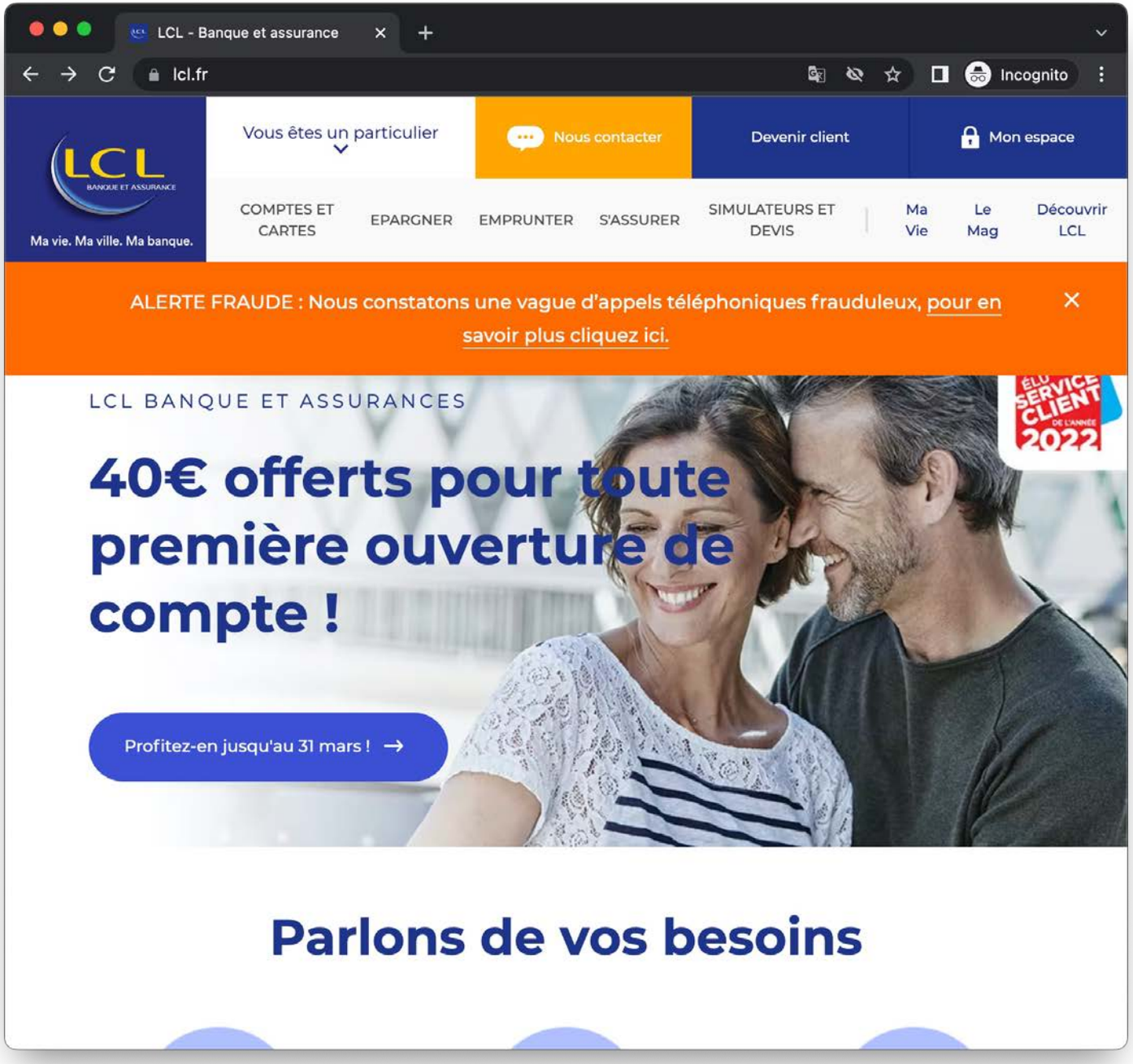
Verified Implementations for Real-World Cryptographic Protocols

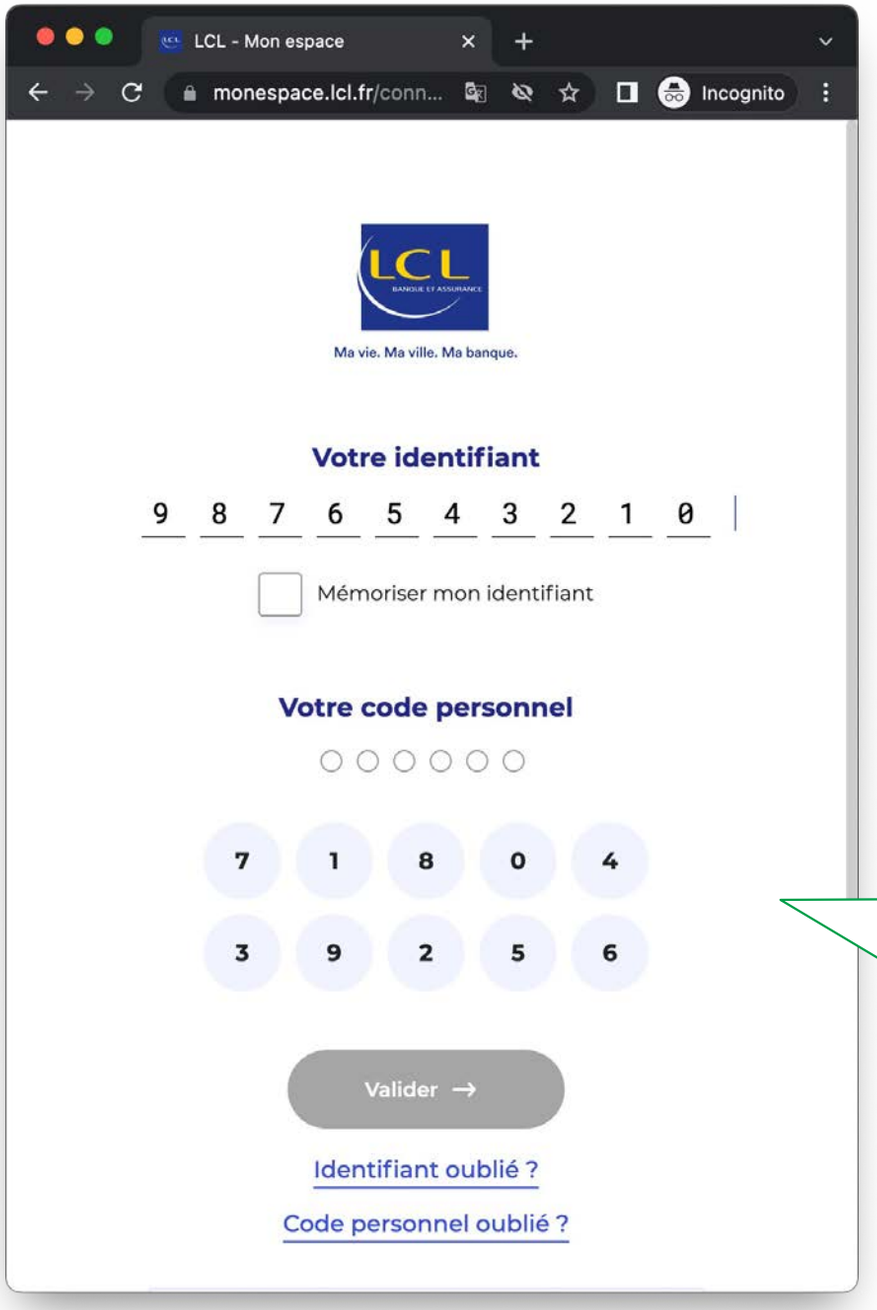
Karthikeyan Bhargavan

Inria

+ many co-authors at Inria, MSR, CMU, Stuttgart, …

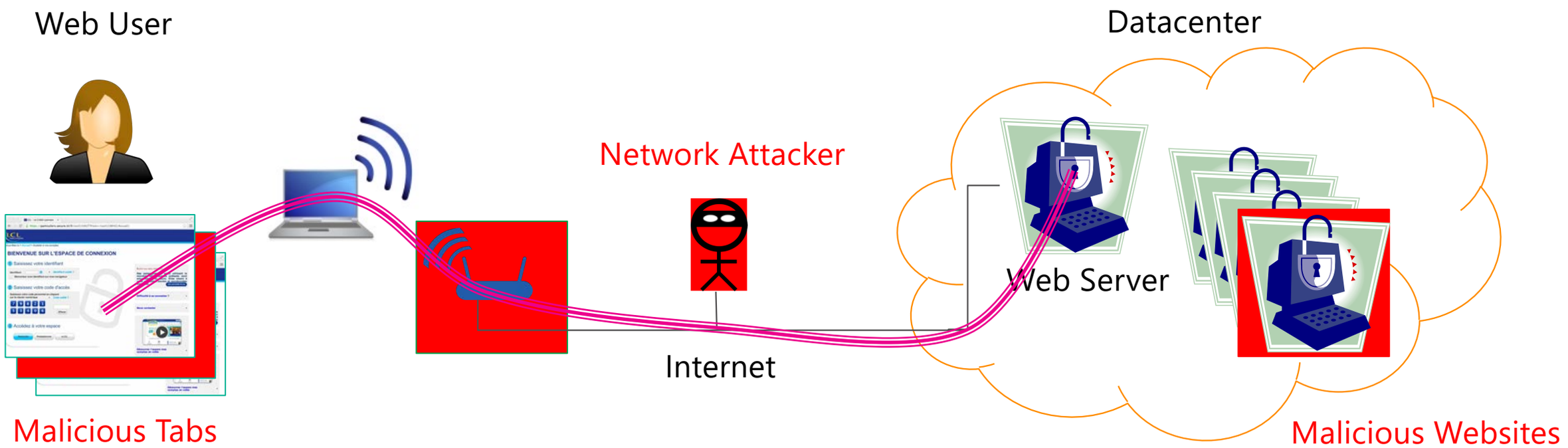
Collège de France, March 31 2022


LCL - Banque et assurance
lcl.fr
Incognito
LCL
BANQUE ET ASSURANCE
Ma vie. Ma ville. Ma banque.
Vous êtes un particulier
Nous contacter
Devenir client
Mon espace
COMPTES ET CARTES
EPARGNER
EMPRUNTER
S'ASSURER
SIMULATEURS ET DEVIS
Ma Vie
Le Mag
Découvrir LCL
ALERTE FRAUDE : Nous constatons une vague d'appels téléphoniques frauduleux, pour en savoir plus cliquez ici.
LCL BANQUE ET ASSURANCES
ÉLU SERVICE CLIENT DE L'ANNÉE 2022
40€ offerts pour toute première ouverture de compte !
Profitez-en jusqu'au 31 mars !
Parlons de vos besoins

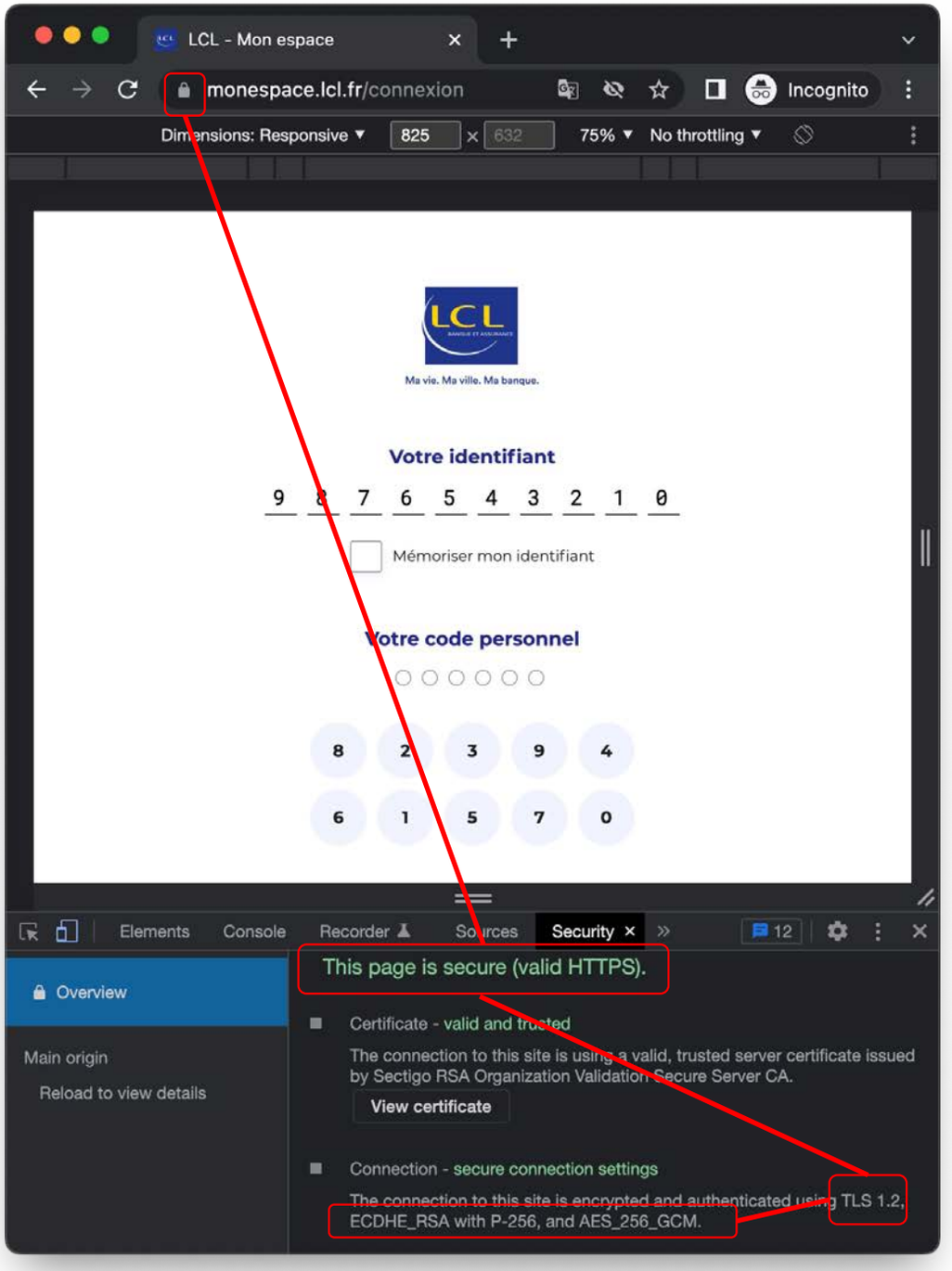


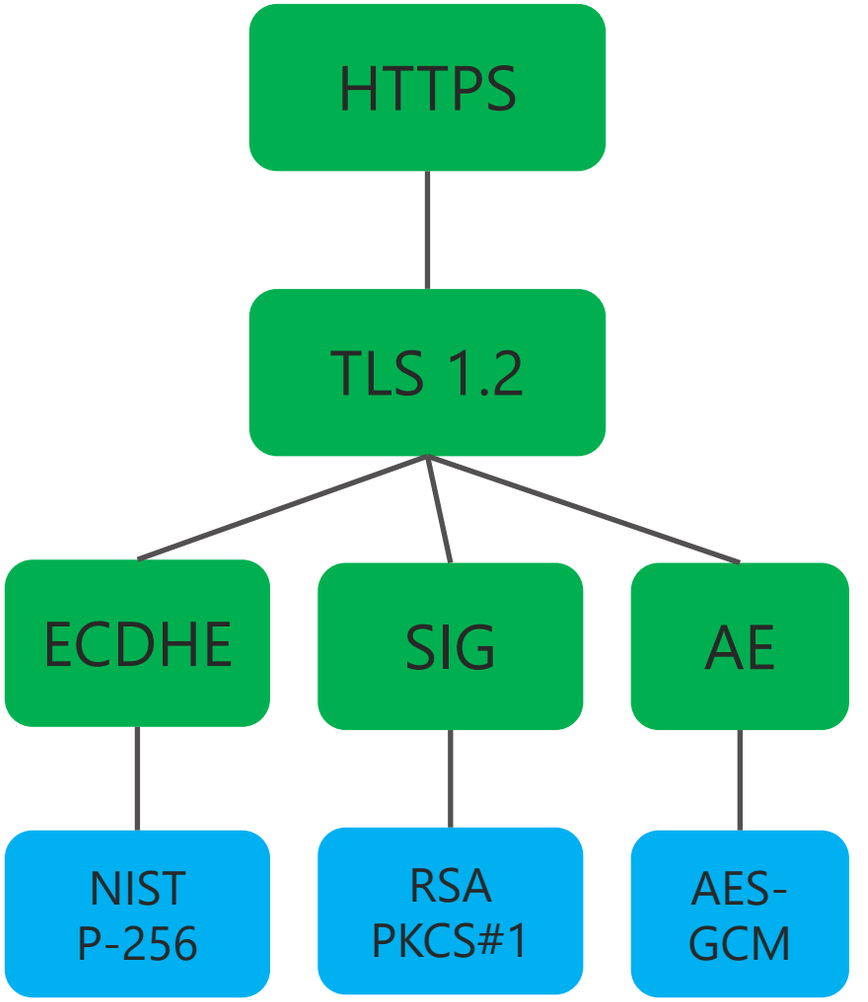
Is it safe to enter my pincode?

Secure Channels over Insecure Networks

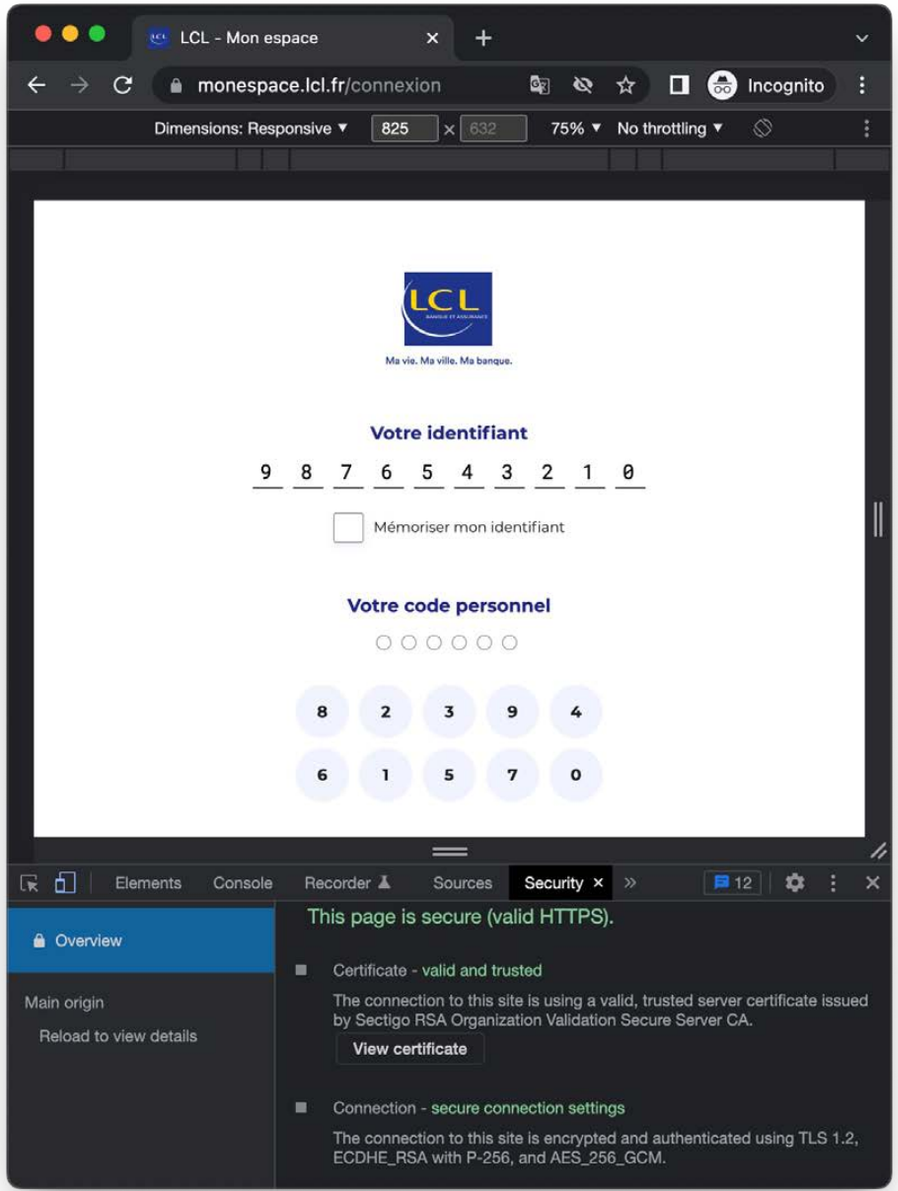


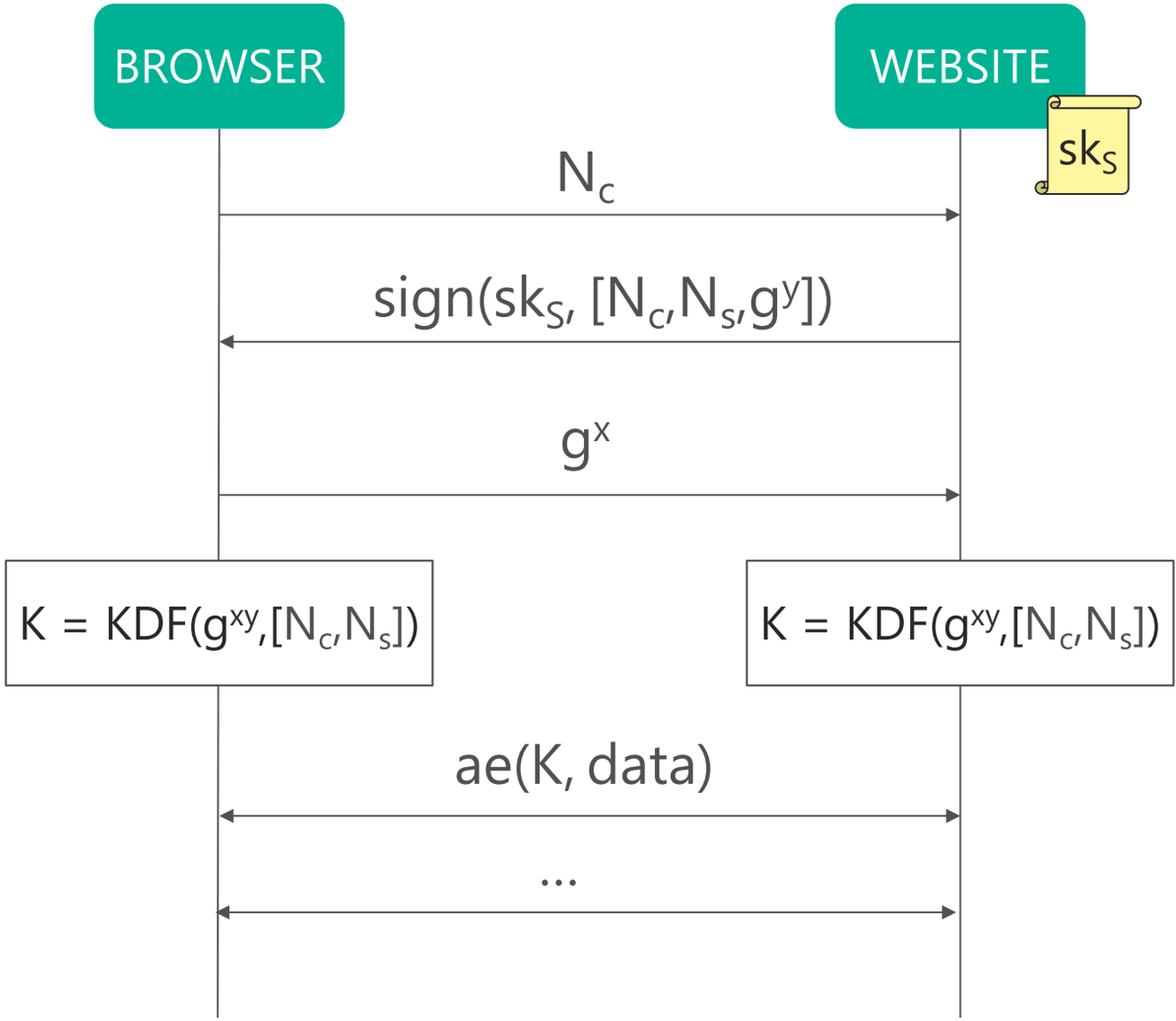
- We need a secure (authentic, confidential) channel
- Security against powerful attackers who may
  - Read all data sent on the network
  - Tamper with the contents of messages
  - Impersonate a user or a server


LCL - Mon espace
monespace.lcl.fr/connexion
Incognito
Dimensions: Responsive
825
632
75%
No throttling
Votre identifiant
9 8 7 6 5 4 3 2 1 0
Mémoriser mon identifiant
Votre code personnel
8 2 3 9 4
6 1 5 7 0
Elements
Console
Recorder
Sources
Security
12
Overview
Main origin
Reload to view details
This page is secure (valid HTTPS).
Certificate - valid and trusted
The connection to this site is using a valid, trusted server certificate issued by Sectigo RSA Organization Validation Secure Server CA.
View certificate
Connection - secure connection settings
The connection to this site is encrypted and authenticated using TLS 1.2, ECDHE_RSA with P-256, and AES_256_GCM.


HTTPS
TLS 1.2
ECDHE
SIG
AE
NIST P-256
RSA PKCS#1
AES-GCM

Secure Channels in TLS [1994-2008]



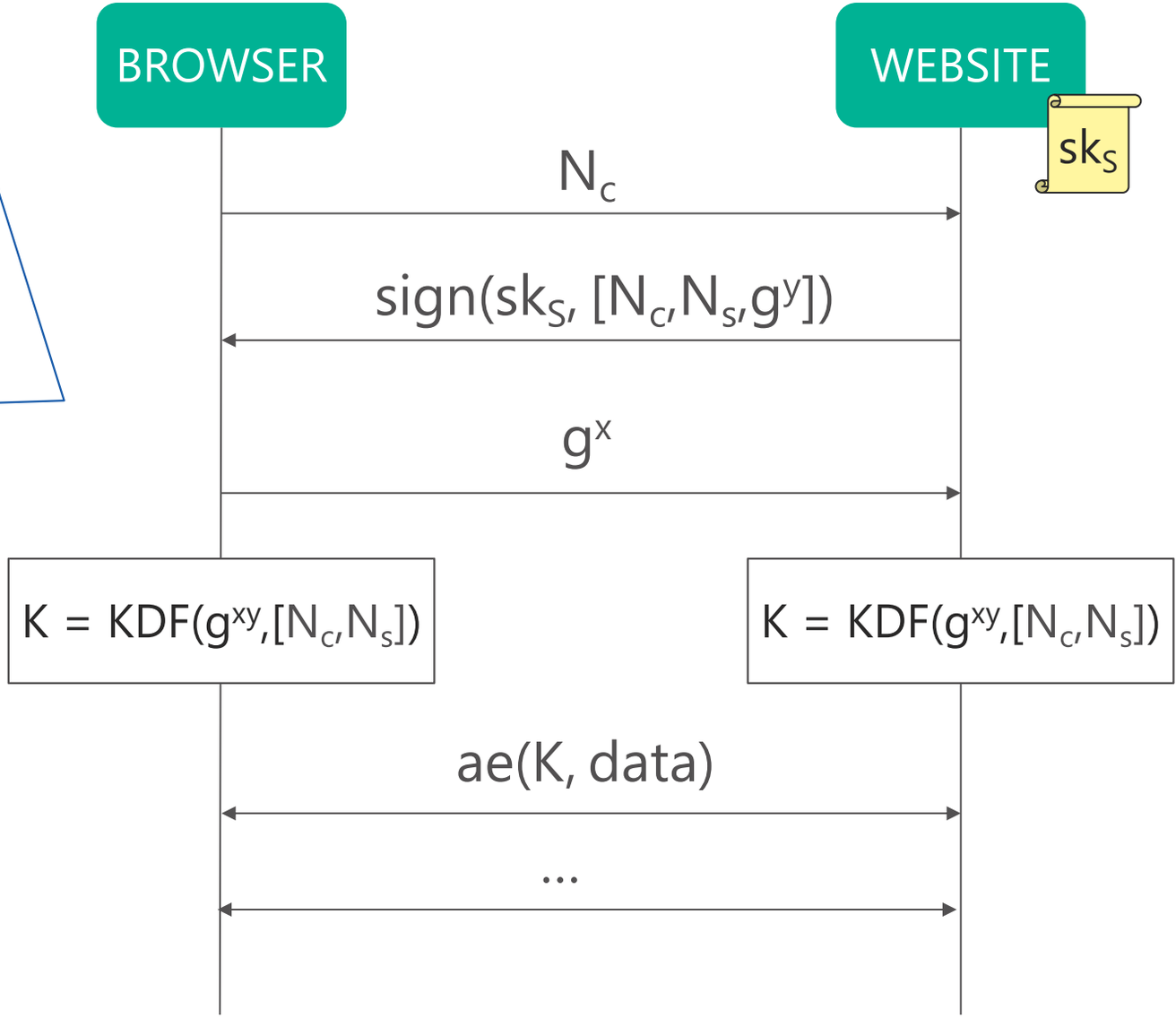


Secure Channels in TLS [1994-2008]

Classic Two-Stage Protocol

1. **Authenticated Key Exchange**
   Both parties compute a shared secret key K
2. **Authenticated Encryption**
   Both parties exchange streams of data encrypted under K

- Many variants (IPsec, SSH, TLS)
- Many security models & proofs
- So, is this a solved problem?



Many Attacks on TLS Deployments [2011-16]

• 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]
• 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 collisions	[May'16]

What goes wrong in TLS Deployments?

Crypto Weaknesses

- RC4, 3DES, MD5, PKCS#1 v1.5

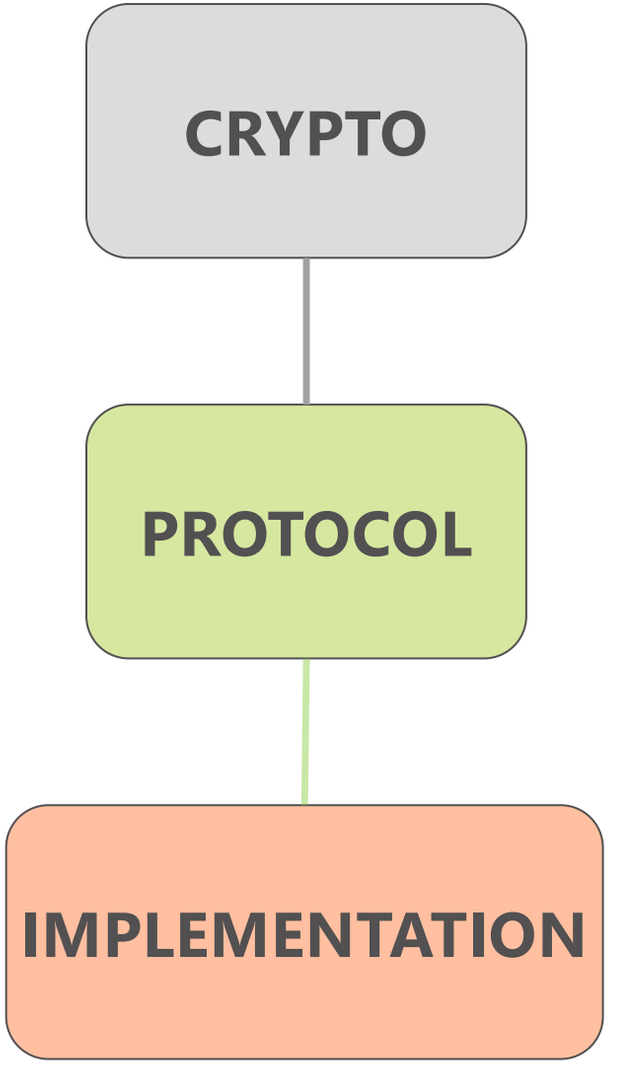
Protocol Design Flaws

- Downgrade attacks, Transcript collisions

Implementation Bugs

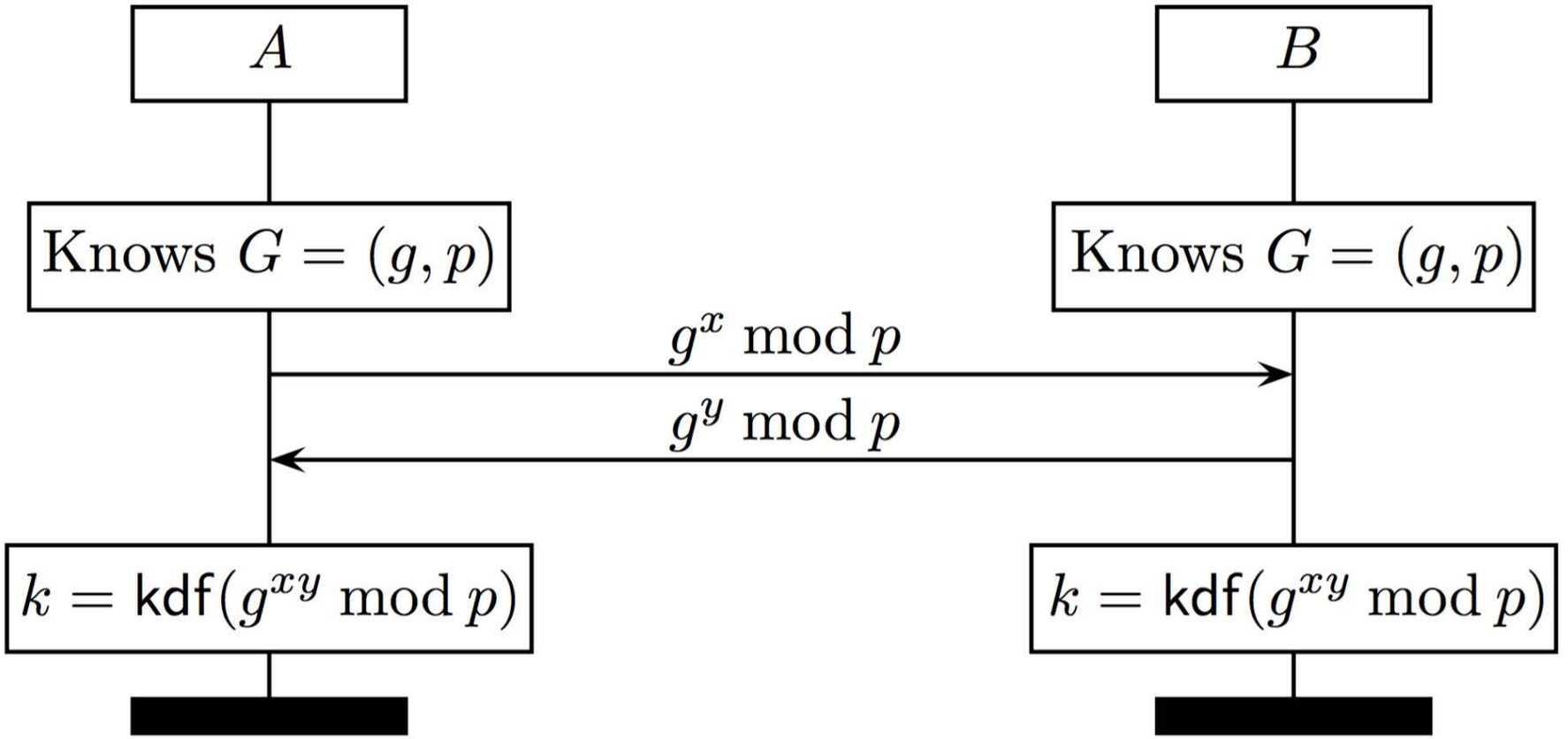
- State machine bugs, Heartbleed

Often, a mix of all of the above!

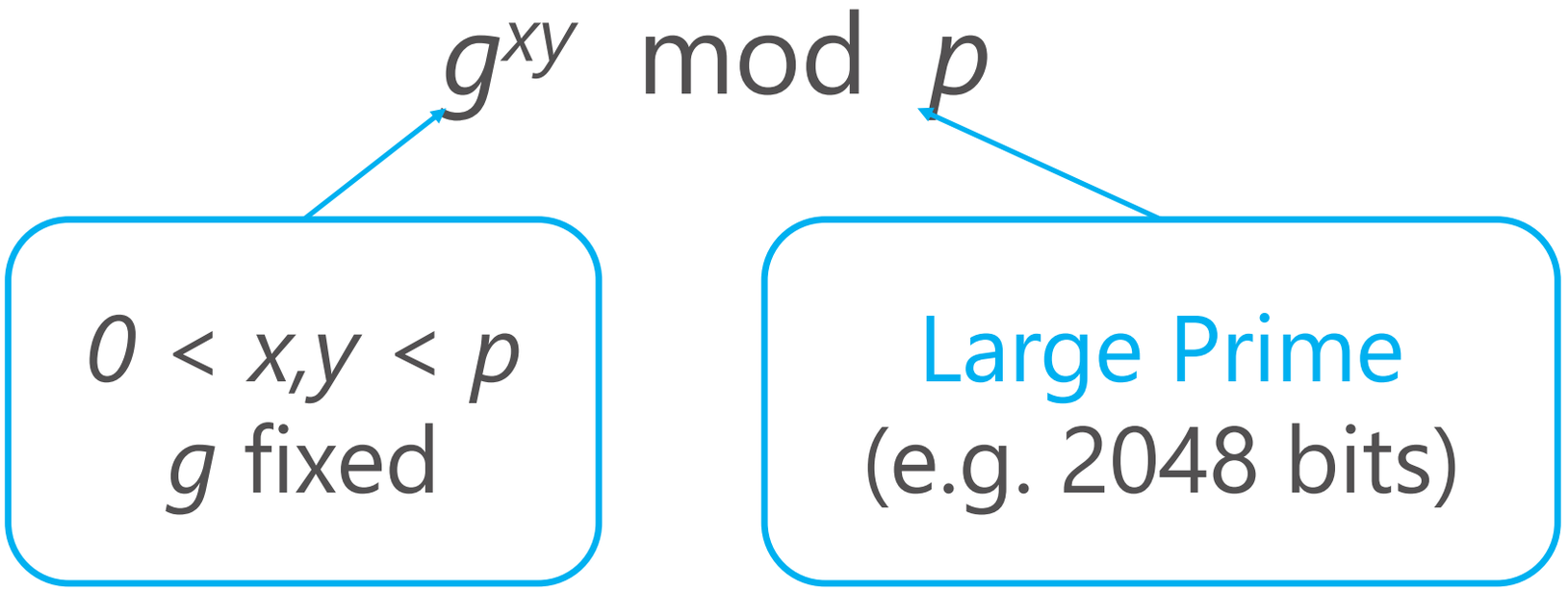


Crypto Weaknesses

Diffie-Hellman Key Exchange



Diffie-Hellman Assumption



Security Assumption: An attacker who does not know x or y cannot compute g^{xy} mod p

Weak Diffie-Hellman Groups

If the prime p is too small,
an attacker can compute the discrete log:

$$y = \log(g^y \bmod p)$$

and hence compute the session key: $g^{xy} \bmod p$

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
- [Boudot et al. 2019] 795-bit prime

Real-World Diffie-Hellman Groups

Internet-wide scan of HTTPS servers (2015)

- 14.3M hosts, 24% support DHE
- 70,000 distinct groups *(p,g)*

Many small-sized prime groups used for TLS

- 84% (2.9M) servers use 1024-bit primes
- 2.6% (90K) servers use 768-bit primes
- 0.0008% (2.6K) servers use 512-bit primes

Many servers support both strong and weak groups.

Protecting Protocols from Weak Crypto

Many deployed crypto algorithms are now considered weak

- RC4, MD5, 3DES, RSA-PKCS#1v1.5

The need for **backwards compatibility**

- Many systems cannot be updated frequently
- Need to continue support for legacy clients/servers

The benefits of **cryptographic agility**

- Gracefully transition from one algorithm to another
- Can already start supporting Post-Quantum algorithms

Protecting Protocols from Weak Crypto

Prove the security of protocols under weak assumptions

- Do you really need a collision-resistant hash function?
- Do you really need an IND-CCA secure encryption algorithm?

Analyze protocols that support both strong and weak crypto

- Prove security for connections that use strong crypto
- Show that strong crypto cannot be bypassed using weak crypto

Analyzing agile protocols by hand is too hard

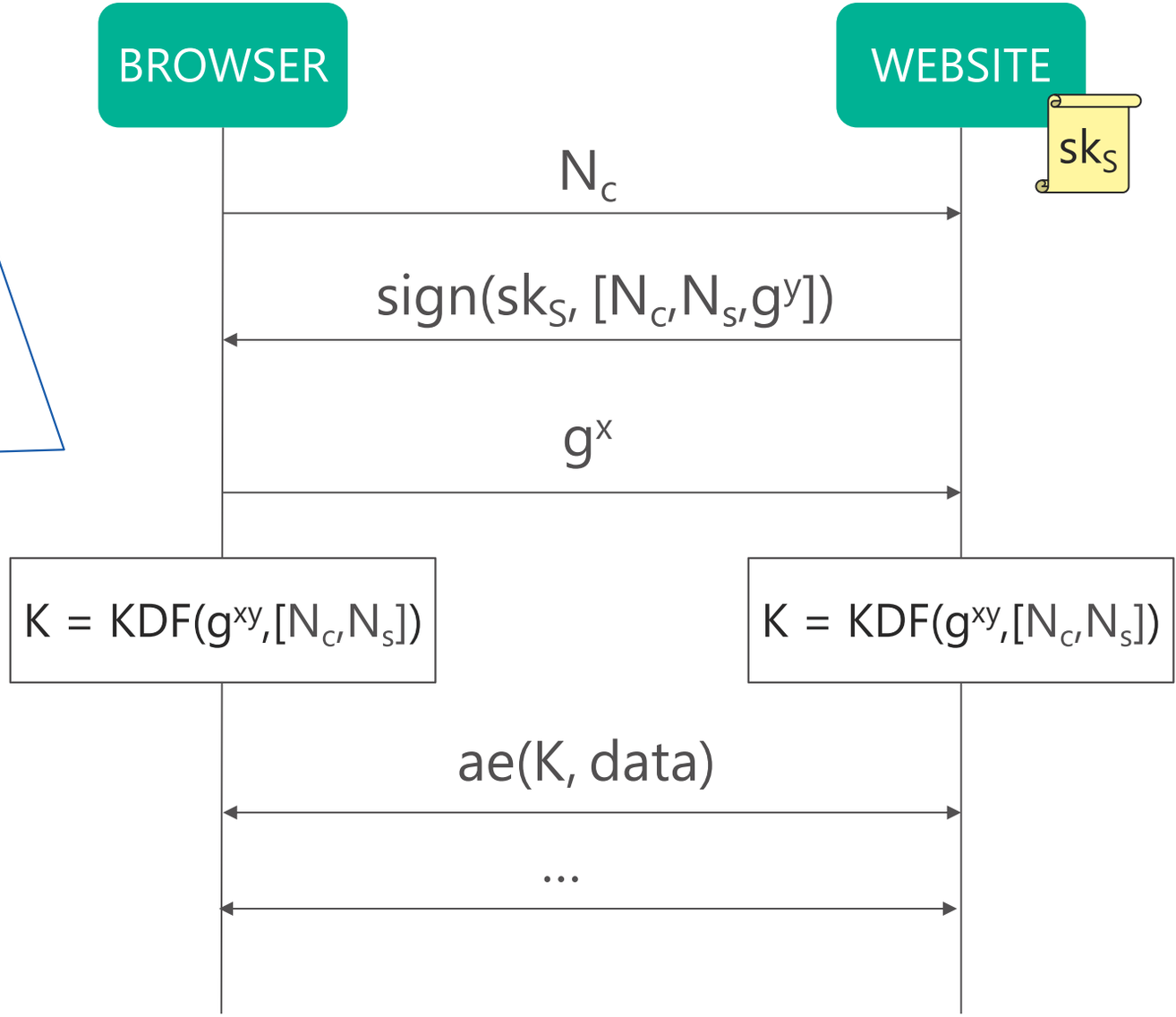
- Large models, subtle assumptions and goals
- Need mechanized protocol verification tools

Protocol Design Flaws

Composing Cryptographic Constructions

Each crypto protocol composes a set of crypto constructions to achieve some target security goals

- TLS = DH + Sign + KDF + AE
- Each crypto algorithm may be individually strong, but they may not collectively achieve the desired security goal



Composing Sub-Protocols

Sequential or Vertical Composition

- Values generated by one protocol are used in the next
- e.g. Authenticated Key Exchange + Authenticated Encryption

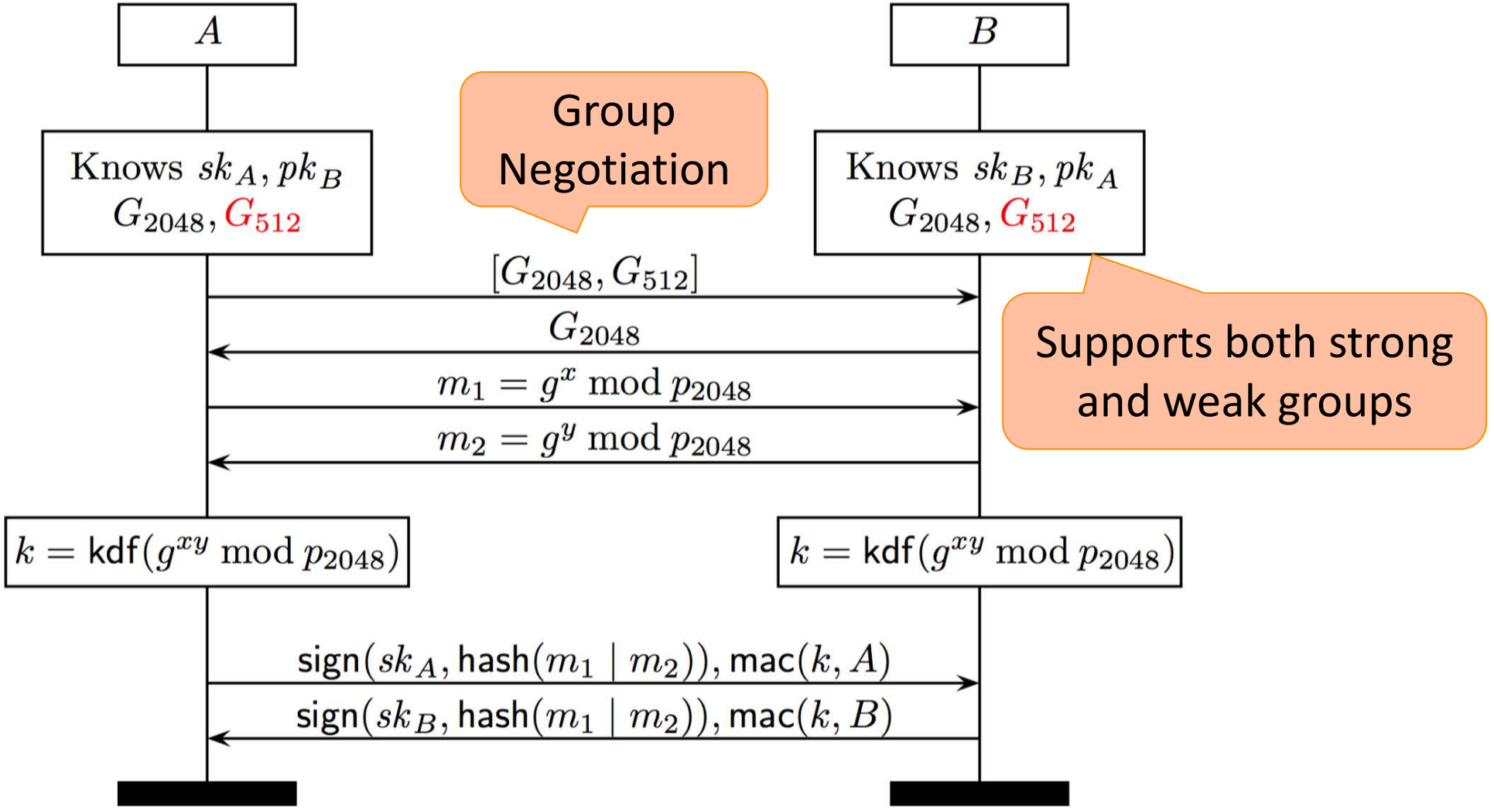
Protocols with Algorithmic Agility

- Support for multiple algorithms within a single protocol
- e.g. allow weak and strong Diffie-Hellman groups

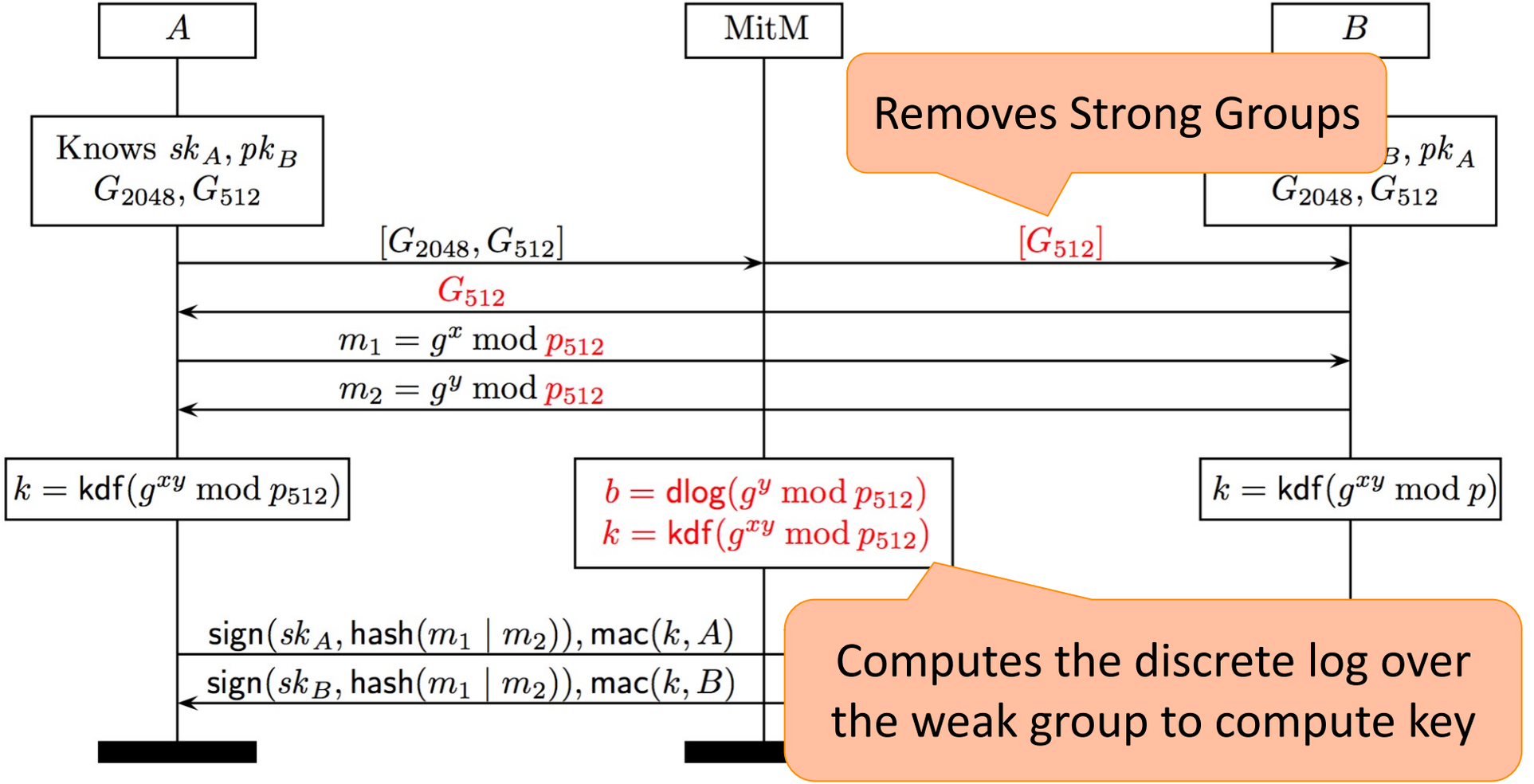
Parallel or Multi-mode Composition

- Many protocol flows to choose from
- Different sessions may choose a different modes

Agility: Diffie-Hellman Group Negotiation



Group Downgrade Attack



What went wrong?

Logjam Attack [2015]

- Cryptographic weakness: Weak Diffie-Hellman Groups
- Logical protocol flaw: Downgrade from Strong to Weak Group

Many other examples of downgrade+crypto attacks

- FREAK, SLOTH, DROWN, …

These attacks only appear when analyzing complex composite protocol deployments

Implementation Bugs

Bugs in Protocol Implementations

Bugs when implementing cryptographic algorithms

- Functional correctness bugs, Side-channel leaks, …
- e.g. Lucky13, Bleichenbacher, see OpenSSL CVEs

Bugs when parsing protocol messages and components

- Memory safety bugs (Heartbleed), Error propagation (Gotofail)
- X.509 certificate parsing errors (many CVEs)

Bugs in protocol state machine implementation (next)

- Allowing incorrect protocol flows (FREAK, SKIP)

Many possible protocol modes of TLS

Protocol versions

- TLS 1.2, TLS 1.1, TLS 1.0, SSLv3, SSLv2

Key exchanges

- ECDHE, FFDHE, RSA, PSK, …

Authentication modes

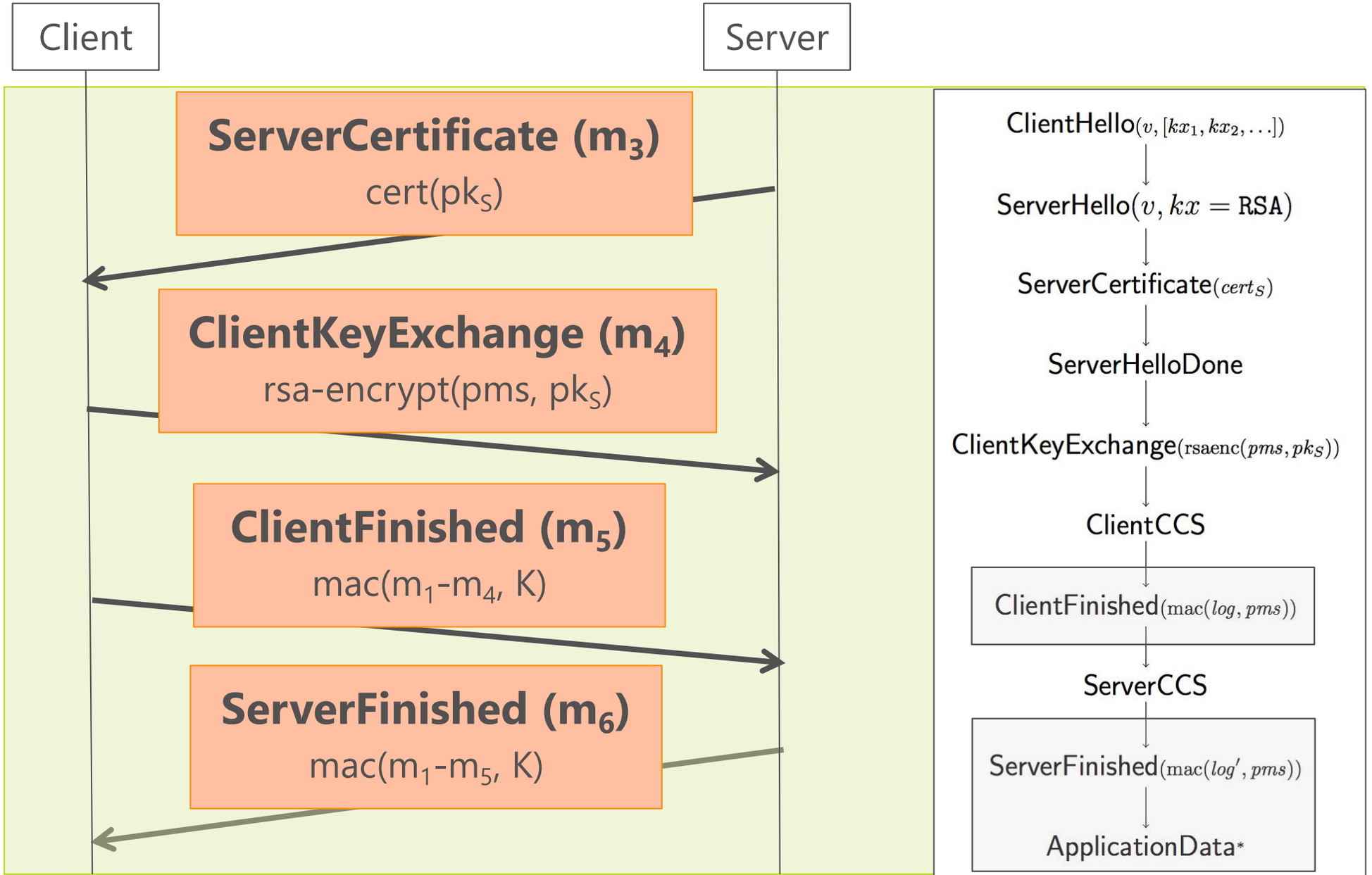
- ECDSA, RSA signatures, PSK,…

Authenticated Encryption Schemes

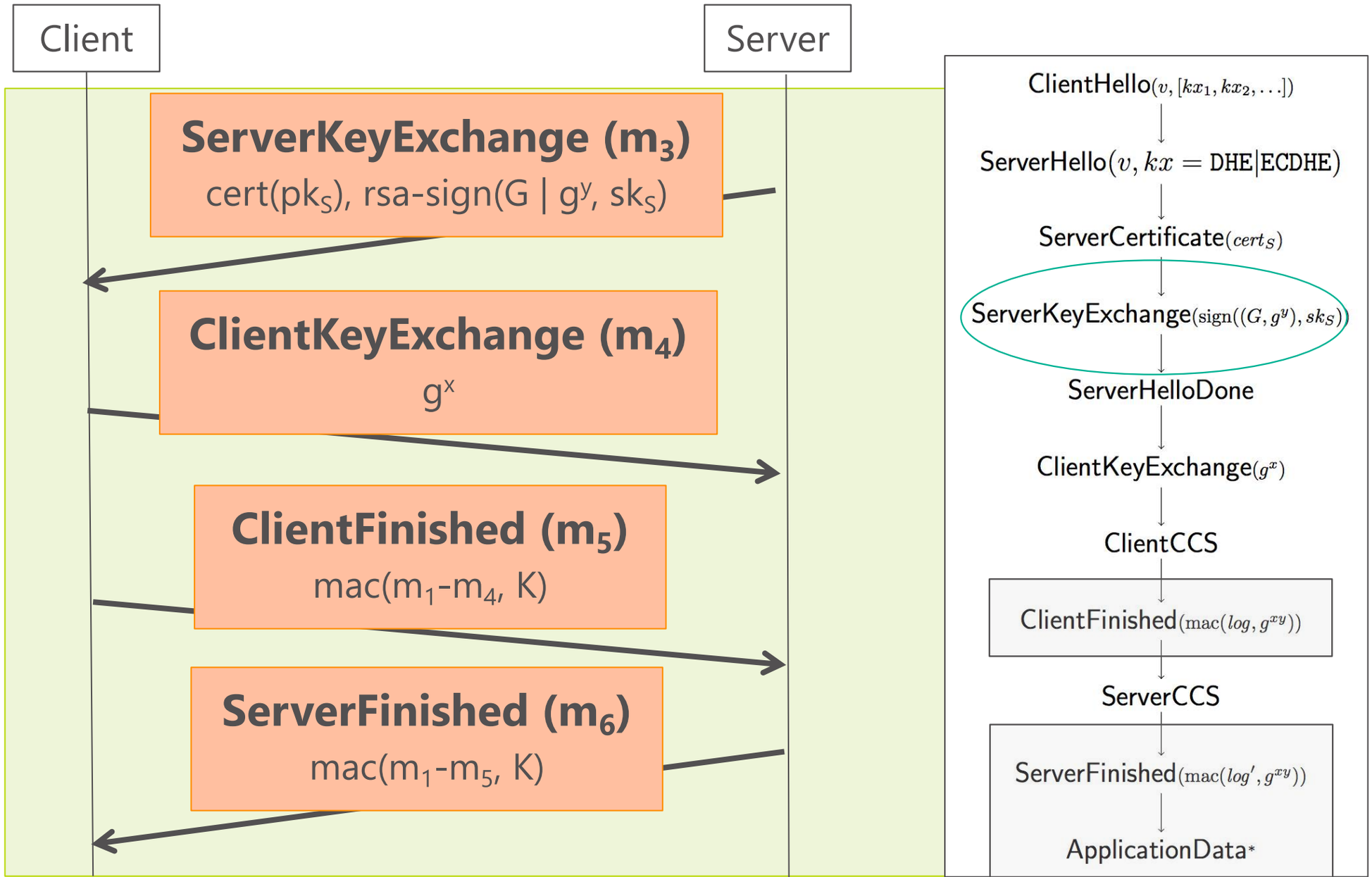
- AES-GCM, CBC MAC-Encode-Encrypt, RC4,…

100s of possible protocol combinations!

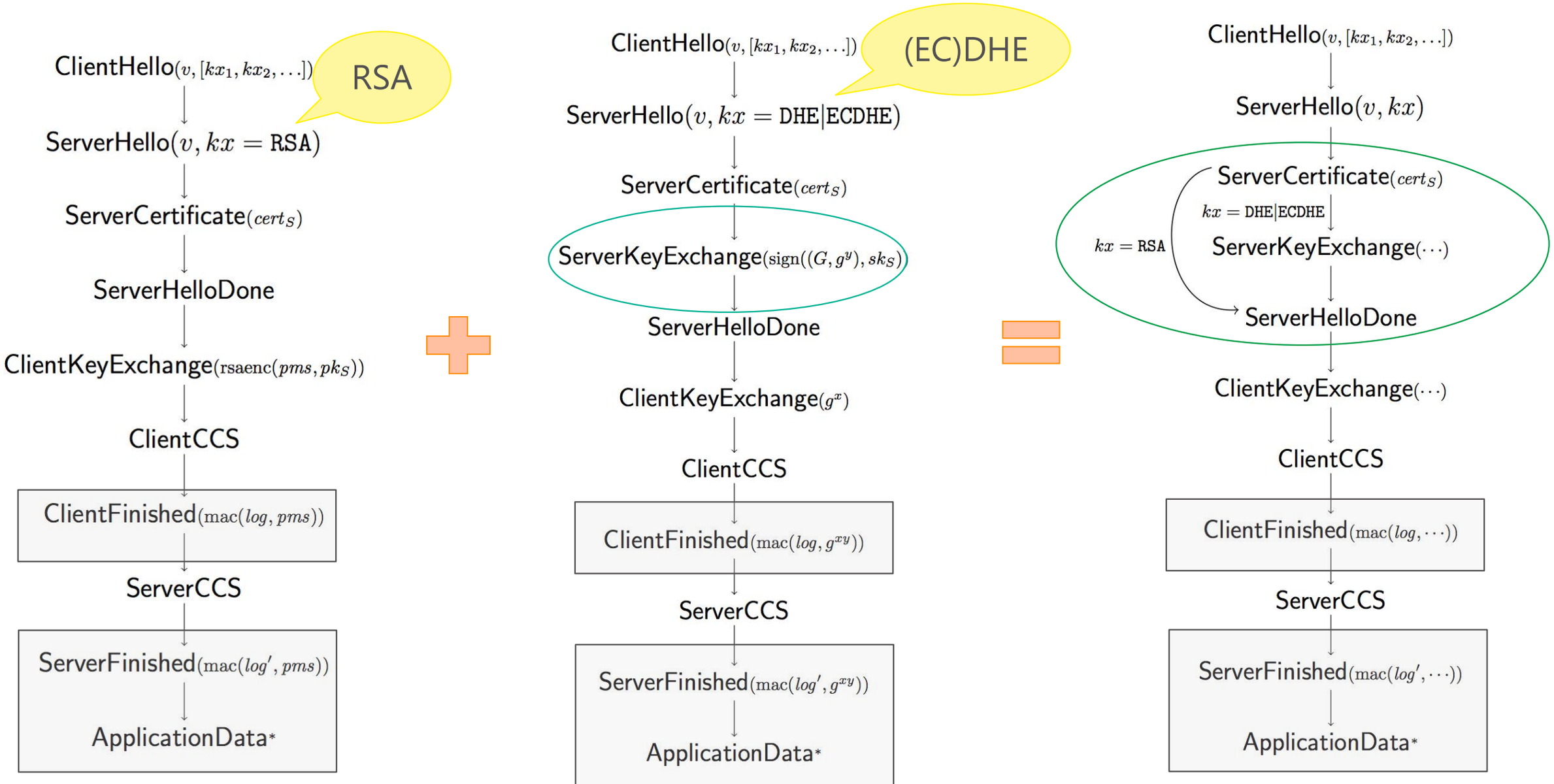
State Machine for TLS-RSA Key Exchange



State Machine for TLS-DHE Key Exchange



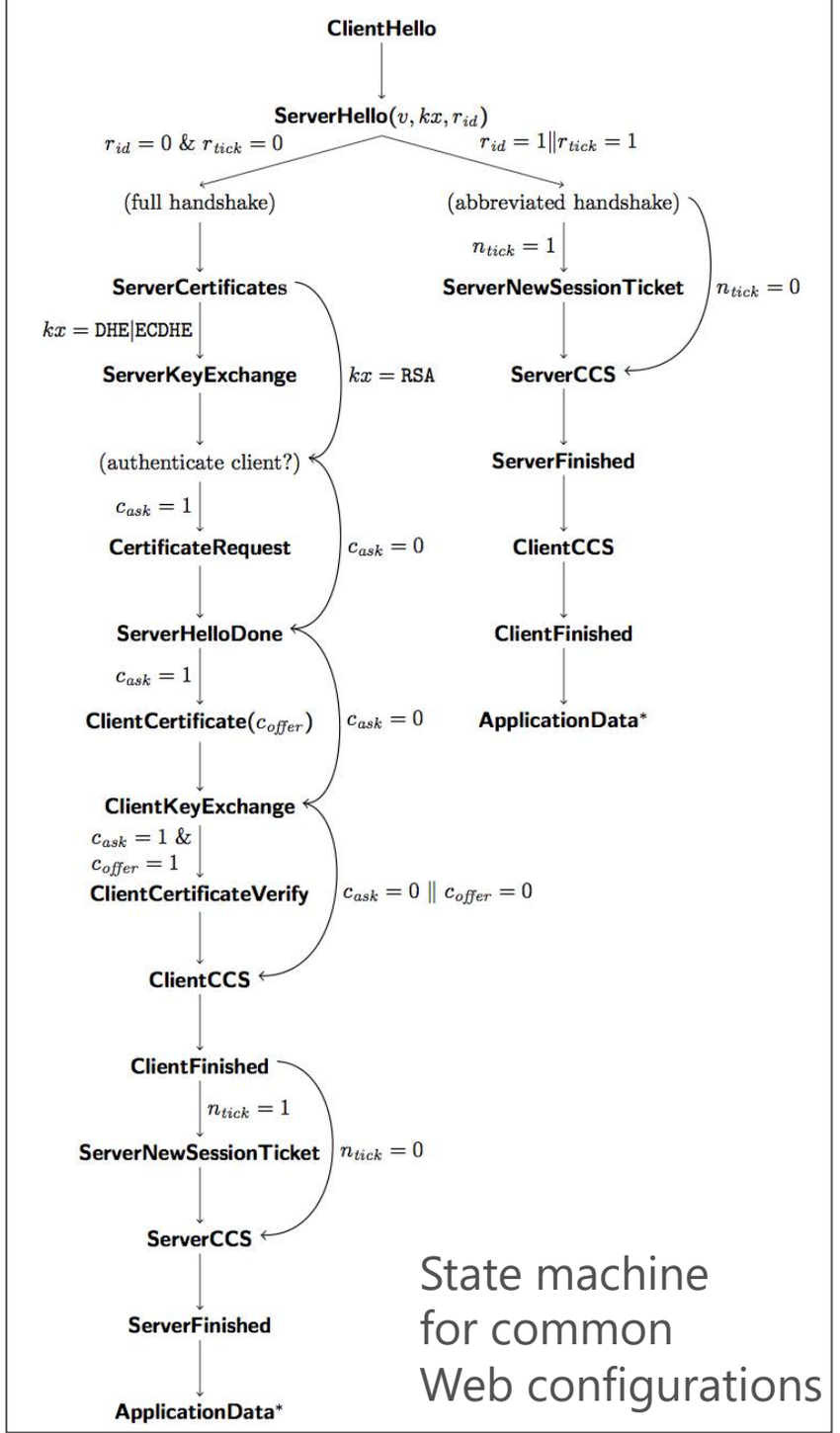
Composing Protocol State Machines



Commonly Deployed TLS State Machine

RSA + DHE + ECDHE
+ Session Resumption
+ Client Authentication

- Covers most features used on the Web

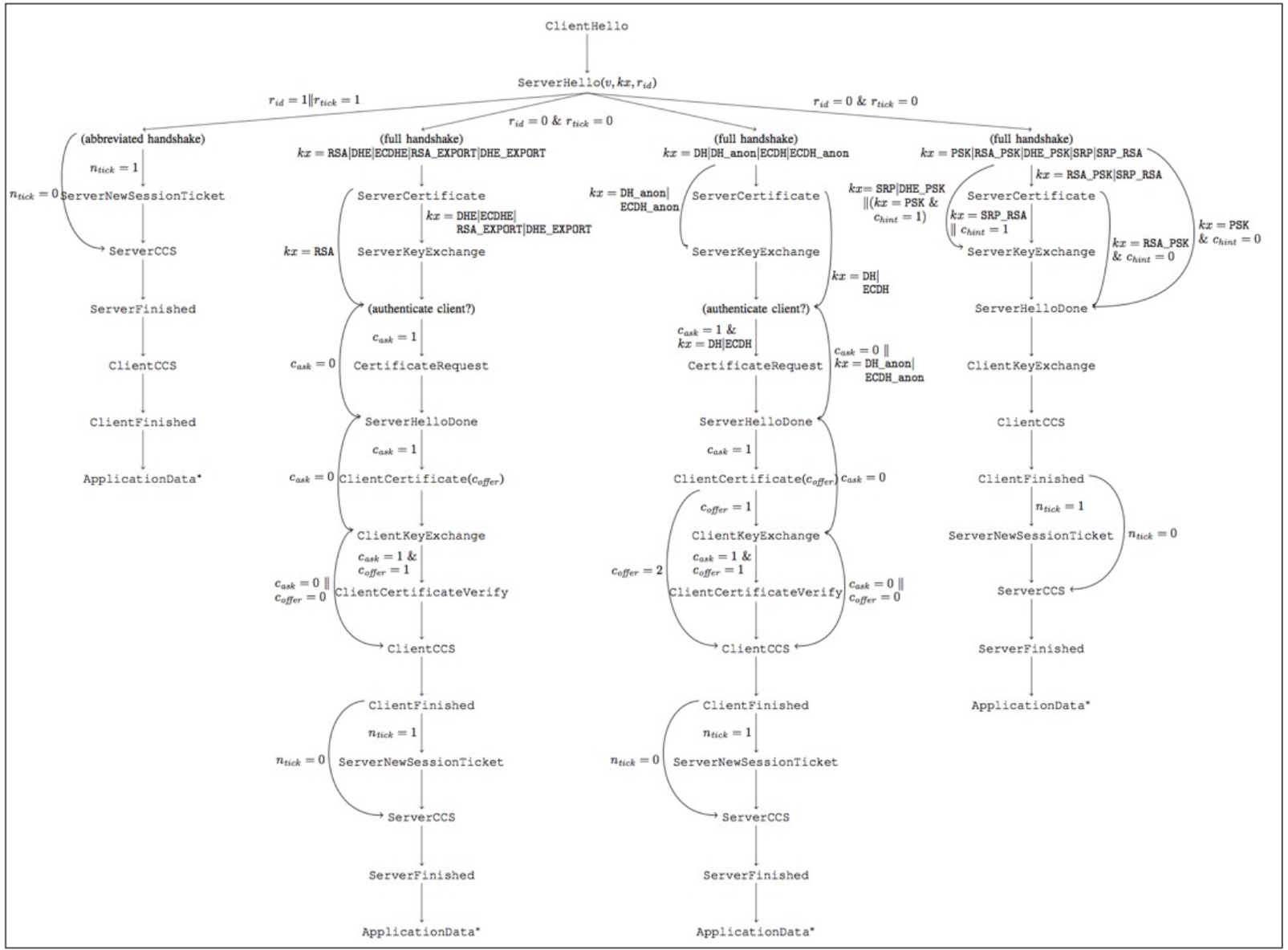


State machine for common Web configurations

Full SSL/TLS State Machine

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

These are all the ones implemented in OpenSSL



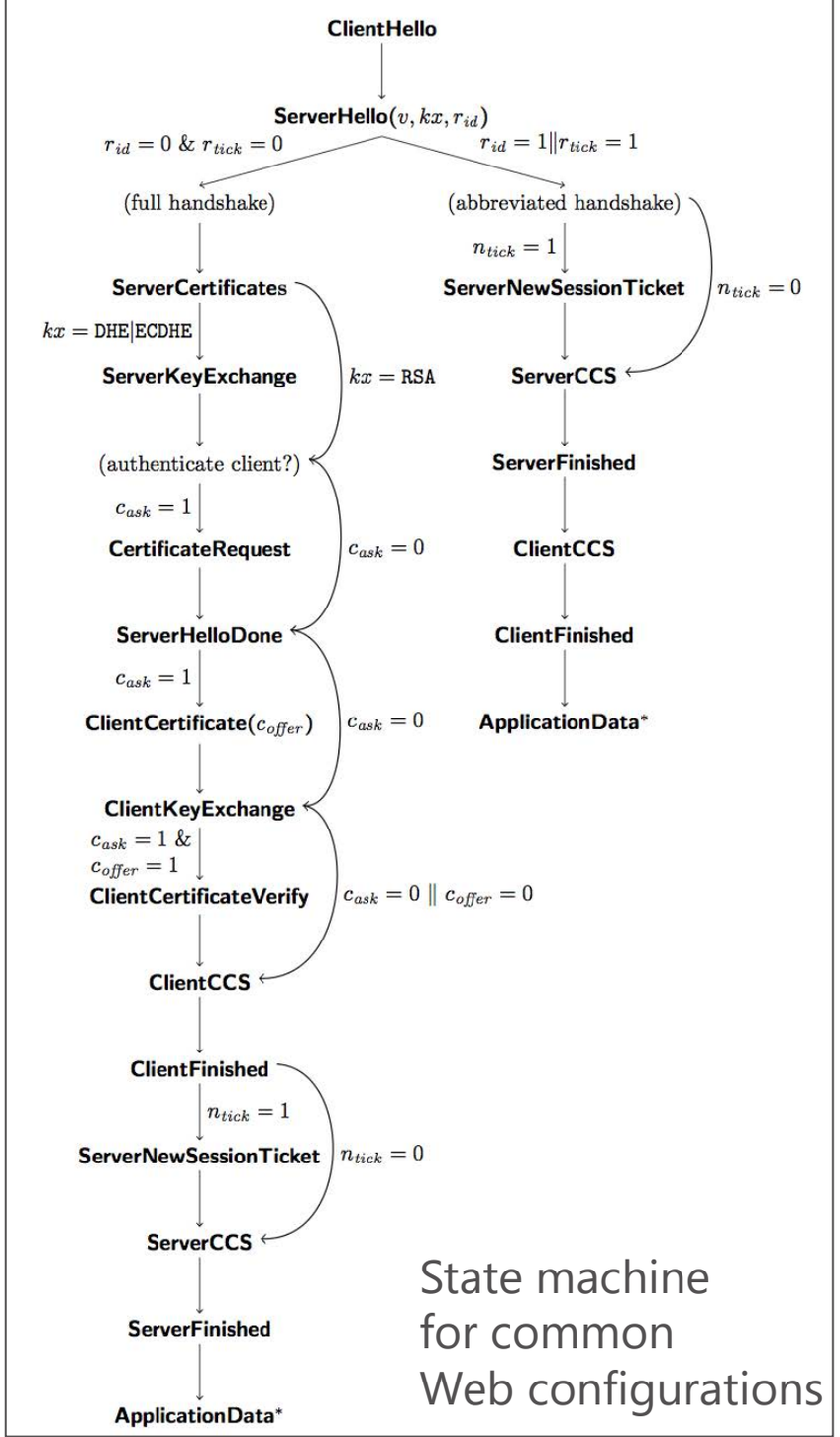
Testing TLS State Machines

Do popular TLS libraries conform to this state machine spec?

- Does OpenSSL? Firefox? Chrome? Safari? Java TLS? IoT devices?

We built a fuzzing framework

- FlexTLS, based on miTLS, a verified implementation of TLS
- Generates non-conforming traces from a *formal state machine spec*
- Tests open-source libraries

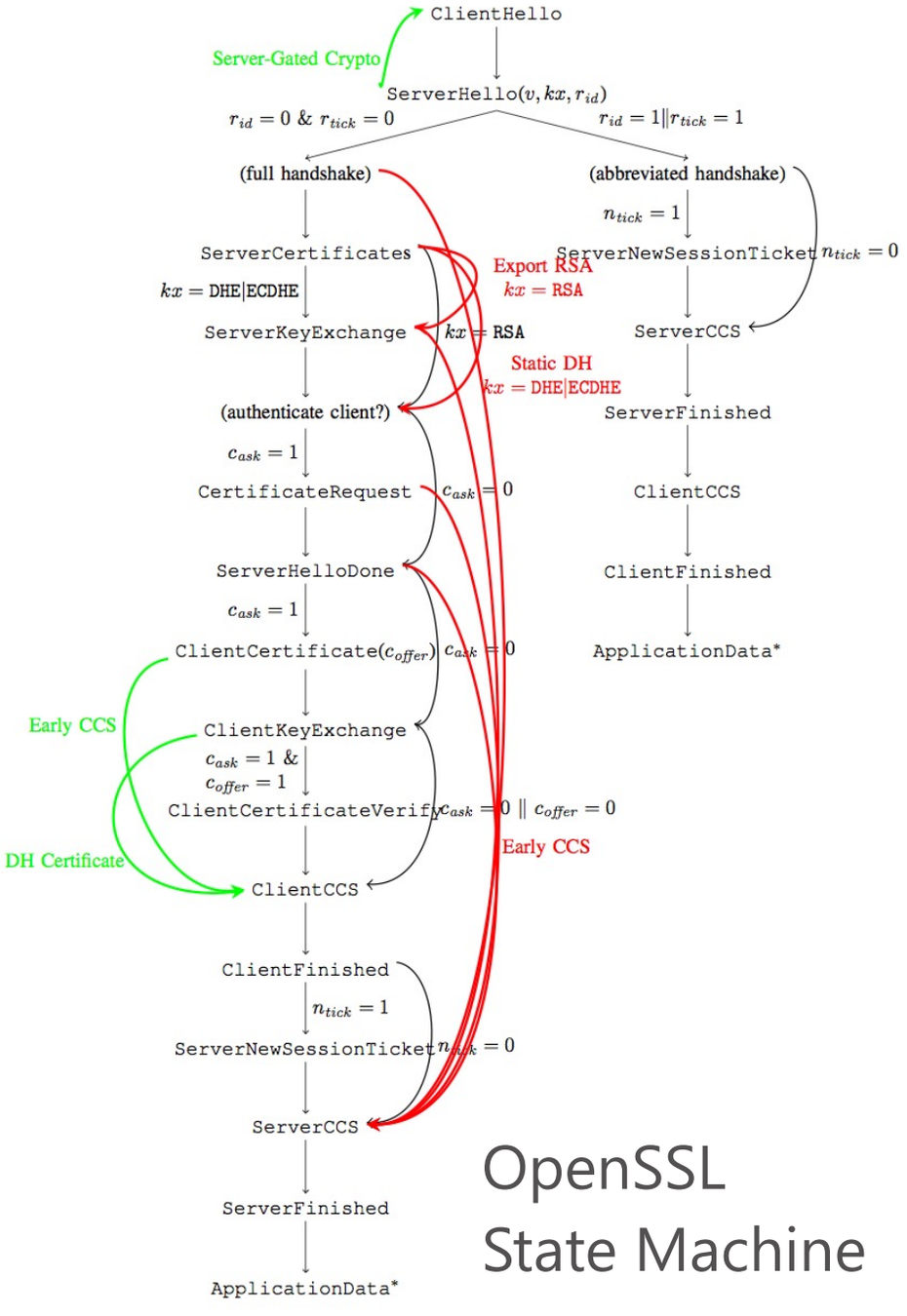


State machine for common Web configurations

Many, Many Bugs

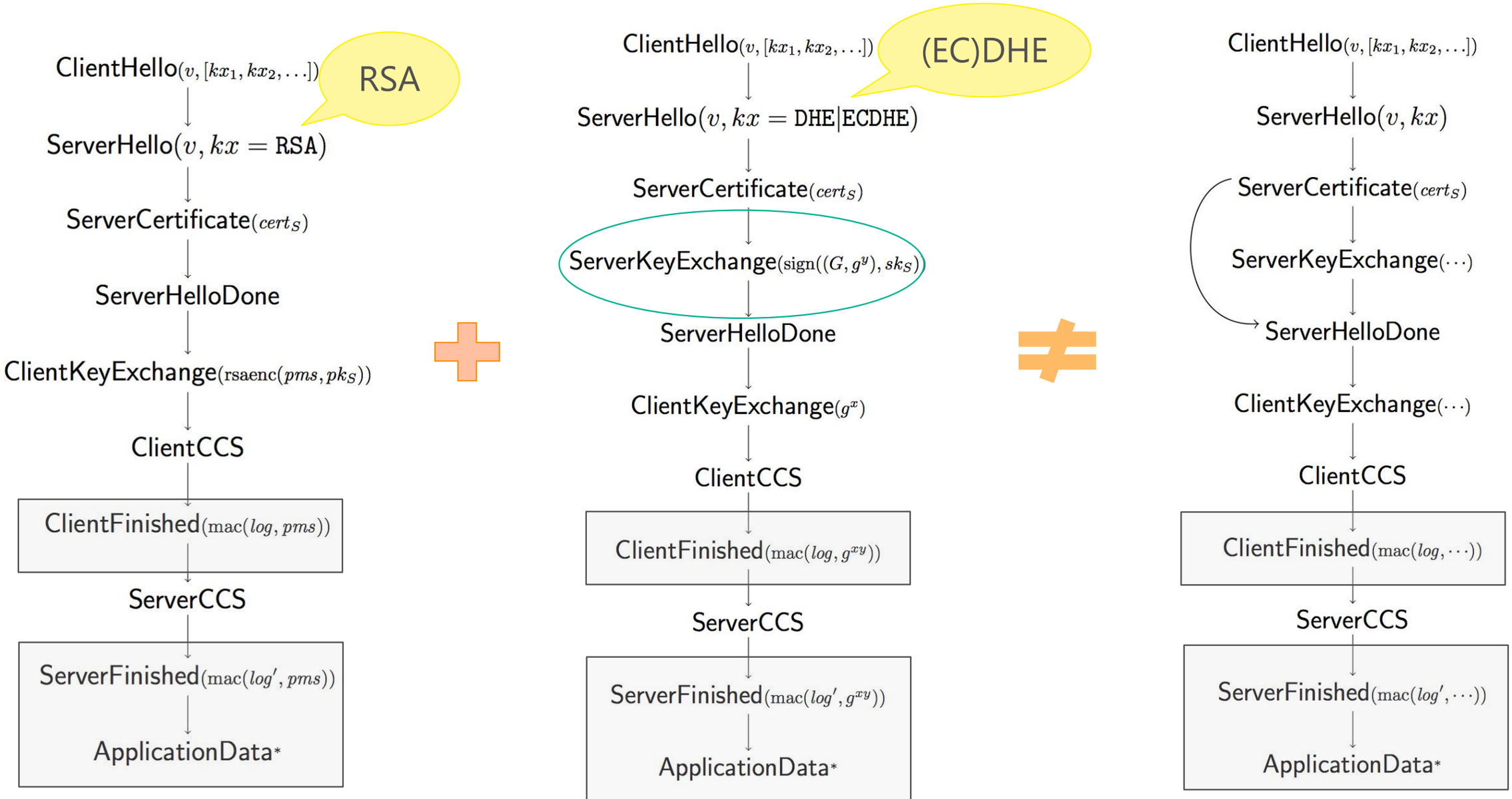
Unexpected state transitions in OpenSSL, NSS, Java, SecureTransport, …

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



OpenSSL
State Machine

Incorrectly Composing State Machines



Incorrectly Composing State Machines

Follows Postel's robustness principle

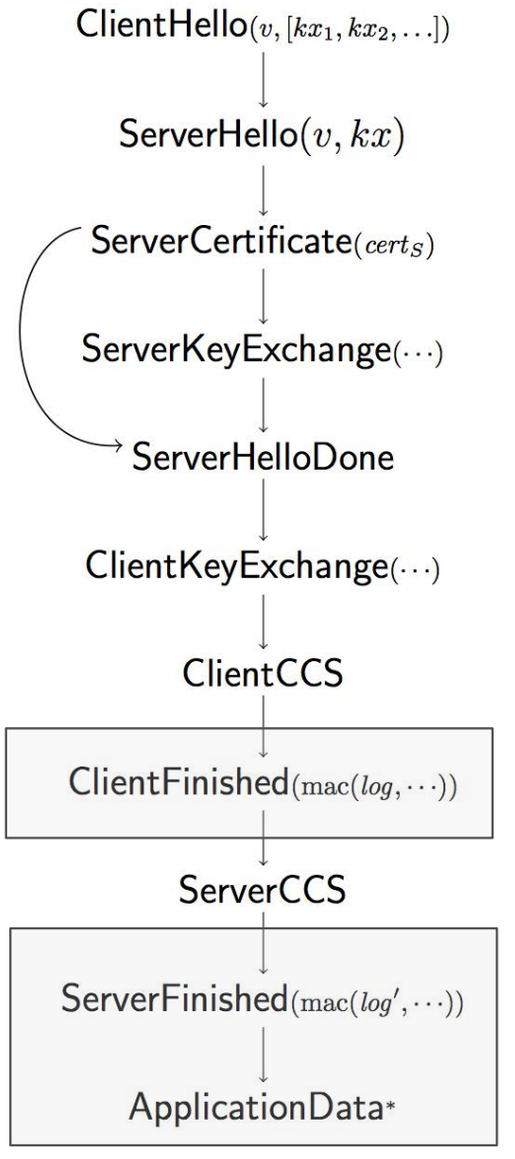
- "Be conservative in what you do,
  be liberal in what you accept from others"
  (BAD for security!)

Introduces unexpected cases at the client

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

Correct clients should reject these cases

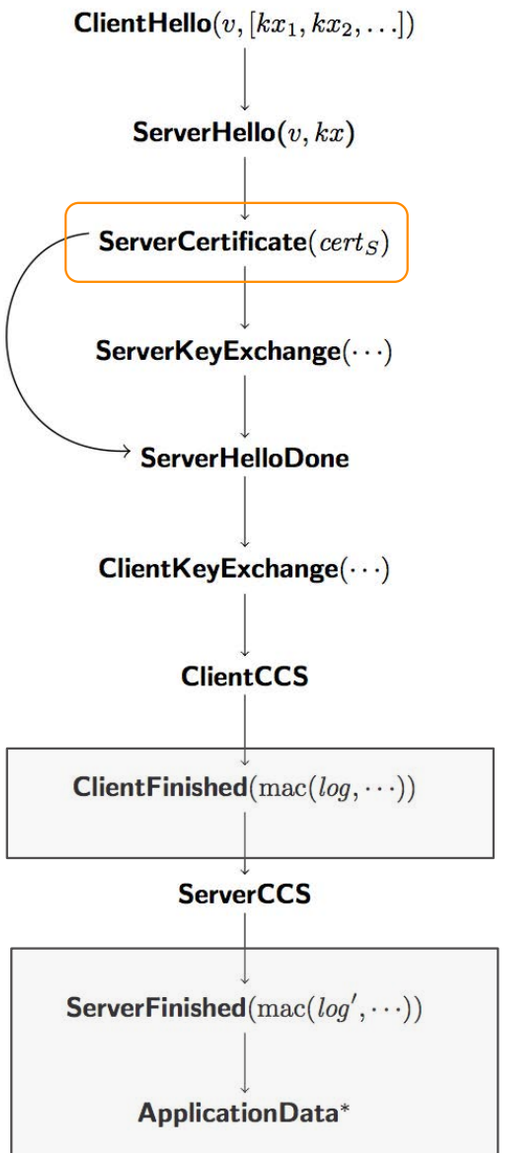
- Otherwise, they are not executing TLS anymore,
  and lose all its security guarantees



SKIP: skipping Messages to Java Clients

Network attacker impersonates api.paypal.com to a JSSE client

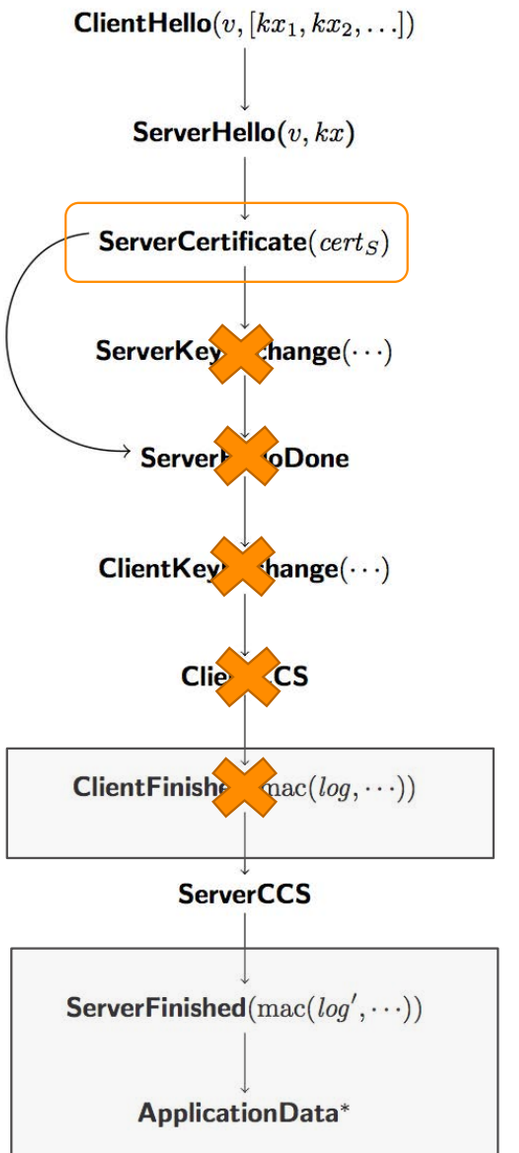
1. Send PayPal's cert



SKIP: skipping Messages to Java Clients

Network attacker impersonates api.paypal.com to a JSSE client

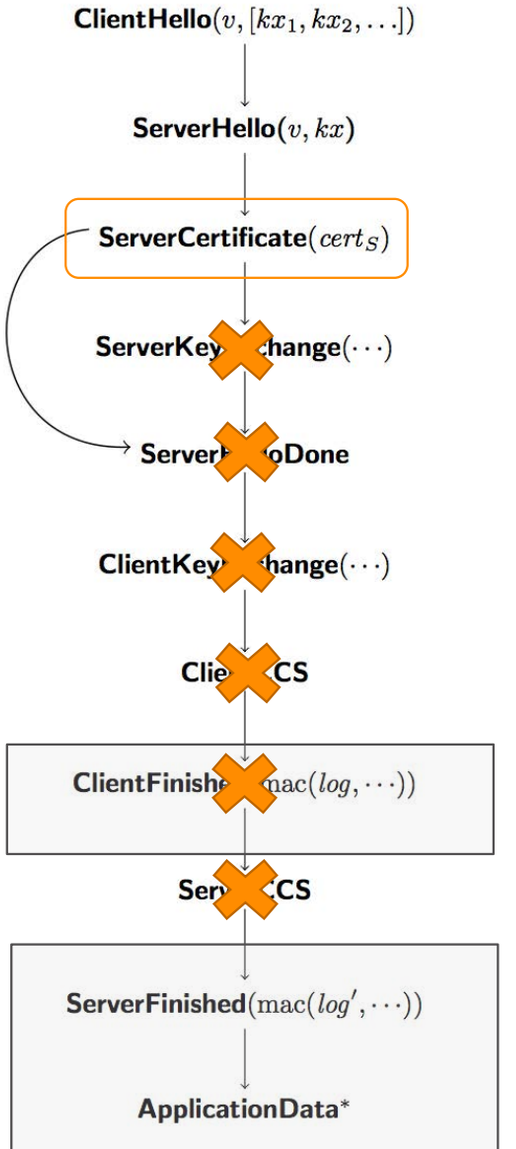
1. Send PayPal's cert
2. SKIP ServerKeyExchange (bypass server signature)
3. SKIP ServerHelloDone



SKIP: skipping Messages to Java Clients

Network attacker impersonates api.paypal.com to a JSSE client

1. Send PayPal'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



State Machine Attacks

Impact of SKIP on Java TLS Clients

- A network attacker can impersonate *any* server (Paypal, Amazon, Google) to *any* Java TLS client
- Affects all versions of Java until Jan 2015 CPU

Many other State Machine bugs in TLS libraries

- FREAK: combines crypto weakness, protocol flaw, and implementation bug

Recap: What goes wrong in TLS Deployments?

Crypto Weaknesses

- RC4, 3DES, MD5, PKCS#1 v1.5

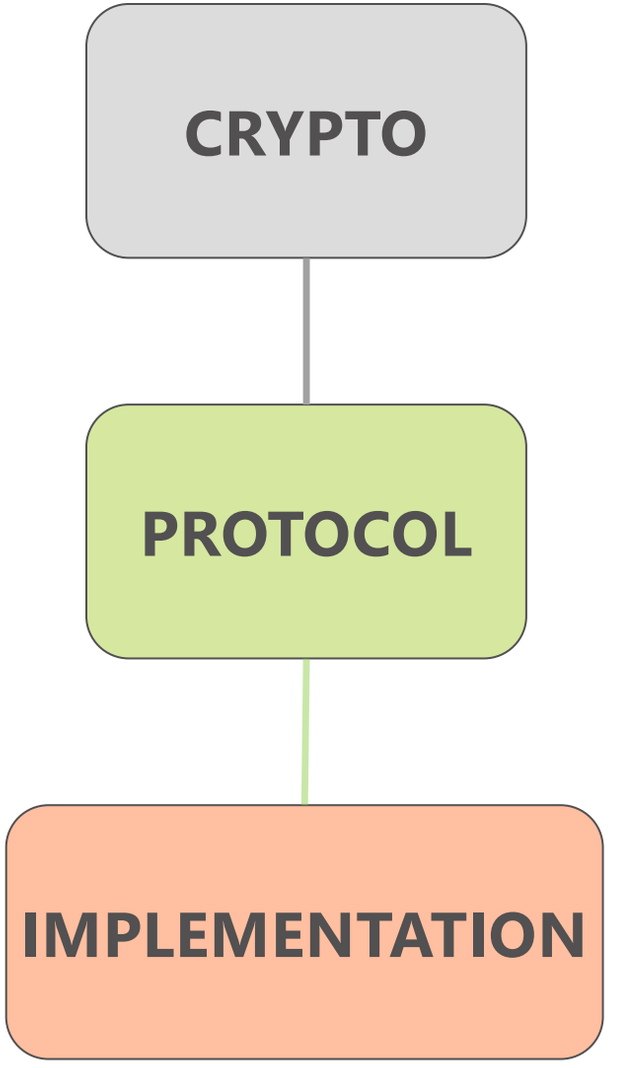
Protocol Design Flaws

- Downgrade attacks, Transcript collisions

Implementation Bugs

- State machine bugs, Heartbleed

Often, a mix of all of the above!



Idea: Implement the full TLS protocol stack in a proof-oriented programming language and formally verify its security and correctness

- Verified Crypto
  (correctness, memory safety, side-channel resistance)
- Verified Parsing
  (memory safety, correctness)
- Verified Protocol Code
  (state machine correctness, security proof)

Verifying Protocol Implementations in F*

F⋆: a security-oriented language and verification framework

- Functional programming language (« à la Ocaml »)
- Dependent type-and-effect system (« à la Coq »)
- Proof automation via the Z3 SMT solver
- Compilers to OCaml, C, WebAssembly

http://fstar-lang.org

Actively developed at Microsoft Research and Inria

Refinement types

A *refinement type* is a base type qualified with a logical formula;
the formula can express invariants, preconditions, postconditions, ...

Refinement types are types of the form $x : T\{C\}$ where
– T is the base type,
– x refers to the result of the expression, and
– C is a logical formula

The values of this type are the values M of type T such that $C\{M/x\}$ holds.

Refinement types, Pre- and Post-Conditions

```
// Sample type and value declarations in F*
type nat   = n:int{ 0 <= n }
val equal:  b:bytes -> b':bytes{ length(b) = length (b')} -> bool
```

```
// Sample cryptographic functional specification in F*
module AES
type key      // abstract type for secrets
type block = b:bytes{ Length(b)=16 }
val encrypt: k:key -> p:block -> c:block {c=AES(k,p)}
val decrypt: k:key -> c:block ->
             r:option block {forall k,p. c=AES(k,p) ==> r = Some p}
```

Modular Typing & Runtime Safety

Type safety implies that refinement formulas hold in all executions

- All invariants, pre- and post-conditions are satisfied
- Can be used to enforce a variety of correctness and security goals
- Type-checking is modular, one function at a time, so scales well

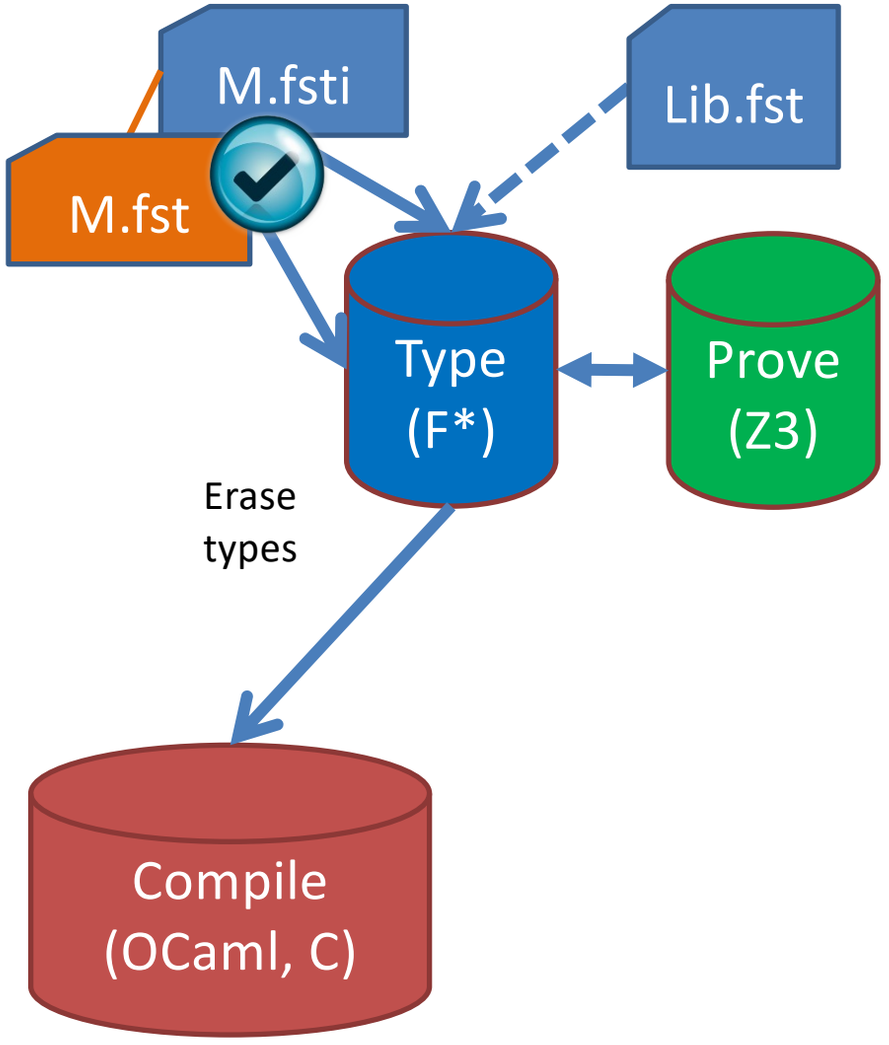
THEOREM 1 (Safety). If $\varnothing \vdash A : T$ then A is safe.

We write $I_0 \vdash A \rightsquigarrow I$ when,
in the typing environment I_0,
the module A is well-typed
and exports the interface I.

If $\varnothing \vdash A_0 \rightsquigarrow I_0$ and $I_0 \vdash A : T$, then $\varnothing \vdash A_0 \cdot A : T$.

F*: verification by type-checking + SMT

1. Write a program as an F* module
2. Specify its properties in an F* interface
3. Your program can call other verified user modules and trusted system libraries
4. Verify the module by typechecking:
   - F* generates proof obligations, and calls Z3, an SMT solver, to discharge them
5. Compile the module to executable code
   - Backends for OCaml, C, and WebAssembly



Verifying Protocol Components with F*

EverParse: verified zero-copy parsers in C [Usenix '19]

- Efficient parsers without memory safety bugs (i.e. no HeartBleed)
- Proofs of functional correctness for parsing
- Applied to TLS, Quic, (X.509), …

HACL*: verified crypto library in C and WebAssembly [CCS'17,'20]

- A full library of modern crypto algorithms
- Verified for memory safety, functional correctness, and secret independence (e.g. no branches on secrets)

Both written in Low*, a C-like subset of F* [ICFP '17]

- Verified code is compiled via KaRaMeL to C and WebAssembly

A Low* Spec for Stateful Crypto Code

```
// An array of secret (opaque) bytes
type block_p = b:buffer uint8{ length(b)=16 }

// Stateful in-place encryption
val encrypt: k:key_p -> p:block_p ->
             Stack unit
             (requires (fun h0 ->
                  live h0 [k;p] /\ disjoint k p))
             (ensures  (fun h0 _ h1 ->
                  modifies [p] h0 h1 /\
                  content h1 p ==
                  AES(content h0 k, content h0 p))
```

Verifying Protocol Implementations with F*

Verified protocol components

- EverParse: verified zero-copy parsers in C
- HACL*: verified crypto library in C and WebAssembly

miTLS: a cryptographically verified implementation of TLS

- Based on precise computational crypto assumptions
- Proofs require a combination of manual and F* proofs

DY*: a symbolic protocol verification framework in F*

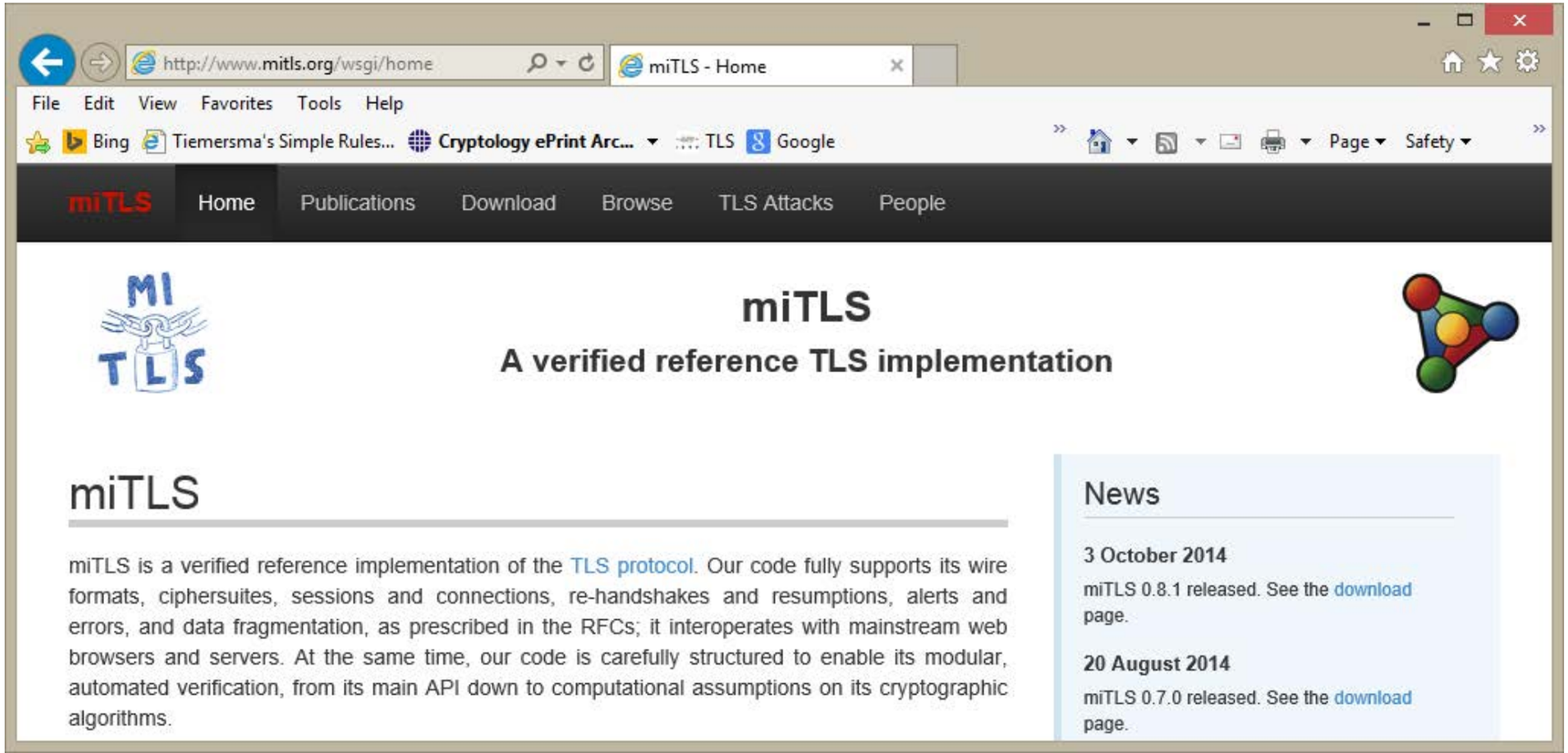
- Based on abstract symbolic crypto assumptions
- Proofs are fully mechanized in F*

Verified Security for Protocol Code: miTLS in F*

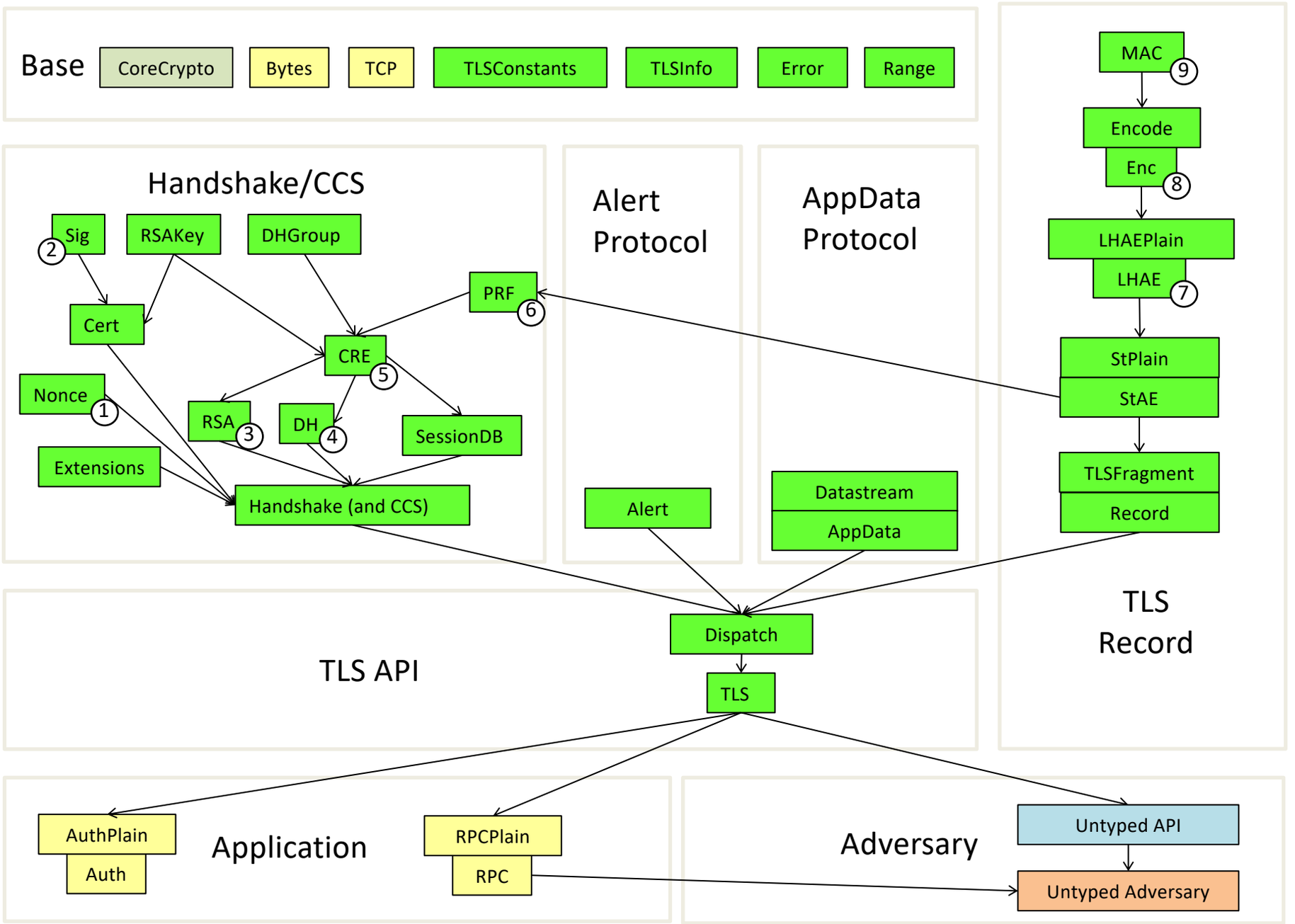
miTLS [2012-2015]

A verified reference implementation of TLS

- Covers TLS 1.0-1.2
- Covers all major protocol modes and ciphersuites



Modular Architecture for miTLS



miTLS Security theorem

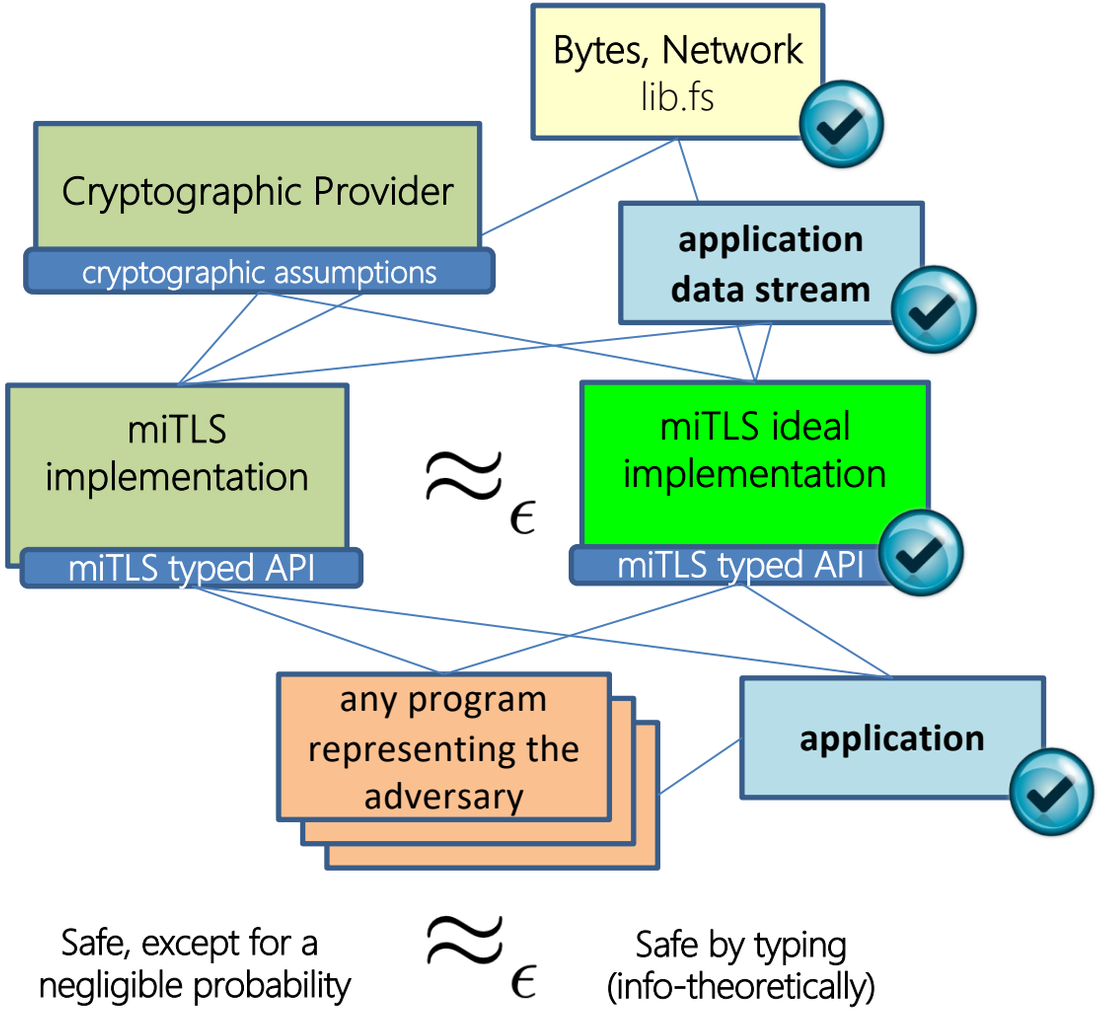
Main crypto theorem: concrete TLS & ideal TLS are computationally indistinguishable

uses F* typing + crypto reasoning

We then prove that ideal miTLS meets its secure channel spec

uses F* typing

We transform one module at a time based on a precise crypto assumption



Example: Message Authentication Codes

```
module MAC

type text = bytes      val macsize
type key  = bytes
type mac  = bytes


val GEN   : unit -> key
val MAC   : key -> text -> mac
val VERIFY: key -> text -> mac -> bool
```

basic F* interface

This interface says nothing on the security of MACs.

Specifying Message Authentication Codes

MAC keys are abstract

```
module MAC

type text = bytes      val macsize
type key
type mac  = bytes


val GEN   : unit -> key
val MAC   : key -> text -> mac
val VERIFY: key -> text -> mac -> bool
```

Specifying Message Authentication Codes

MAC keys are abstract

```
module MAC
type text = bytes      val macsize
type key
type mac  = b:bytes{Length(b)=macsize}


val GEN   : unit -> key
val MAC   : key -> text -> mac
val VERIFY: key -> text -> mac -> bool
```

MACs are fixed sized

Specifying Message Authentication Codes

ideal F* interface

MAC keys are abstract

```
module MAC
type text = bytes      val macsize
type key
type mac  = b:bytes{Length(b)=macsize}
predicate Msg of key * text
val GEN    : unit -> key
val MAC    : k:key -> t:text{Msg(k,t)} -> mac
val VERIFY: k:key -> t:text -> mac
              -> b:bool{ b=true => Msg(k,t)}
```

MACs are fixed sized

Msg is specified by protocols using MACs

"All verified messages have been MACed"

Specifying Message Authentication Codes

MAC keys are abstract

```
module MAC
type text = bytes      val macsize
type key
type mac  = b:bytes{Length(b)=macsize}
predicate Msg of key * text
val GEN   : unit -> key
val MAC   : k:key -> t:text{Msg(k,t)} -> mac
val VERIFY: k:key -> t:text -> mac
            -> b:bool{ b=true => Msg(k,t)}
```

MACs are fixed sized

Msg is specified by protocols using MACs

"All verified messages have been MACed"

This can't be true! (collisions)

ideal F* interface

```
module MAC
open System.Security.Cryptography
let macsize = 20
let GEN()   = randomBytes 16
let MAC k t = (new HASHMACSHA1(k)).ComputeHash t
let VERIFY k t m = (MAC k t = m)
```

concrete F* implementation (using real crypto)

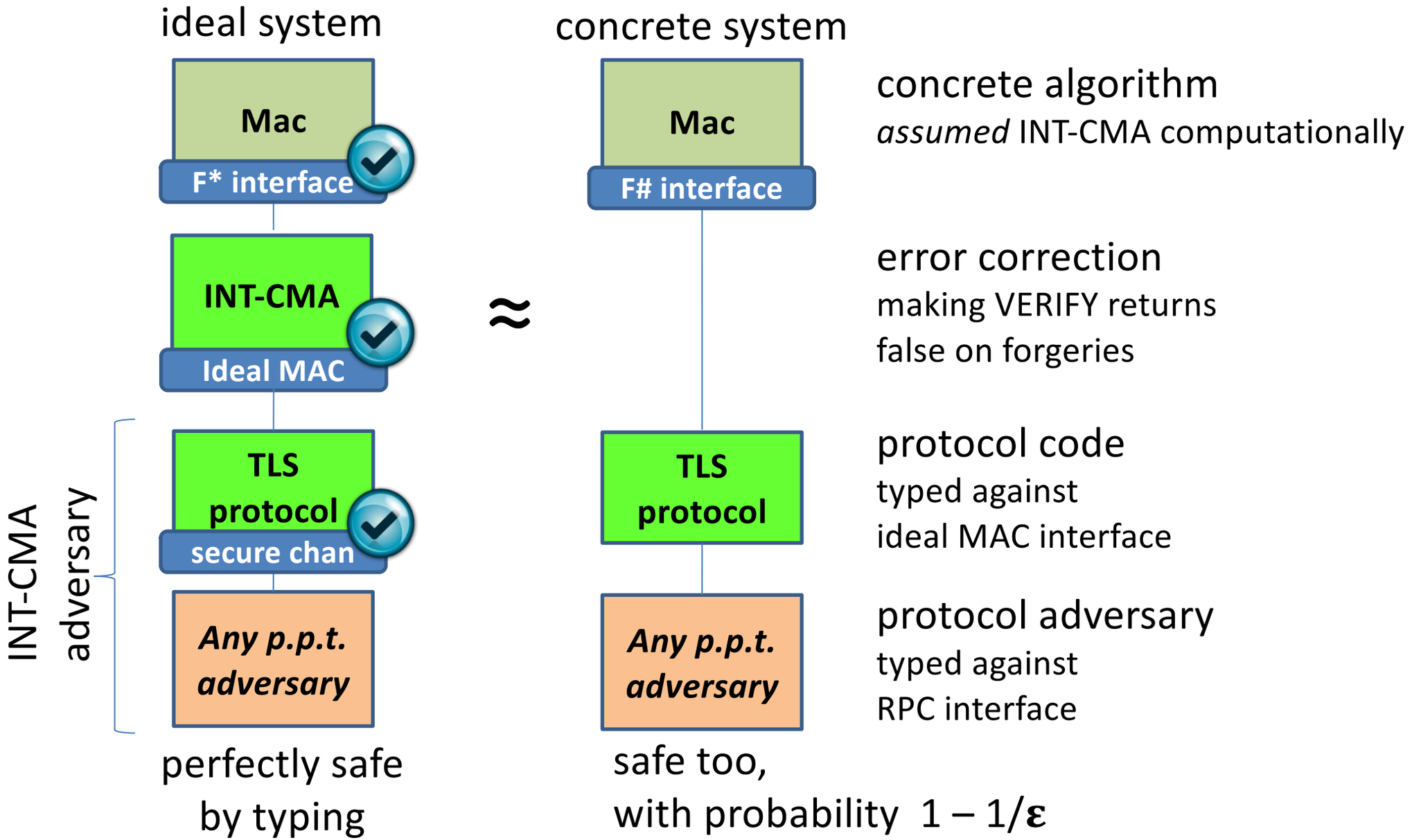
Crypto Assumption: INT-CMA
Resistance to Chosen-Message Existential Forgery Attacks

```
module INT_CMA_Game
open Mac
Let private k = GEN()
let private log = ref []
let mac t =
  log := t::!log
  MAC k t
let verify t m =
  let v = VERIFY k t m in
  if v && not (mem t !log) then FORGERY
  v
```

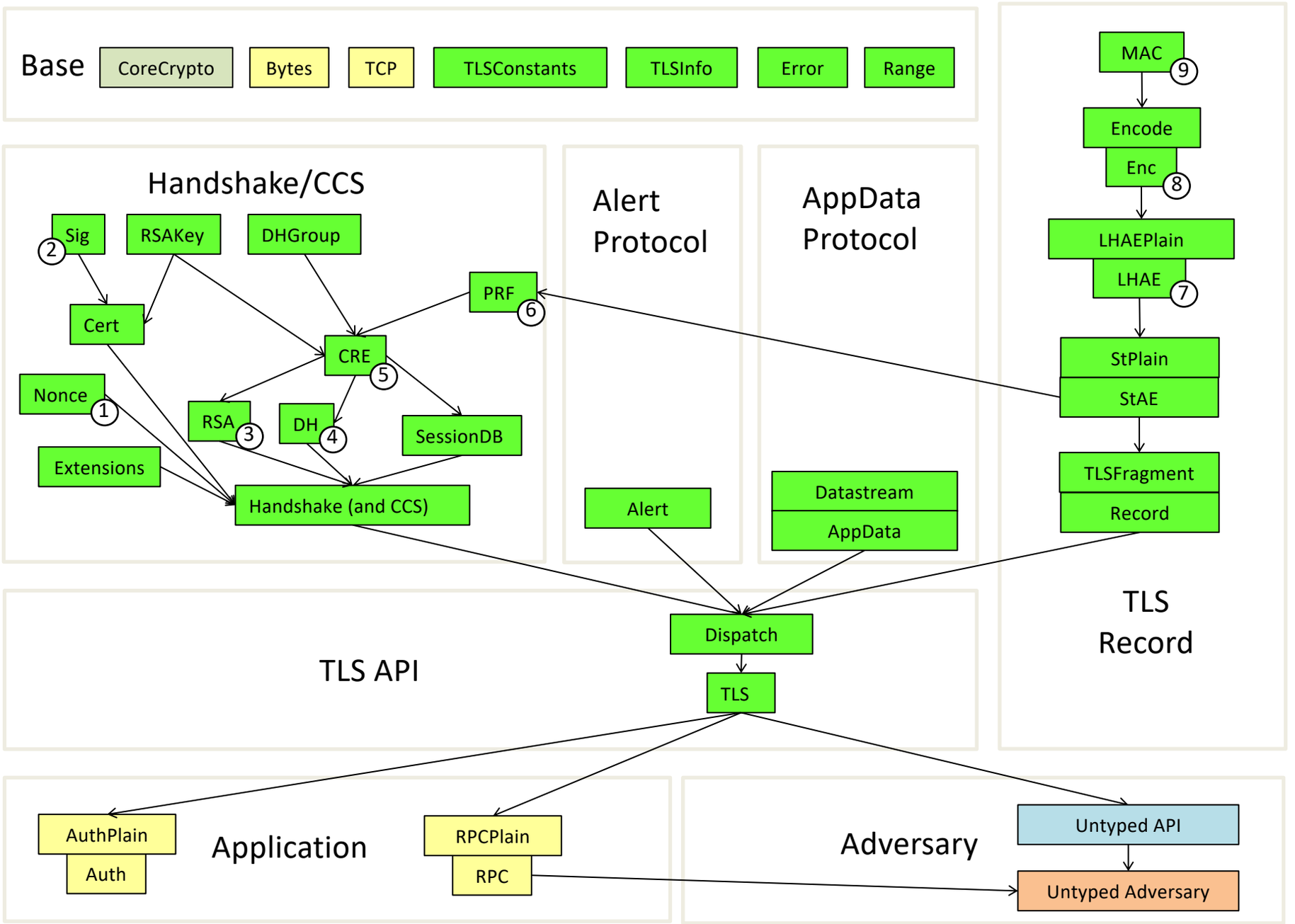
Computational Safety
a probabilistic polytime program calling **mac** and **verify** forges a MAC only with negligible probability [2]

CMA game
(coded in F*)

Computational Safety for Ideal MACs



Modular Architecture for miTLS



A High-Level Secure Channel API for TLS

- Secrecy and authenticity for application data
- Multiple ciphers (Agile)
- Multiple protocol modes
- Accounts for key compromise

Prevents large classes of attacks

- no state machine bugs
- no downgrade attacks

```
type cn // for each local instance of the protocol

// creating new client and server instances
val connect: TcpStream -> params -> (;Client) nullCn Result
val accept:  TcpStream -> params -> (;Server) nullCn Result

// triggering new handshakes, and closing connections
val rehandshake: c:cn{Role(c)=Client} -> cn Result
val request:     c:cn{Role(c)=Server} -> cn Result
val shutdown:    c:cn -> TcpStream Result

// writing data
type (;c:cn,data:(;c) msg_o) ioresult_o =
| WriteComplete of c':cn
| WritePartial  of c':cn * rest:(;c') msg_o
| MustRead      of c':cn
val write: c:cn -> data:(;c) msg_o -> (;c,data) ioresult_o

// reading data
type (;c:cn) ioresult_i =
| Read      of c':cn * data:(;c) msg_i
| CertQuery of c':cn
| Handshake of c':cn
| Close     of TcpStream
| Warning   of c':cn * a:alertDescription
| Fatal     of a:alertDescription
val read : c:cn -> (;c) ioresult_i
```

miTLS Impact and Verification Effort

First verified implementation of real-world protocol

- 3600 lines of code
- 2000 lines of type annotations
- 30 pages of crypto proofs,
  supported by 3000 lines of EasyCrypt proof
- 3 years of concerted effort by a team of 5-10 researchers

Measurable impact on real-world protocol design

- Helped find attacks on TLS: Triple Handshake, SKIP, FREAK, Logjam
- Influenced design of TLS 1.3 at IETF

Can we scale up and generalize this approach to other protocols?

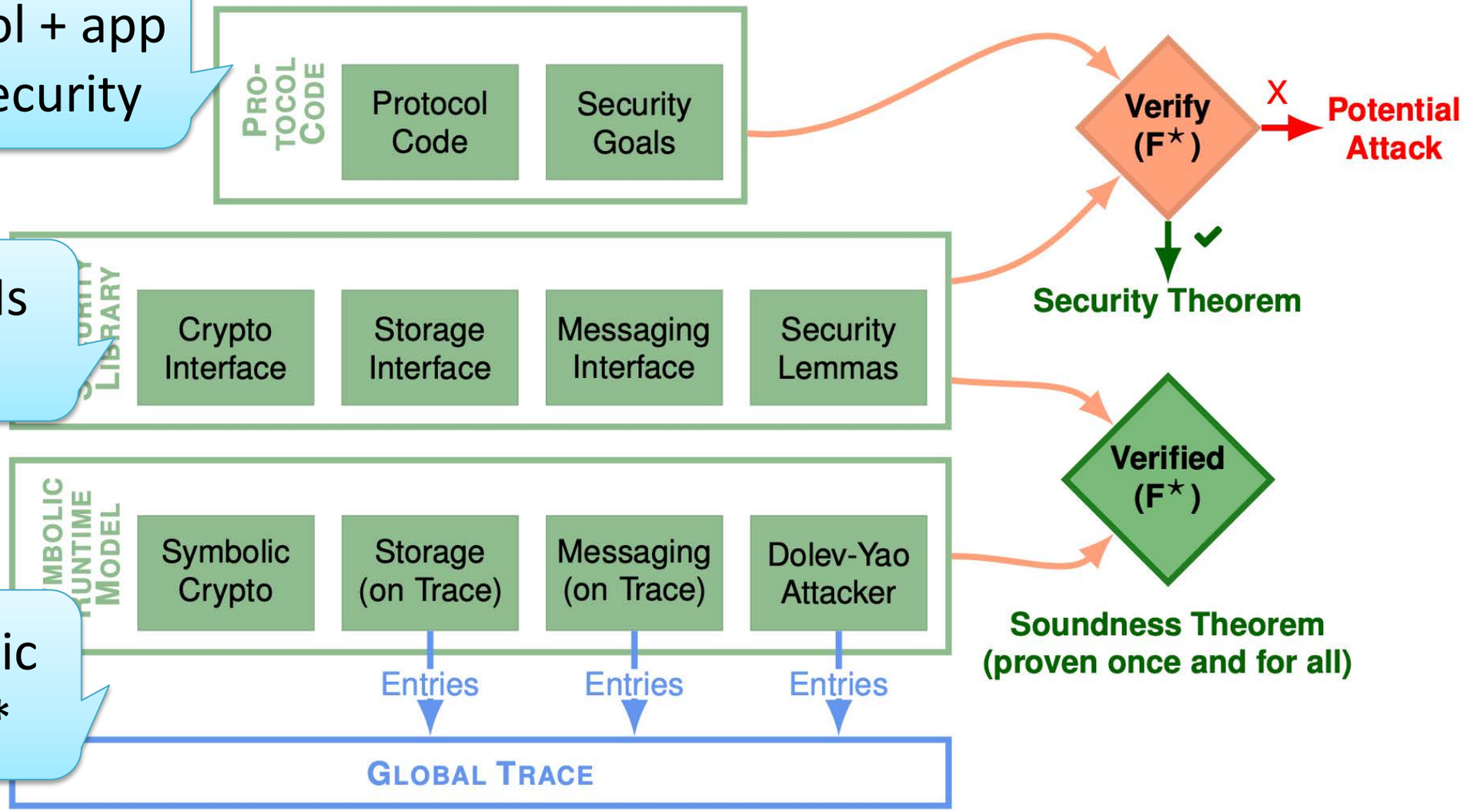
DY* Verification Architecture

[Euro S&P 2021, ACM CCS 2021]

Executable protocol + app code verified for security

Abstract labeled APIs proved sound in F*

Trace-based symbolic runtime model in F*



A taste of DY*

```
type message =
| Msg1: i:principal → n_i: bytes → message
| Msg2: n_i: bytes → n_r:bytes → message
| Msg3: n_r: bytes → message
val serialize_message: message → bytes
val parse_message: bytes → result message
val parse_message_correctness_lemma: m:message →
   Lemma (parse_message (serialize_message m) == Success m)
```

Precise Message Formats

- serialization and parsing with correctness proofs

```
type session_st =
| SecretKey: secret_key: bytes → session_st
| PublicKey: peer:principal → public_key:bytes → session_st
| ISentMsg1: r:principal → n_i:bytes → session_st
| RSentMsg2: i:principal → n_i:bytes → n_r:bytes → session_st
| ISentMsg3: r:principal → n_i:bytes → n_r:bytes → session_st
| RReceivedMsg3: i:principal → n_r:bytes → session_st
val serialize_session_st: session_st → bytes
val parse_session_st: bytes → result session_st
```

Protocol State Machine

- Stateful protocol code
- Session state storage
- Fine-grained compromise

A taste of DY*

Typed Crypto API encodes symbolic crypto assumptions

Using secrecy labels and authentication predicates

```
val pke_enc: #i:nat → #l:label → #s:string →
      public_enc_key i l s →
      m:msg i l{pke_pred m} → msg i Public
val pke_dec: #i:nat → #l:label → #s:string →
      private_dec_key i l s → msg i Public →
      result (m:msg i l{is_publishable i m ∨ pke_pred m})
```

Security Label in Signal Protocol

```
Meet (Join (Can_Read [P i]) (Can_Read [P r]))
 (Meet (Join (Can_Read [V i sid_i 0]) (Can_Read [P r]))
  (Meet (Join (Can_Read [V i sid_i 0]) (Can_Read [P r]))
        (Join (Can_Read [V i sid_i 0]) (Can_Read [V r sid_r 0]))))
```

DY*: scalable symbolic security verification

	Modules	FLoC	PLoC	Verif. Time	Primitives
Generic DY*	9	1,536	1,344	≈ 3.2 min	-
NS-PK	4	439	-	(insecure)	PKE
NSL	5	340	188	≈ 0.5 min	PKE
ISO-DH	5	424	165	≈ 0.9 min	DH, Sig
ISO-KEM	4	426	100	≈ 0.7 min	PKE, Sig
Signal	8	836	719	≈ 1.5 min	DH, Sig, KDF, AEAD, MAC

- Proofs require between 50% and 90% annotation overhead
- Verification time grows linearly with protocol size

Conclusions

End-to-End verification of protocol stacks is now feasible

- Using proof-oriented programming languages like F*
- Crypto (HACL*), Parsers (EverParse), Protocols (miTLS, DY*)

Area is still maturing and is under active research

- Verifying optimized low-level code in C and assembly
- Verifying protocols that use ZK proofs, or MPC, or FHE, or PQ

Many open problems for future work

- Proving the absence of side channel attacks
- Verifying code written by non-verification experts