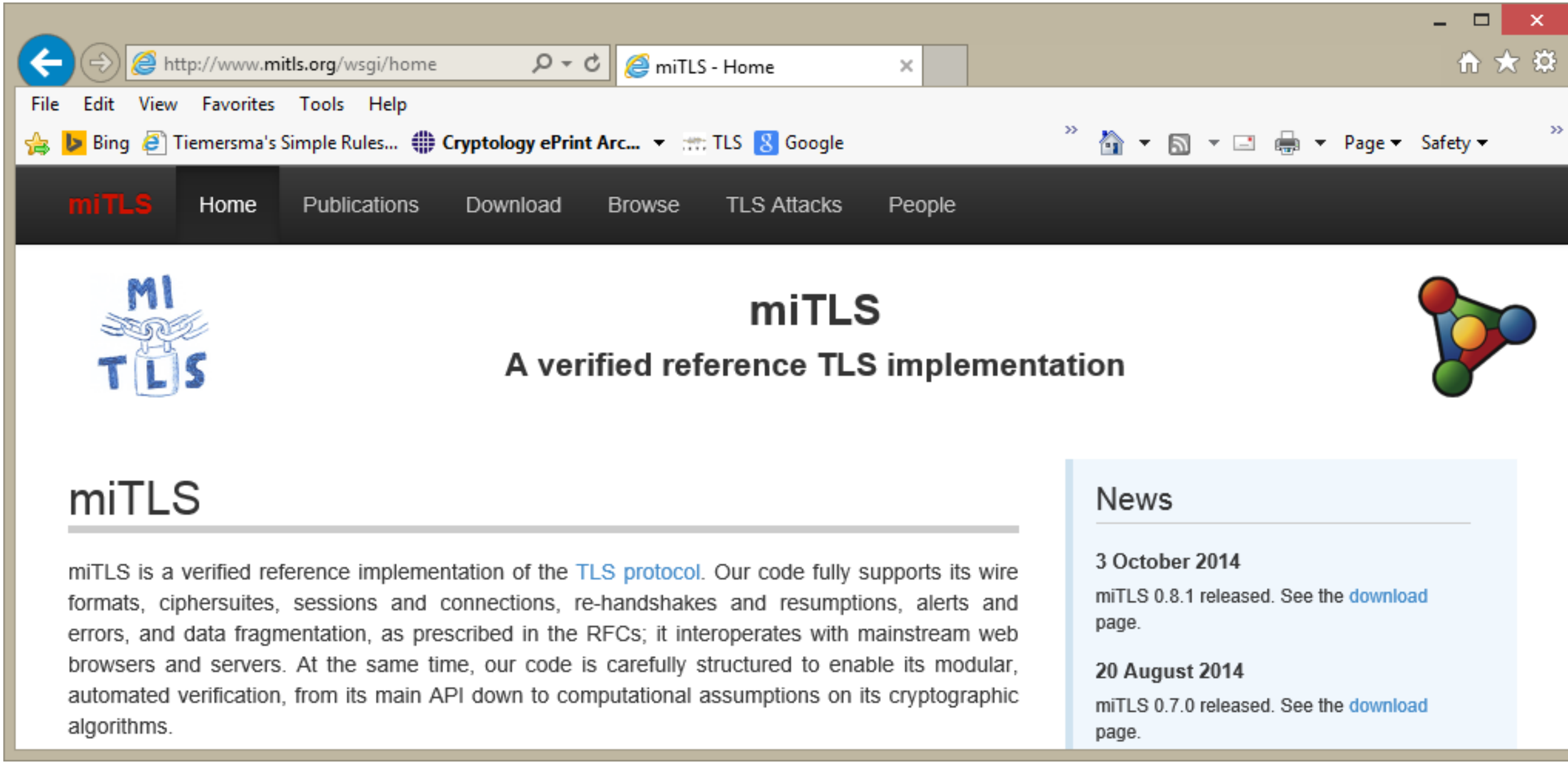


Implementing TLS correctly

Use formal methods!

- Use a type-safe programming language
  - **F#**, OCaml, Java, C#,…
  - (No buffer overruns, no Heartbleed)
- Verify the logical correctness of your code
  - Use a software verifier: **F7/F***, Why3, Boogie, Frama-C,…
- Link software invariants to cryptographic guarantees
  - Use a crypto verifier: **EasyCrypt**, CryptoVerif, ProVerif
  - Hire a cryptographer!

miTLS: a verified implementation



- How does this verification link to crypto assumptions and the secure channel goal?

State of the art

[Jager et al. '11] Security for TLS-DHE + authenticated encryption in the standard model

Monolithic proof (ACCE model), does not cover TLS-RSA

[Krawczyk, Paterson, Wee '13] Security for TLS-DHE + TLS-RSA + authenticated encryption

KEM abstraction (SACCE model), single ciphersuite, does not cover resumption, renegotiation

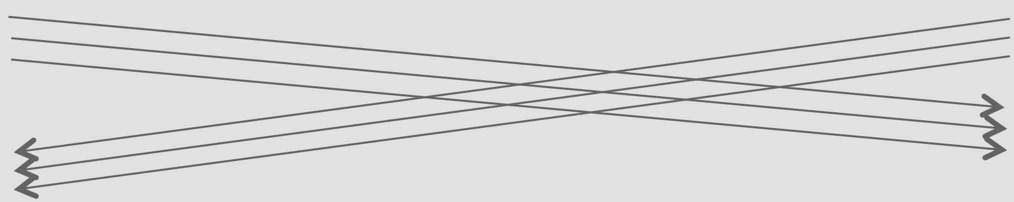
[Bhargavan et al. ’14] Comprehensive modular treatment of a TLS handshake implementation

Multi-ciphersuite, multi-handshake security

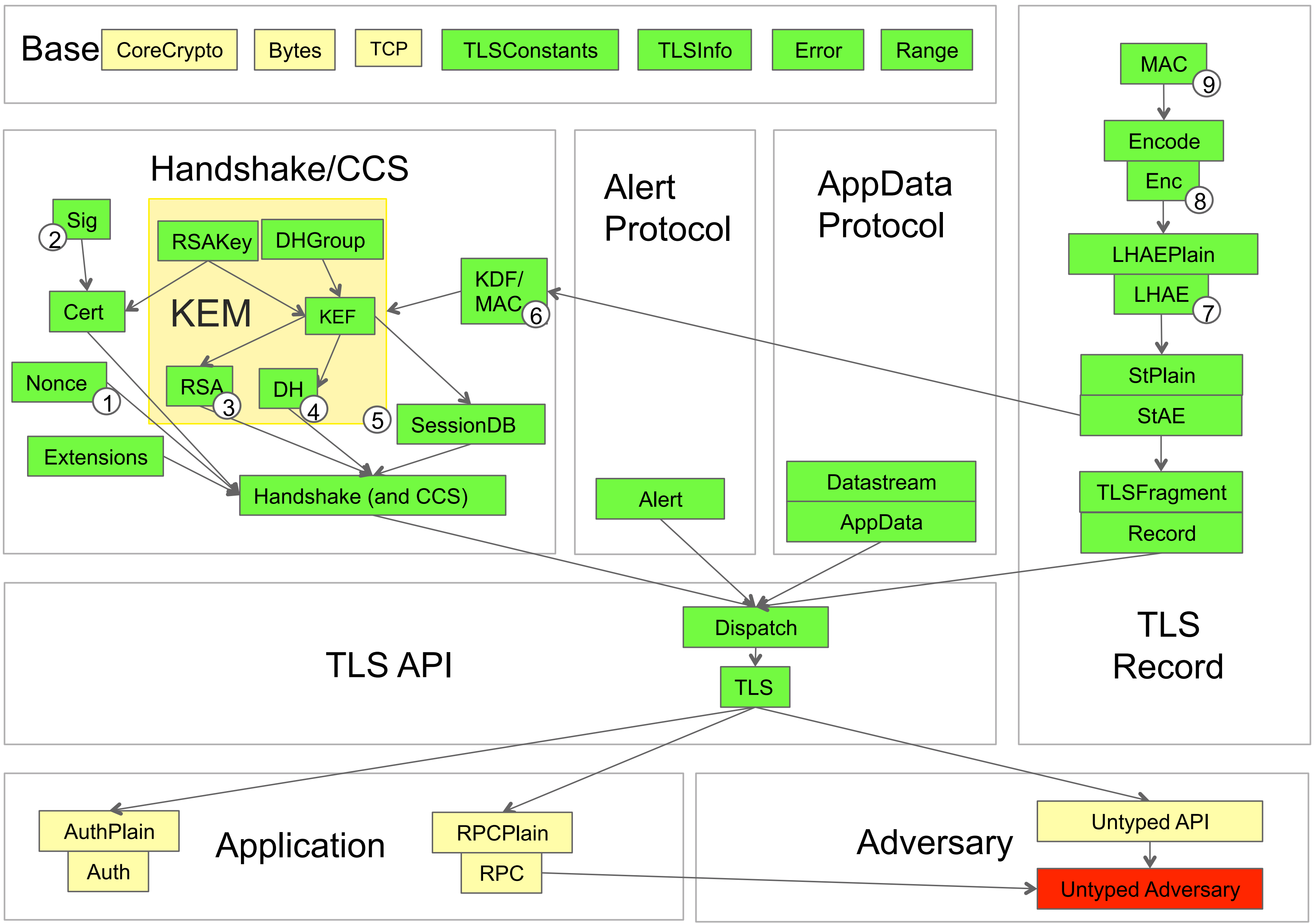
Cryptographic core of TLS

Client		Server
$\ell_C \leftarrow \$;$	—— `ClientHello`$[\ell_C]$ ⟶	$\ell_S \leftarrow \$;\ \ell := \ell_C \,\|\, \ell_S;$
	`ServerHello`$[\ell_S]$	
$\ell := \ell_C \,\|\, \ell_S;$	← `ServerCertificate`$[certs_S]$ —	
$pk := \mathsf{pk}(certs_S)$	`ServerHelloDone`	
$c, ms \leftarrow \mathsf{Enc}(pk, \ell)$		
$k := \mathsf{KDF}(ms, \ell)$	—— `ClientKeyExchange`$[c]$ ⟶	$ms \leftarrow \mathsf{Dec}(sk, \ell, c)$
$log_C := \langle \text{all prior messages} \rangle$		
$tag_C := \mathsf{MAC}(ms, \text{“C”}, log_C)$	—— `ClientFinished`$[tag_C]$ ⟶	$log_C := \langle \text{all prior messages} \rangle$
		$tag_C \stackrel{?}{=} \mathsf{MAC}(ms, \text{“C”}, log_C)$
		$k := \mathsf{KDF}(ms, \ell)$
$log_S := \langle \text{all prior messages} \rangle$		$log_S := \langle \text{all prior messages} \rangle$
$tag_S \stackrel{?}{=} \mathsf{MAC}(ms, \text{“S”}, log_S)$	⟵ `ServerFinished`$[tag_S]$ ——	$tag_S := \mathsf{MAC}(ms, \text{“S”}, log_S)$

AppData



miTLS concrete implementation



miTLS API & ideal functionality (outline)

Standard socket API with embedded security specification

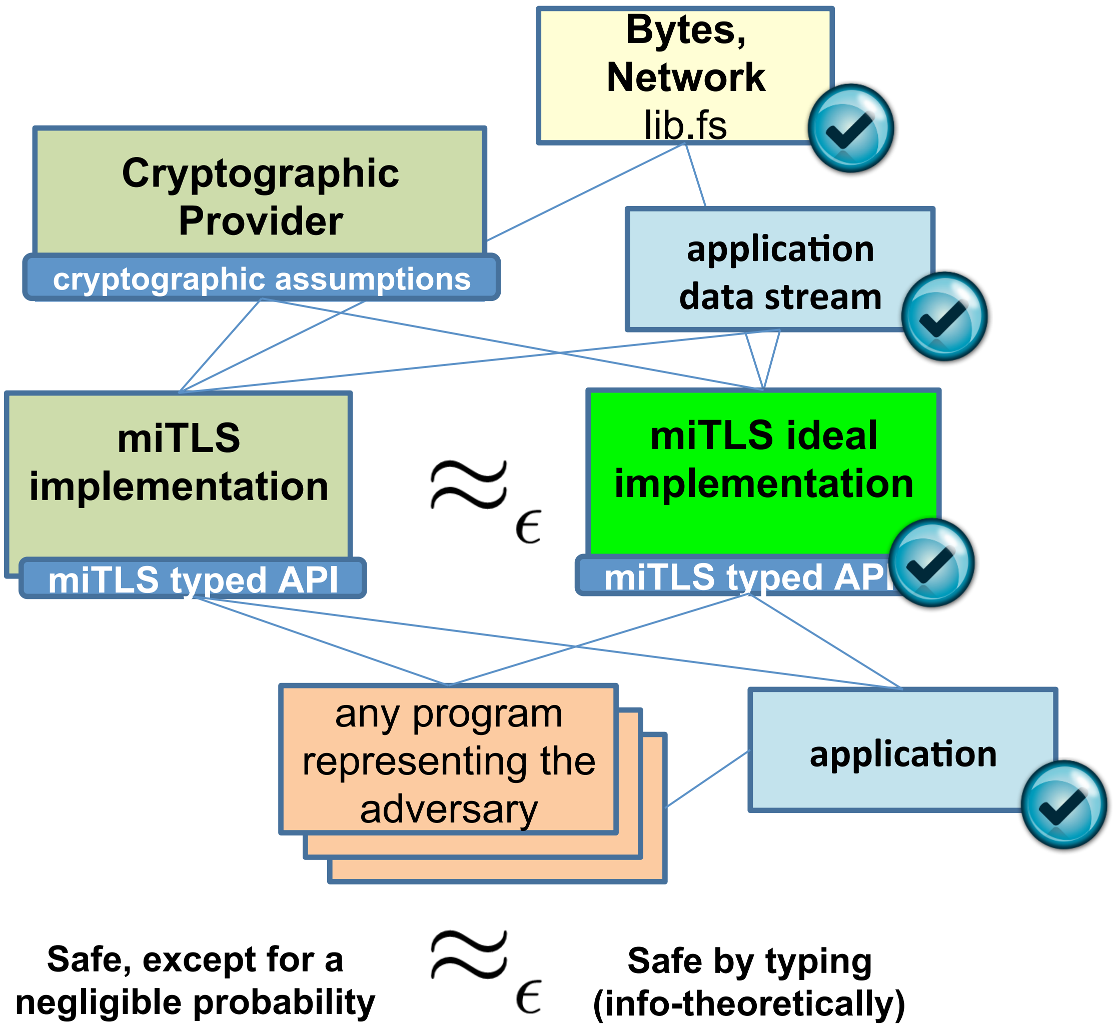
- Abstract types for confidentiality (a la information flow)
- Refinements for authenticity (a la contracts/ pre-/post-conditions)

```
type Connection // for each local instance of the protocol
type (;c:Connection) AppData

// creating new client and server instances
val connect: TcpStream -> Params -> Connection
val accept:  TcpStream -> Params -> Connection

// reading data
type (;c:Connection) IOResult_i =
| Read      of c':Connection * data:(;c) AppData
| CertQuery of c':Connection
| Complete  of c':Connection { Agreement(c') }
| Close     of TcpStream
| Warning   of c':Connection * a:AlertDescription
| Fatal     of a:AlertDescription
val read : c:Connection -> (;c) IOResult_i

// writing data
type (;c:Connection,data:(;c) AppData) IOResult_o =
| WriteComplete of c':Connection
| WritePartial  of c':Connection * rest:(;c') AppData
| MustRead      of c':Connection
val write: c:Connection -> data:(;c) AppData -> (;c,data) IOResult_o

// triggering new handshakes, and closing connections
val rehandshake: c:Connection -> Connection Result
val request:     c:Connection -> Connection Result
val shutdown:    c:Connection -> TcpStream Result
```

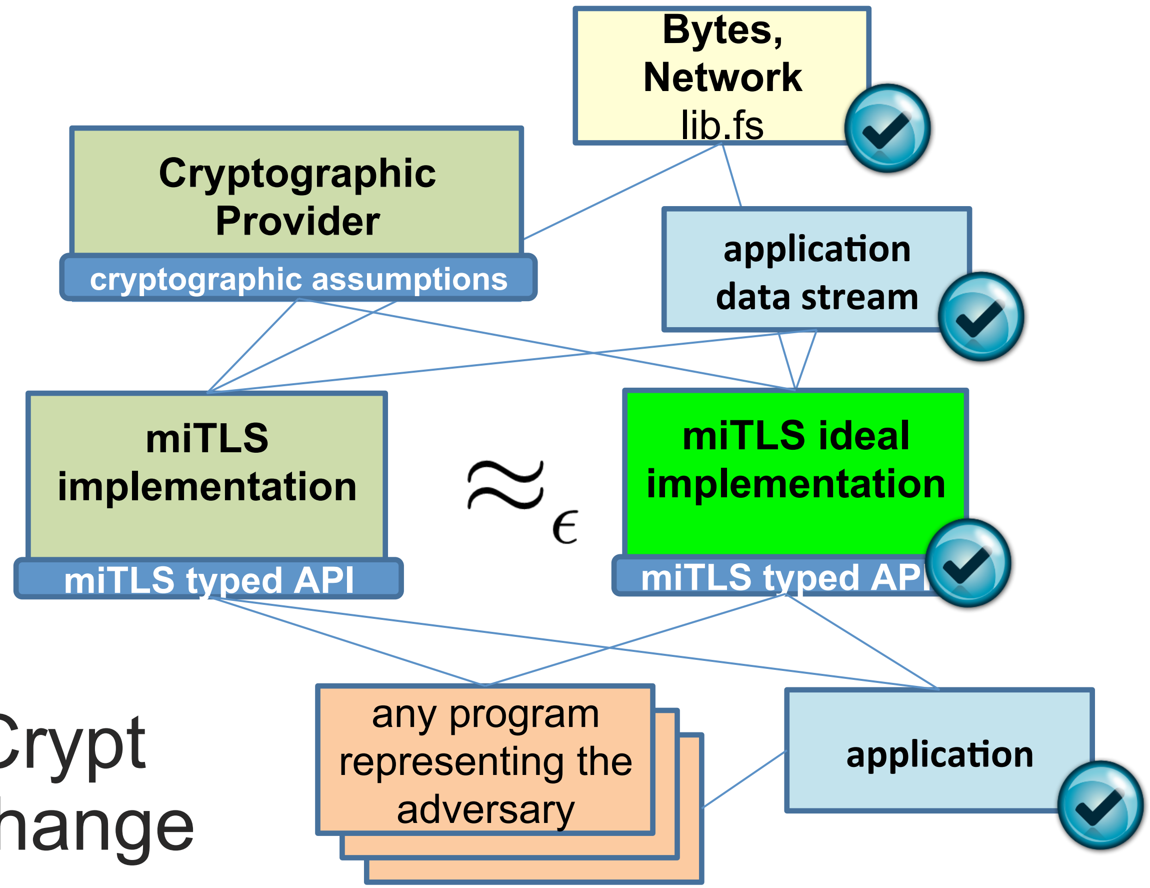
Security theorem

Main crypto result:
concrete TLS & ideal TLS are computationally indistinguishable

We prove that ideal miTLS meets its secure channel specification
using standard program verification (typing)



Security theorem



Proof automation

7,000 lines of F#
checked against
3,000 lines of F7
type annotations

+

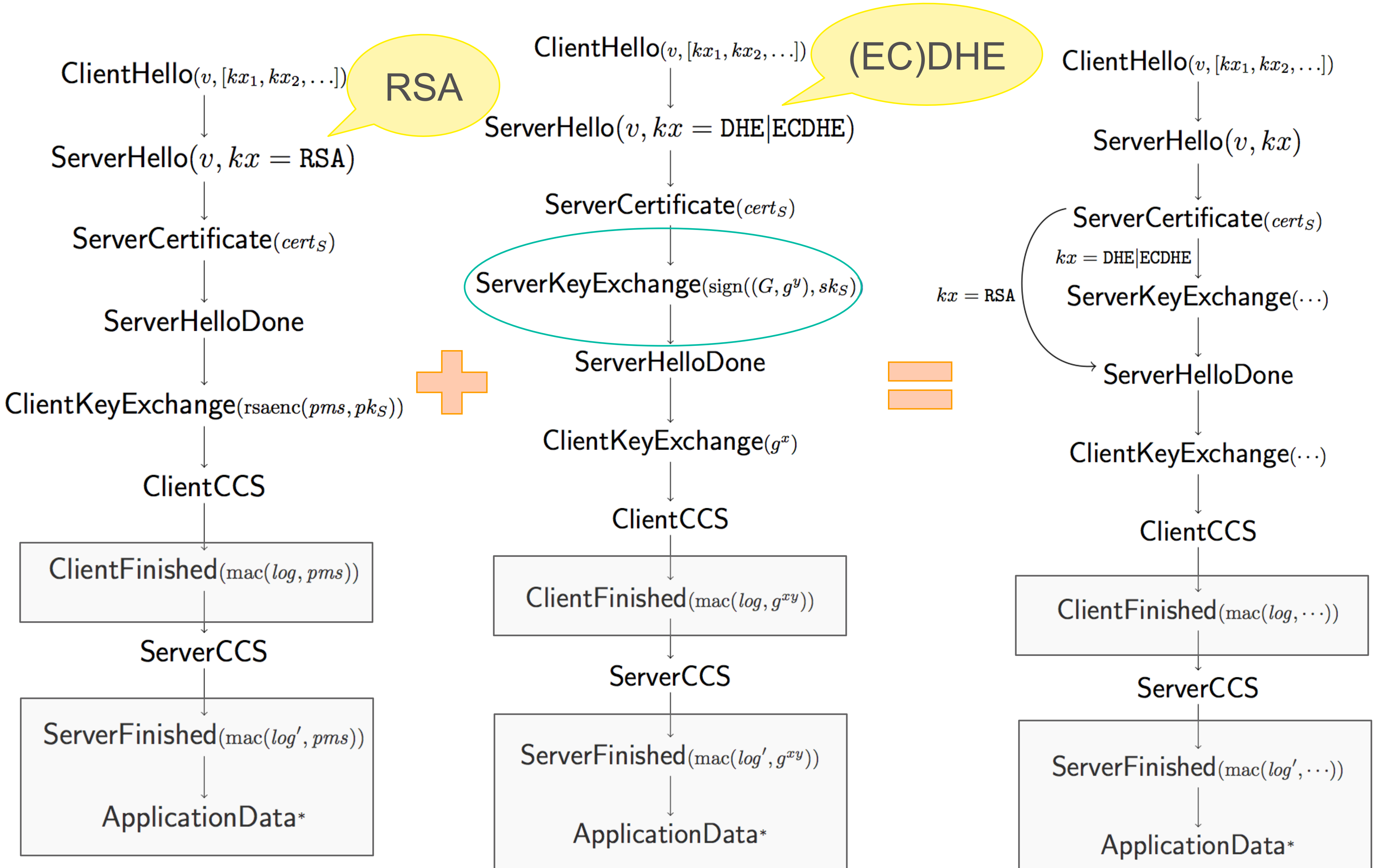
3,000 lines of EasyCrypt
for the core key exchange

Ongoing work

ECDHE, GCM, Certificates, Side-channels

Mission accomplished?

Composing Key Exchanges

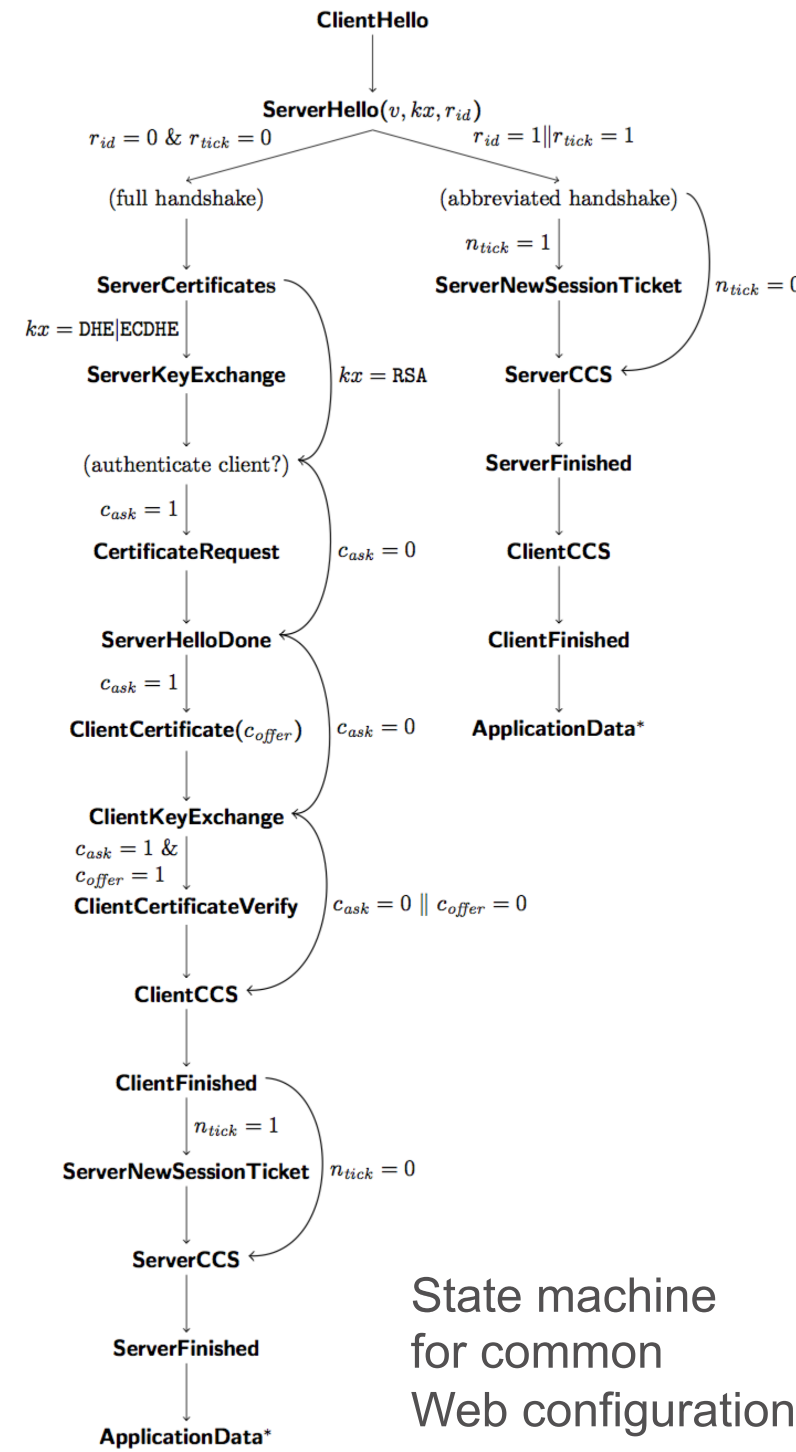


TLS State Machine

RSA + DHE + ECDHE
+ Session Resumption
+ Client Authentication

- Covers most features used on the Web
- Composition implemented and proved for miTLS [IEEE S&P'13, CRYPTO'14]
- Only works for reference code written for verification, in F# (dialect of OCaml)

Can this proof technique be applied to OpenSSL?

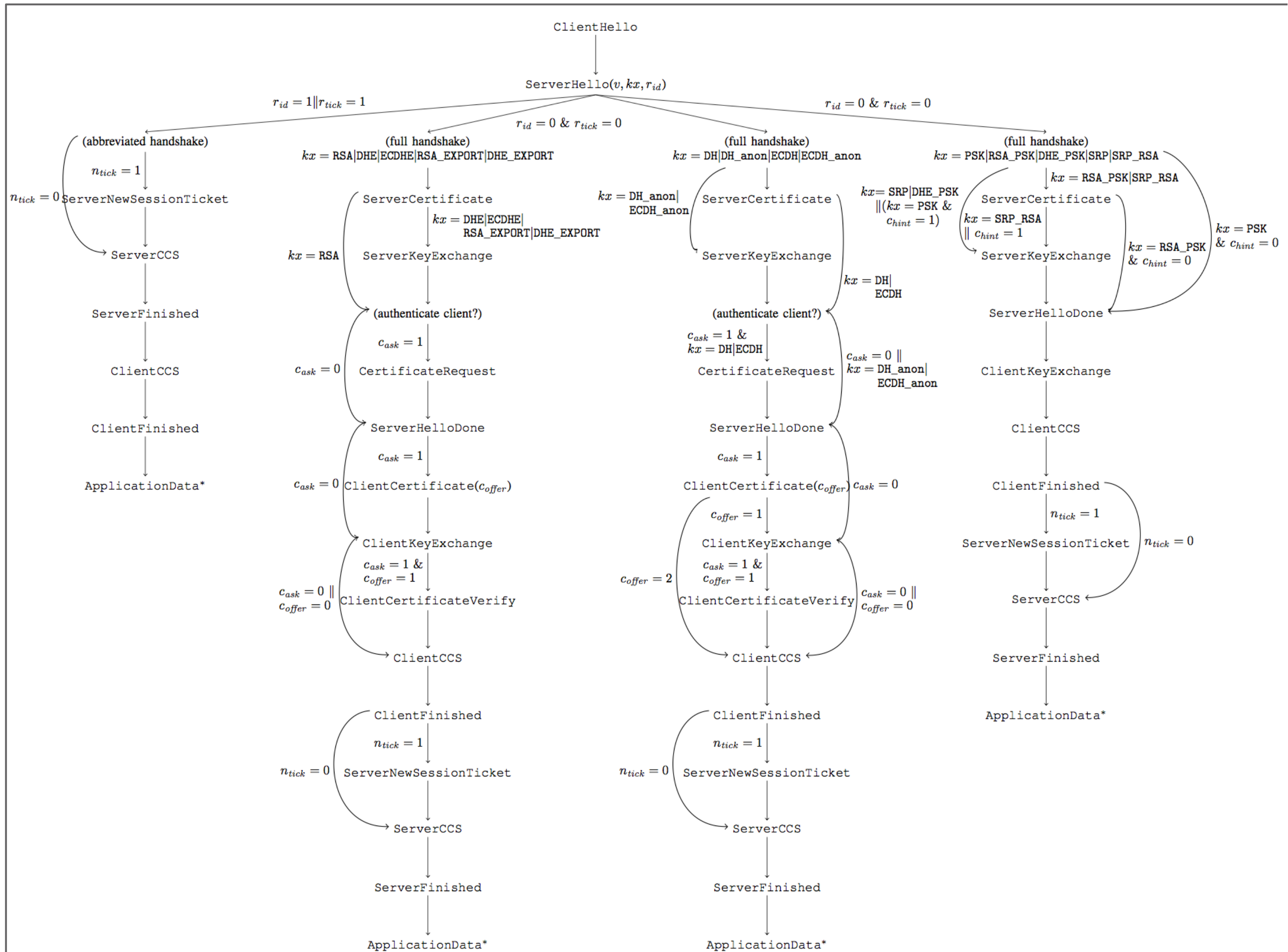


State machine for common Web configurations

OpenSSL State Machine

+ Fixed_DH
+ DH_anon
+ PSK
+ SRP
+ Kerberos
+ *_EXPORT
+ …

We cannot ignore all these because they share code/keys with RSA/DHE



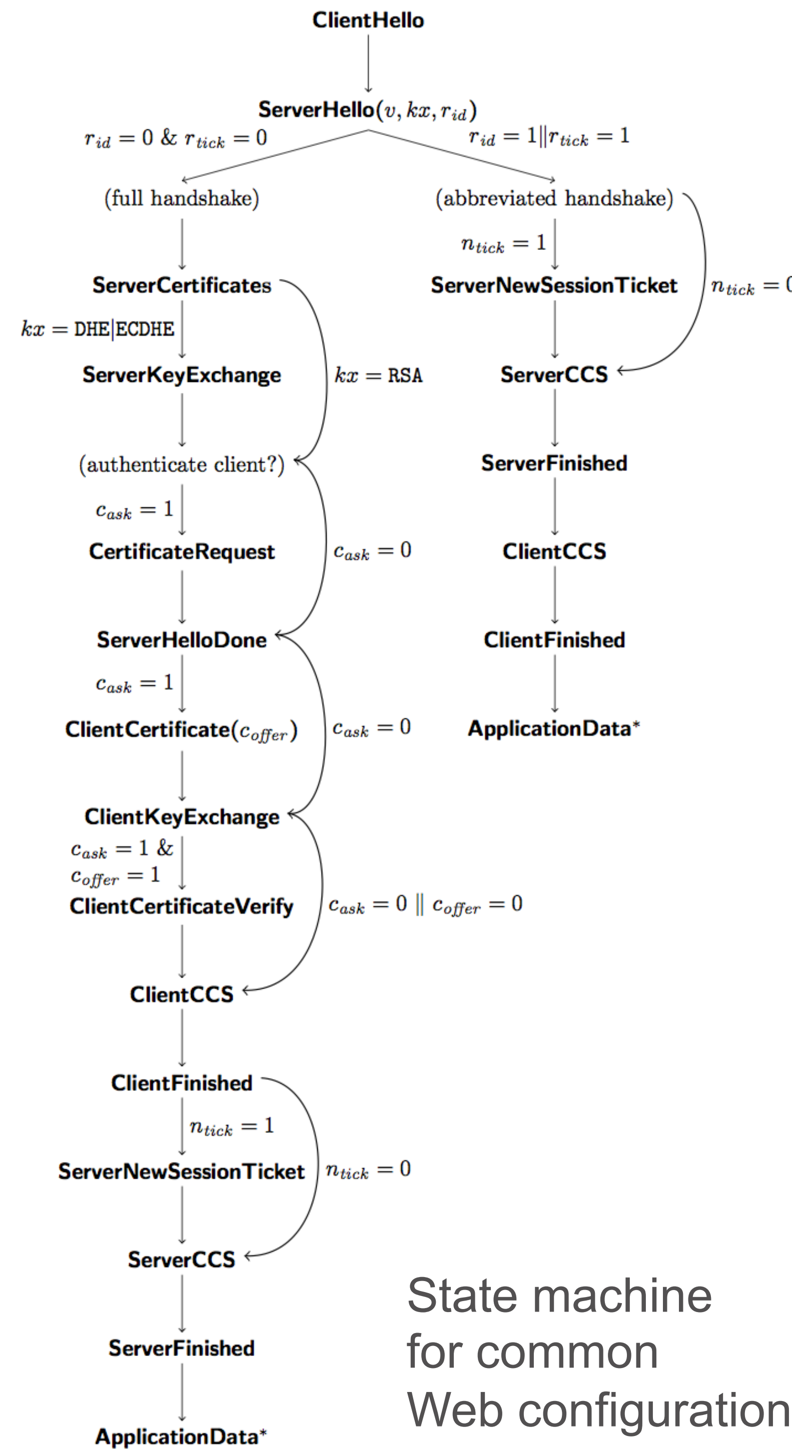
Fuzzing TLS

Does OpenSSL conform to the miTLS state machine?

- There are known attacks if it doesn't [EarlyCCS 2014]

We built a test framework

- FlexTLS, based on miTLS
- Generates 100s of non-conforming traces from a *state machine specification*
- We tested many TLS libraries



State machine for common Web configurations

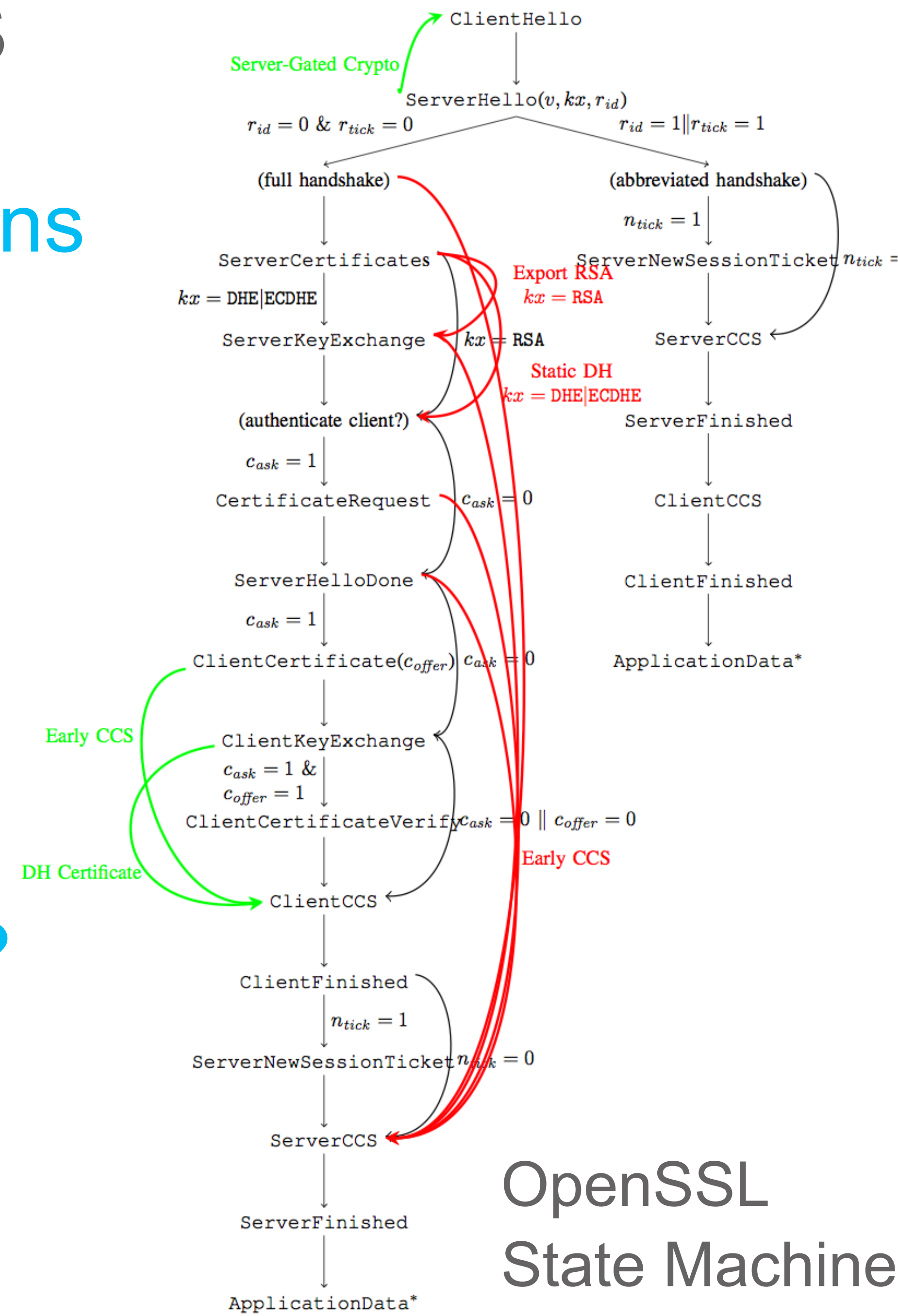
Many, Many Bugs

Unexpected state transitions in OpenSSL, NSS, Java, SecureTransport, …

- Required messages are allowed to be skipped
- Unexpected messages are allowed to be received
- CVEs for many libraries

How come all these bugs?

- In independent code bases, sitting in there for years
- Are they exploitable?



OpenSSL State Machine

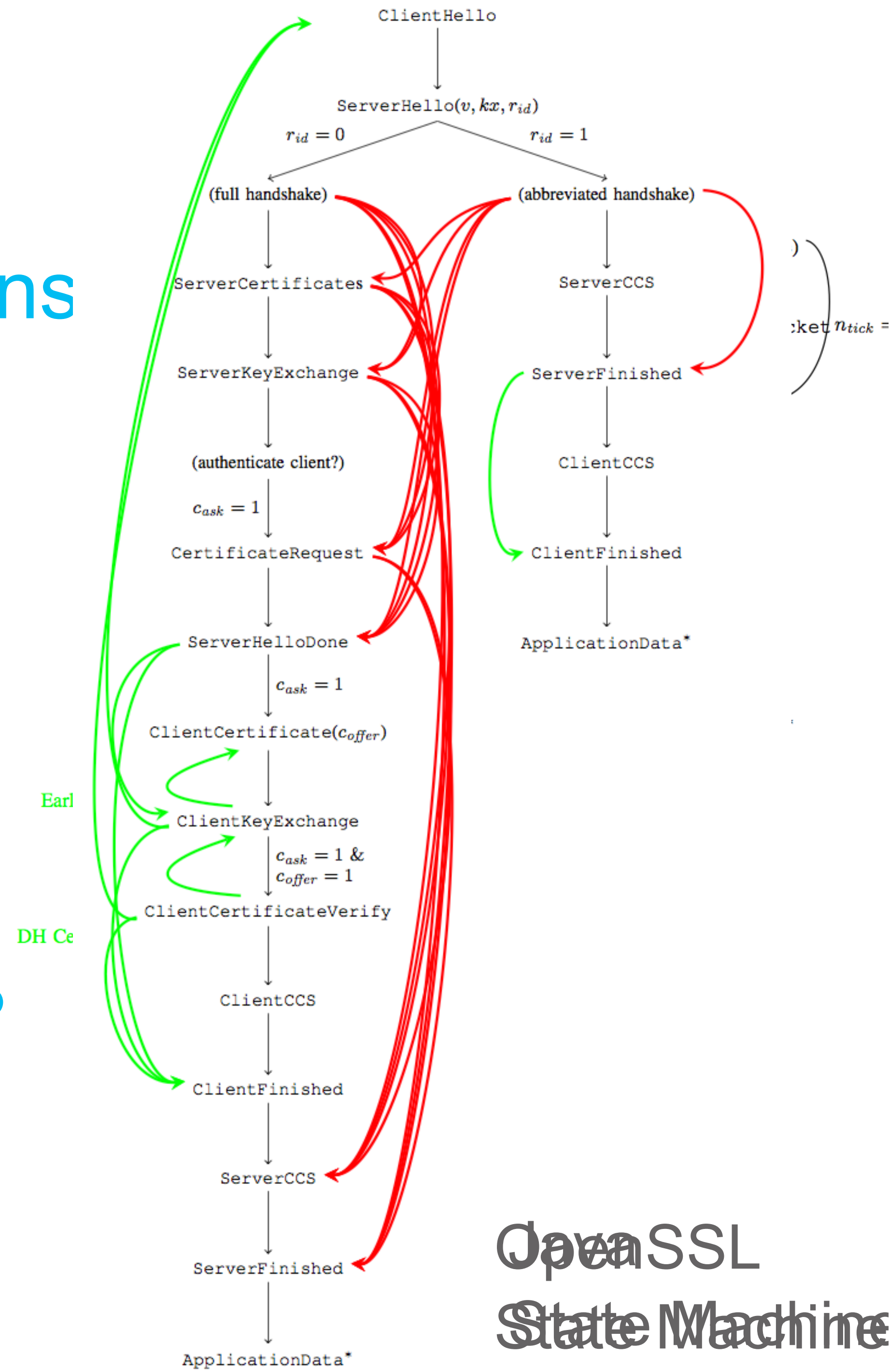
Many, Many Bugs

Unexpected state transitions in OpenSSL, NSS, Java, SecureTransport, …

- Required messages are allowed to be skipped
- Unexpected messages are allowed to be received
- CVEs for many libraries

How come all these bugs?

- In independent code bases, sitting in there for years
- Are they exploitable?



OpenSSL / Java State Machine

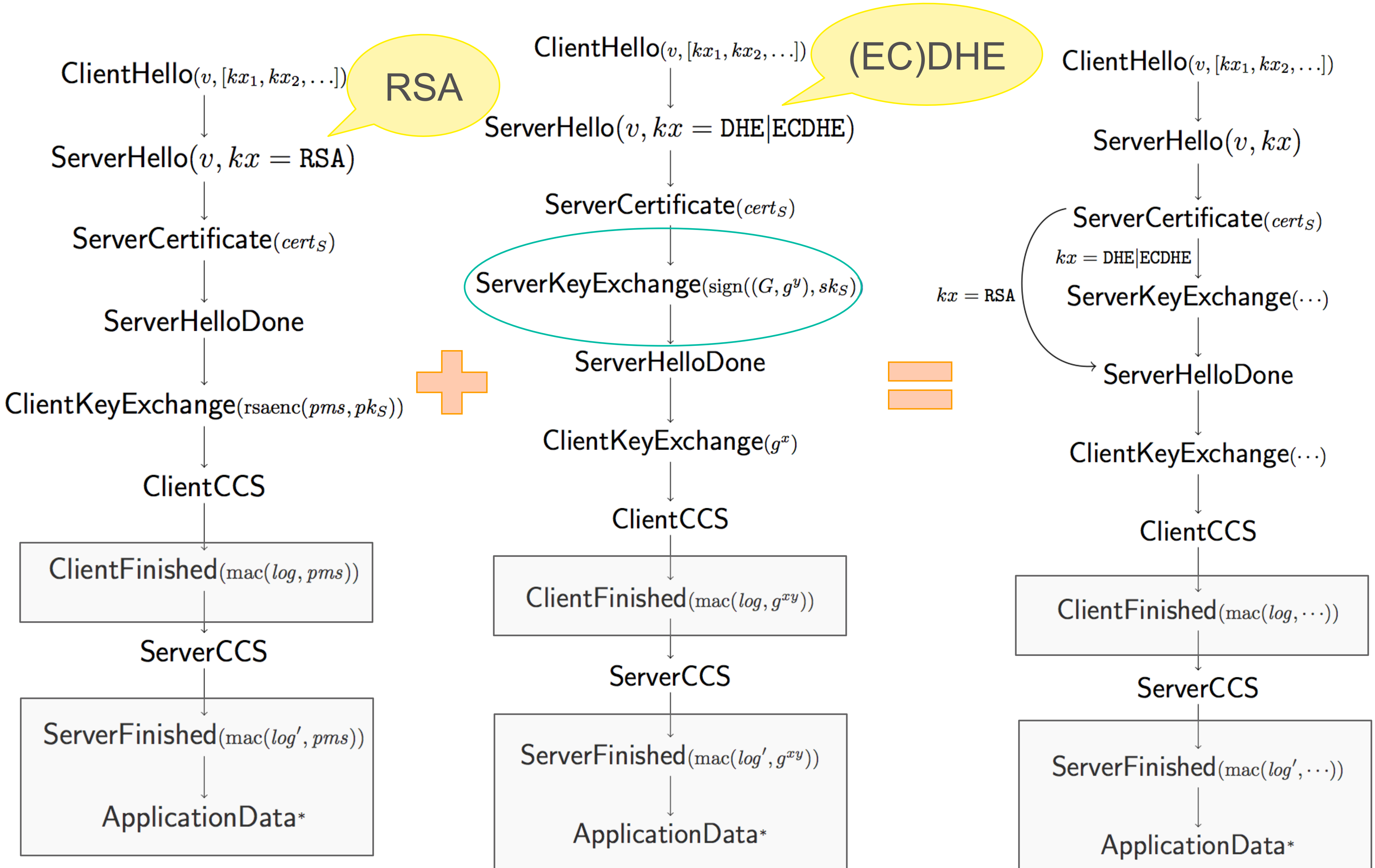
Culprit: Underspecified State Machine

TLS specifies a ladder diagram with optional messages

- Handshake ends with agreement on transcript

```
RFC 5246                          TLS                          August 2008

      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               ServerKeyExchange*
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                     -------->
                                               [ChangeCipherSpec]
                                   <--------             Finished
      Application Data             <------->     Application Data

             Figure 1.  Message flow for a full handshake
```

Composing Key Exchanges



Composing with Optional Messages

Treat ServerKeyExchange as optional

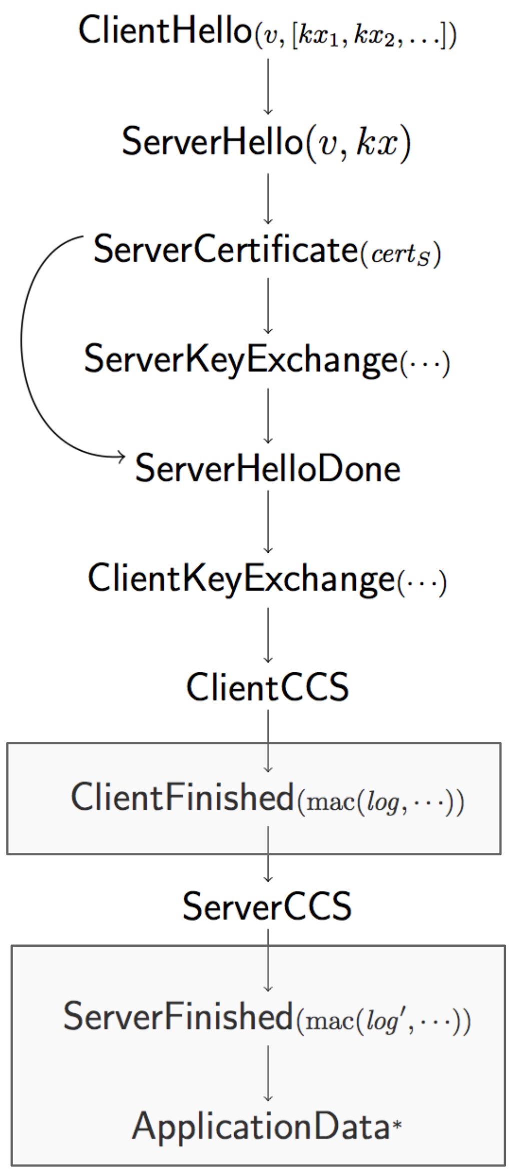
- Server decides to send it or not
- Client tries to handle both cases
- Consistent with Postel's principle:
  "*be liberal in what you accept*"

Unexpected cases at the client

- Server skips ServerKeyExchange in DHE
- Server sends ServerKeyExchange in RSA

Clients should reject these cases

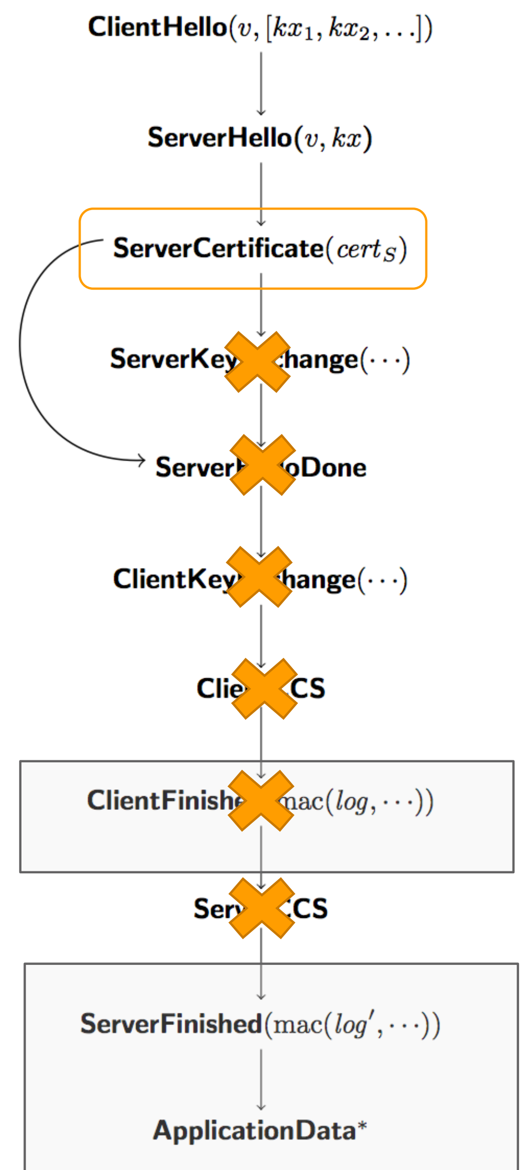
- In practice: clients accept and perform unexpected cryptographic computations, breaking the security of TLS



SKIP: Server Impersonation with DHE

Network attacker impersonates S.com to a Java TLS client

1. Send S's cert
2. SKIP ServerKeyExchange (bypass server signature)
3. SKIP ServerHelloDone
4. SKIP ServerCCS (bypass encryption)
5. Send ServerFinished using uninitialized MAC key (bypass handshake integrity)
6. Send ApplicationData (unencrypted) as S.com



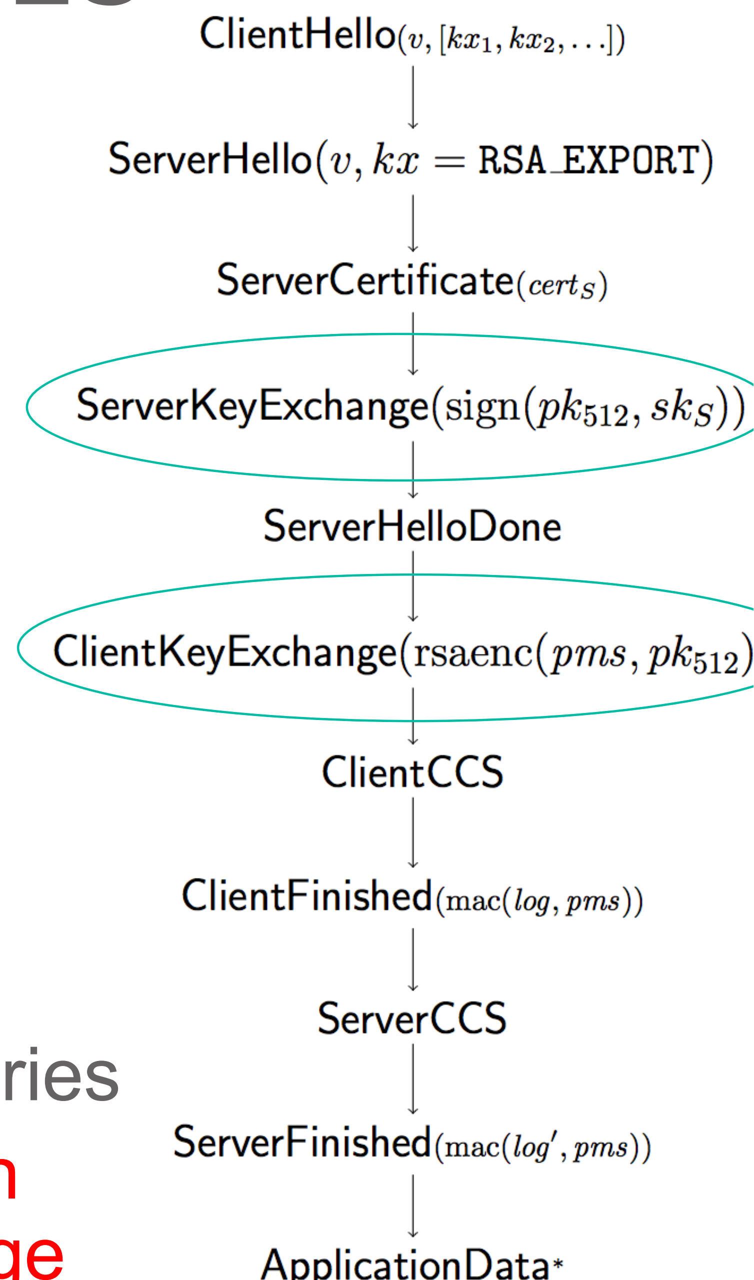
Export-Grade RSA in TLS

TLS 1.0 supported weakened ciphers to comply with export regulations in 1990s

- RSA keys limited to 512 bits
- Export keys are sent in a signed **ServerKeyExchange**
- Client uses the 512-bit key instead of S's public key

EXPORT deprecated in 2000

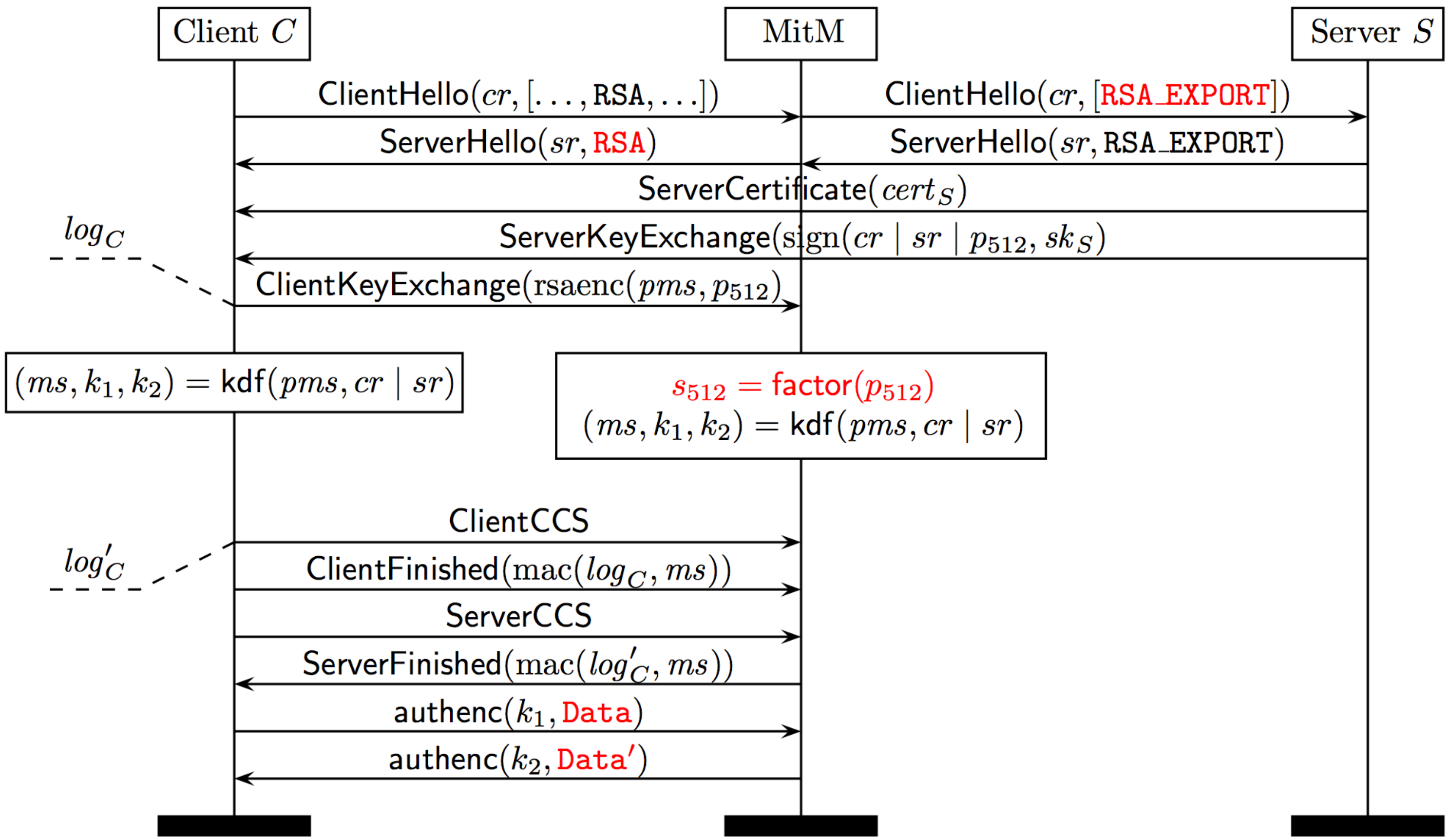
- (Dead) code still exists in OpenSSL and many other libraries
- Can be triggered by sending an unexpected ServerKeyExchange



FREAK: Downgrade to RSA_EXPORT

A man-in-the-middle attacker can:

- impersonate servers that support RSA_EXPORT,
- at buggy clients that allow ServerKeyExchange in RSA



FREAK: Exploit and Impact

The Washington Post

The Switch

'FREAK' flaw undermines security for Apple and Google users, researchers discover

By **Craig Timberg** March 3

The Economist

Computer security

The law and unintended consequences

The perils of deliberately sabotaging security

Mar 7th 2015 | From the print edition

COMPUTERS are notoriously insecure. Usually, this is by accident rather than design. Modern operating systems contain millions of lines of code, with millions more in the applications that do the things people want done. Human brains are simply too puny to build something so complicated without making mistakes.

BBC NEWS

TECHNOLOGY

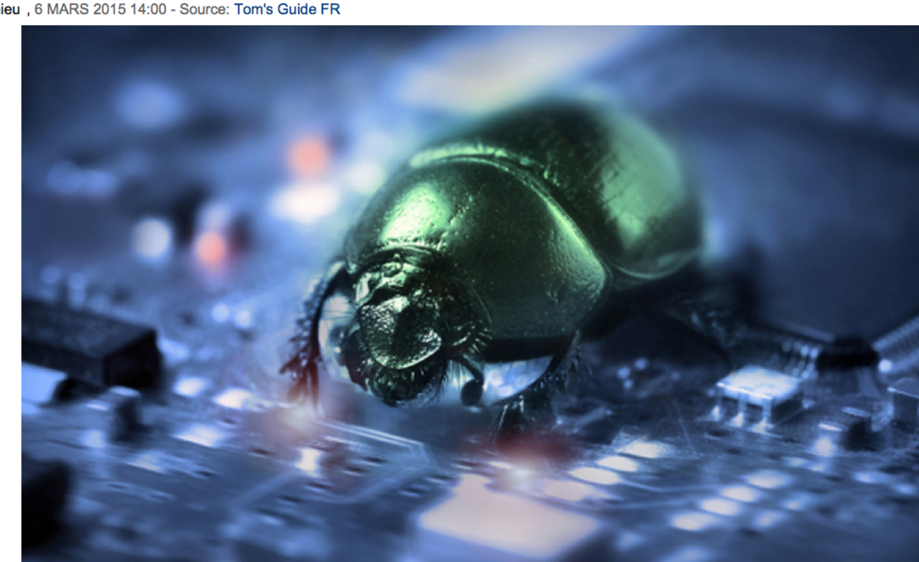
6 March 2015 Last updated at 13:05 GMT

Millions at risk from 'Freak' encryption bug

tom's GUIDE

[Màj] Faille de sécurité : le « freak », ça pique

Par Bruno Mathieu , 6 MARS 2015 14:00 - Source: Tom's Guide FR



Export-Grade DHE in TLS

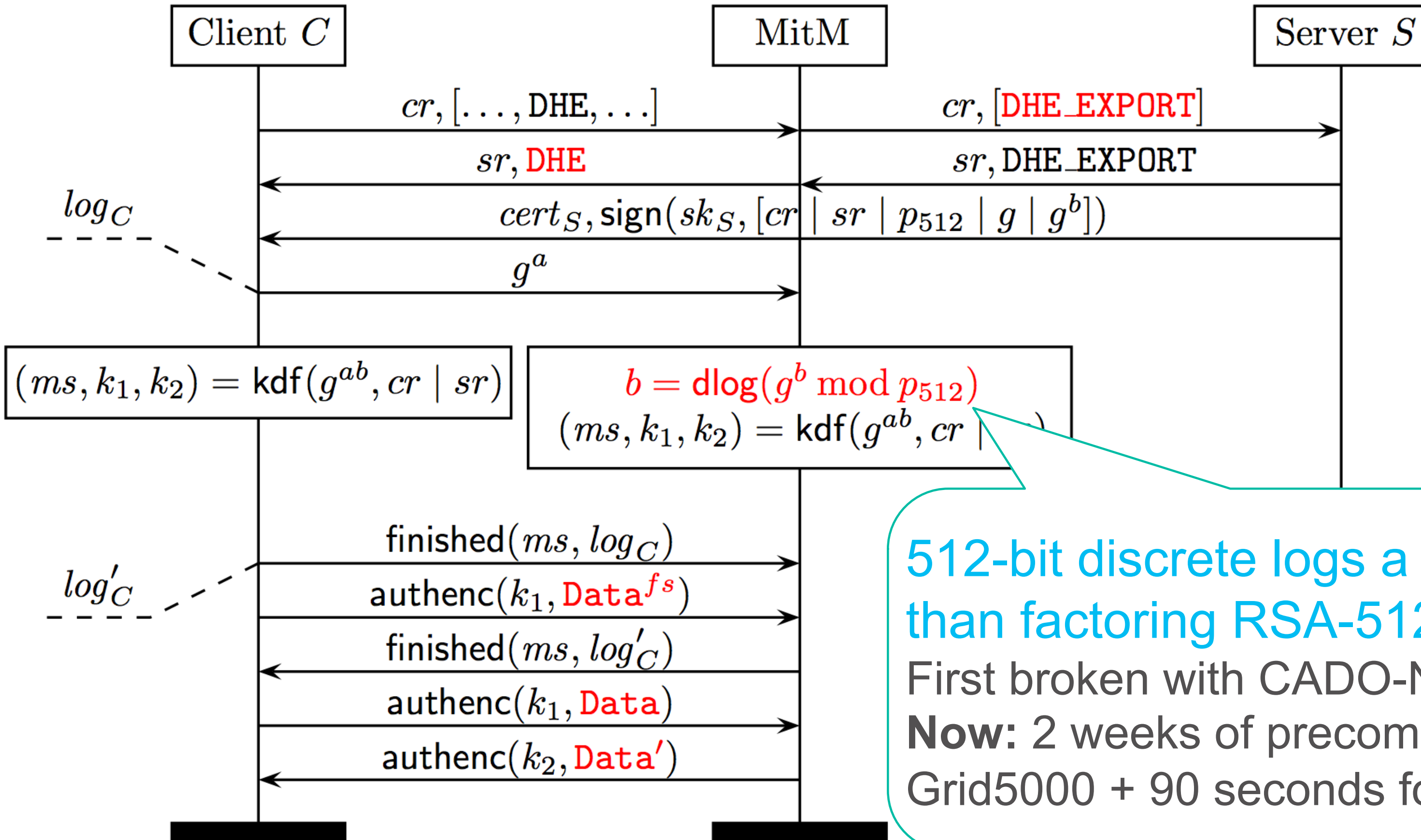
Yet another export-grade cipher in TLS

- Diffie-Hellman groups limited to 512 bits
- Protocol flaw: Messages look the same as regular DHE!

Logjam: Downgrade to DHE_EXPORT

A man-in-the-middle attacker can:

- impersonate servers that support DHE_EXPORT,
- at **ALL** clients that accept 512-bit DH groups



512-bit discrete logs a bit harder than factoring RSA-512

First broken with CADO-NFS in 2014

Now: 2 weeks of precomputation on Grid5000 + 90 seconds for each key

Logjam: Exploit and Impact

Of course, many servers still offer DHE_EXPORT

- 8.4% of Alexa Top 1M websites in March 2015
- Vulnerable sites included fbi.gov, tcl.tk, …
- See demos at weakdh.org

New worry: 768-bit,1024-bit discrete logs are feasible

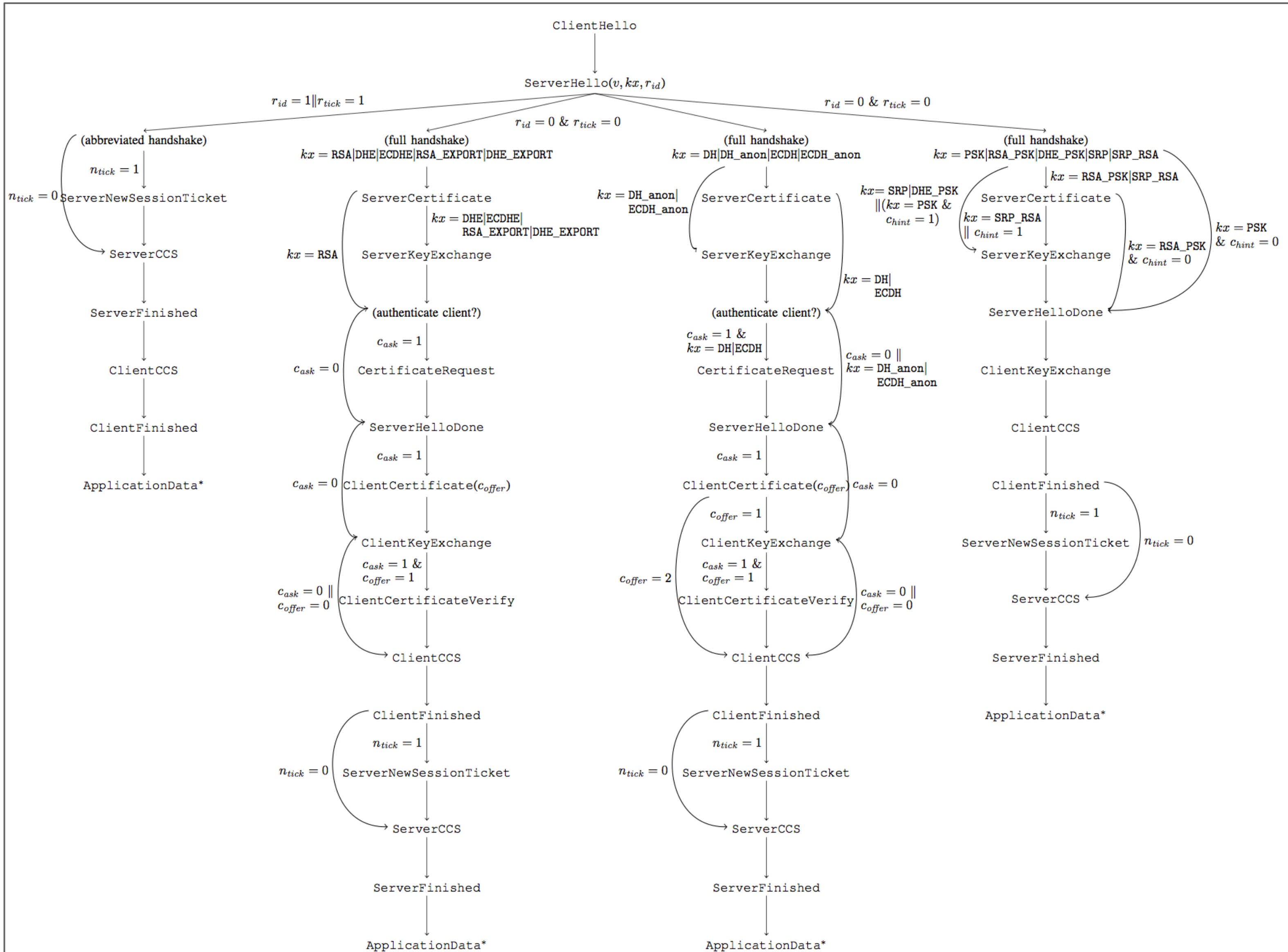
- 768-bits would require months of precomputation
- 1024-bits would require supercomputers
- IPsec, SSH, TLS all use 768-bit and 1024-bit primes!

Security updates to major browsers and websites

- Disabling 512-bit, then 768-bit, then 1024 bit
- Move to 20148-bit freshly-generated safe primes

Long-term Solutions?

A Verified State Machine for OpenSSL



A Verified State Machine for OpenSSL

OpenSSL has two state machines (client/server)

- A bit of a mess: many protocol versions, extensions, optional, and experimental features

We rewrote this code and verified it with Frama-C

- 750 lines of code, 460 lines of specification
- 1 month of a PhD student's time
- Reused logical specification from miTLS
- Eliminates all state machine bugs in OpenSSL
- No impact on performance!

A new protocol: TLS 1.3

Stronger key exchanges, fewer options

- ECDHE and DHE by default, no RSA key transport
- Strong DH groups (> 2047 bits) and EC curves (> 255 bits)
- Only AEAD ciphers (AES-GCM), no CBC, no RC4

Signatures, session keys bound to handshake params

- Session hash for key derivation (proposed by us)
- Server signature covers ciphersuite (preventing Logjam)

Faster: lower latency with 1 round-trip

- 0-round trip mode also available
- Security analysis ongoing

Conclusions

Cryptographic protocol testing needs work

- We used a specification-driven fuzzing tool to find critical state machine bugs in a number of libraries
- This should be done systematically by developers

Open source code is not immune from attack

- Security bugs can hide in plain sight for years

Verification of production code is feasible

- We focused on the core state machine, one small step towards verifying OpenSSL

Beware of deliberately weakened cryptography

- Backdoors come back to bite you even decades later

Questions?

mitls.org

smacktls.com

weakdh.org