Breaking and Fixing Token-based Web Authentication

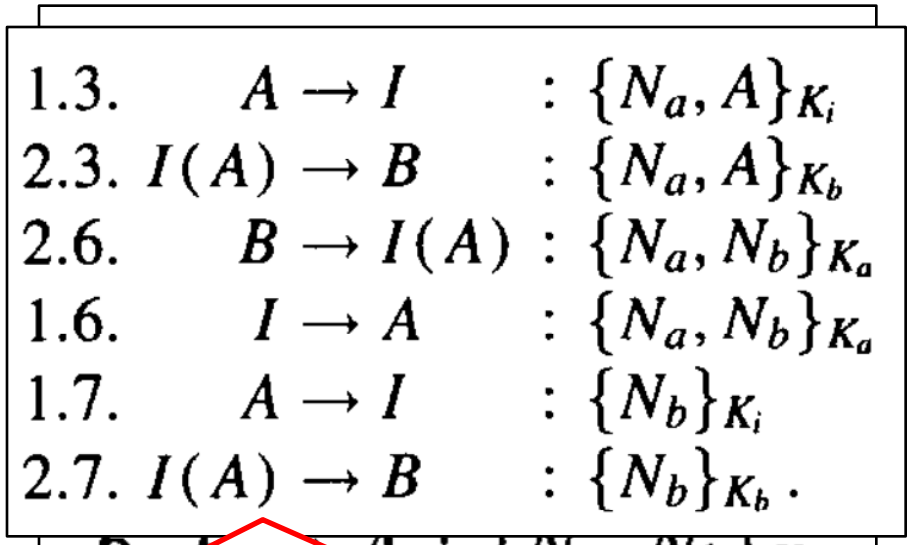
Karthik Bhargavan

Joint work with
*A Delignat-Lavaud, C Bansal, S Maffeis, G Leurent, and the **miTLS** team*

OAuth Workshop, Trier 2016

A need for verified protocols

- Long history of broken authentication protocols
  - Even those using standard strong crypto
  - Even those designed by famous cryptographers
  - Even those used in widely used frameworks like Kerberos, EAP, etc.
- How do we verify that we got it right?



$$
\begin{aligned}
&1.3. \quad A \rightarrow I \quad : \{N_a, A\}_{K_i} \\
&2.3. \quad I(A) \rightarrow B \quad : \{N_a, A\}_{K_b} \\
&2.6. \quad B \rightarrow I(A) \quad : \{N_a, N_b\}_{K_a} \\
&1.6. \quad I \rightarrow A \quad : \{N_a, N_b\}_{K_a} \\
&1.7. \quad A \rightarrow I \quad : \{N_b\}_{K_i} \\
&2.7. \quad I(A) \rightarrow B \quad : \{N_b\}_{K_b} .
\end{aligned}
$$

Lowe's Man-in-the-Middle Attack on Needham-Schroeder [1995]

If A logs into malicious server I,
I can then login as A at B

A logs into B using trusted server S

A need for verified protocols

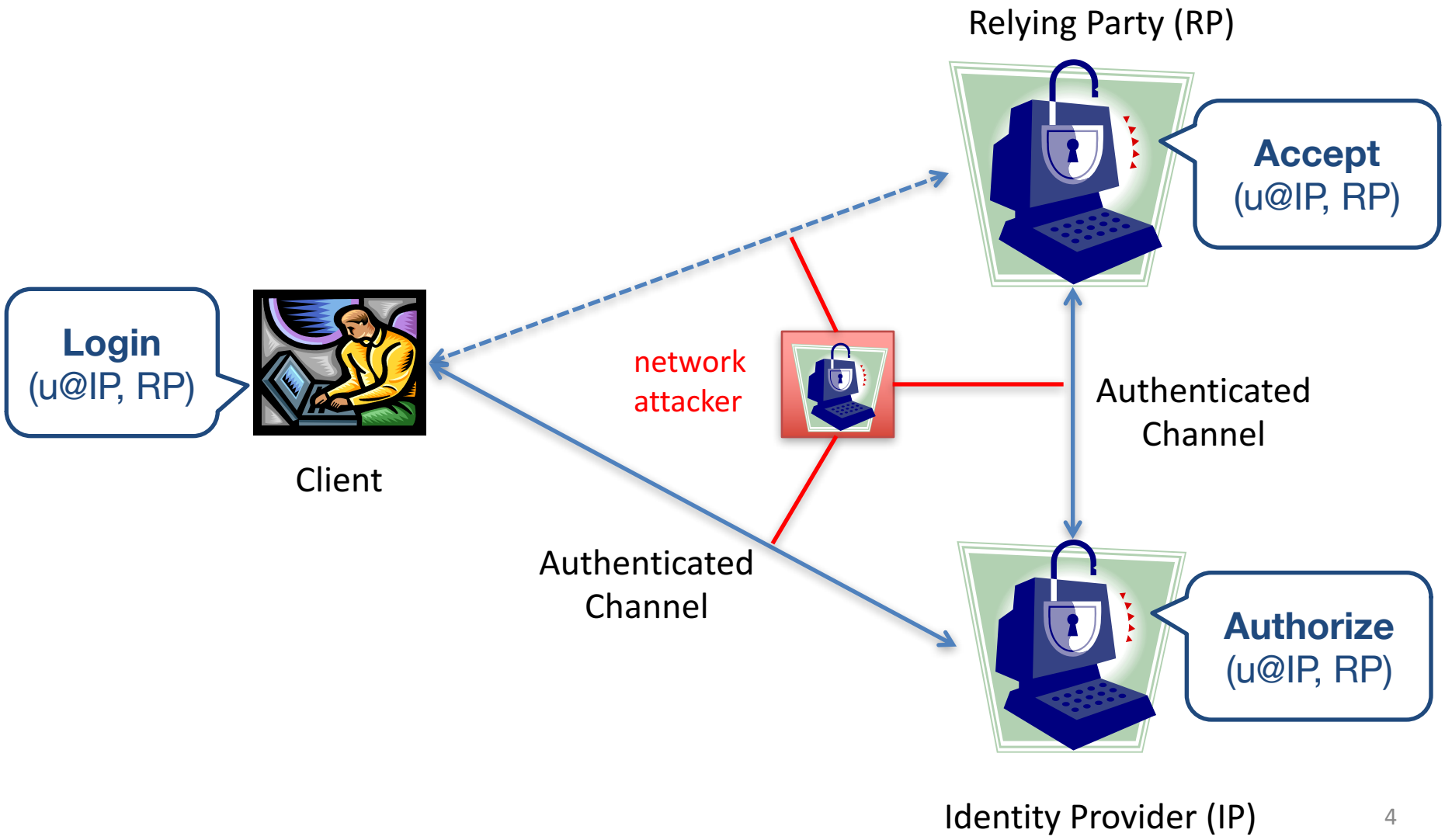
- A long line of research on formal protocol analysis
  - Logics for authentication and authorization (BAN)
  - Tools for automated symbolic verification (ProVerif)
  - Tools for computational crypto proofs (EasyCrypt)
- Impact on TLS 1.2 & 1.3
  - Many bugs, many proofs
  - Can they apply to OAuth?

$$
\begin{array}{lrcl}
1.3. & A & \rightarrow I & : \{N_a, A\}_{K_i} \\
2.3. & I(A) & \rightarrow B & : \{N_a, A\}_{K_b} \\
2.6. & B & \rightarrow I(A) & : \{N_a, N_b\}_{K_a} \\
1.6. & I & \rightarrow A & : \{N_a, N_b\}_{K_a} \\
1.7. & A & \rightarrow I & : \{N_b\}_{K_i} \\
2.7. & I(A) & \rightarrow B & : \{N_b\}_{K_b} .
\end{array}
$$

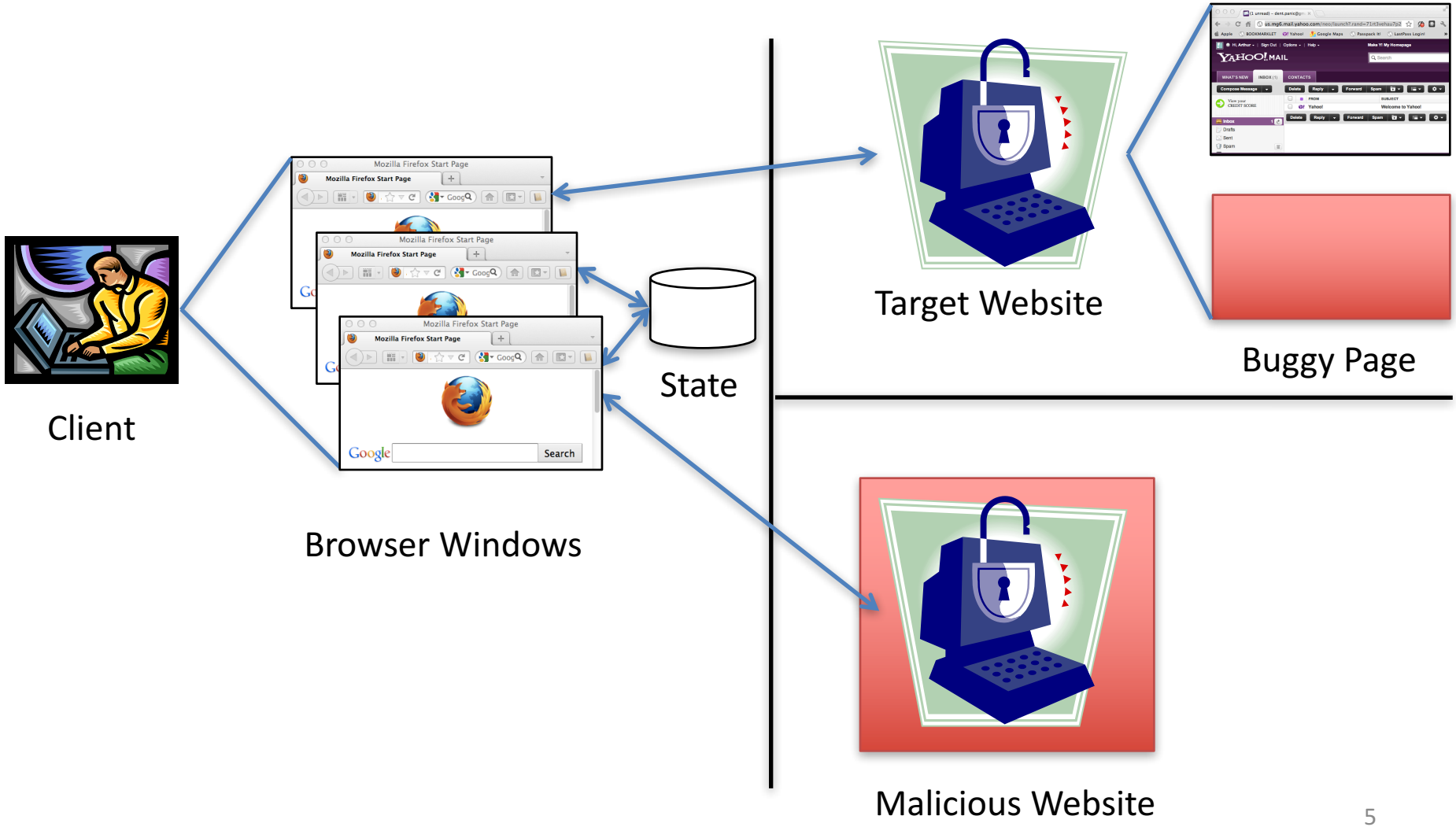
Lowe's Man-in-the-Middle Attack on Needham-Schroeder [1995]

If A logs into malicious server I,
I can then login as A at B

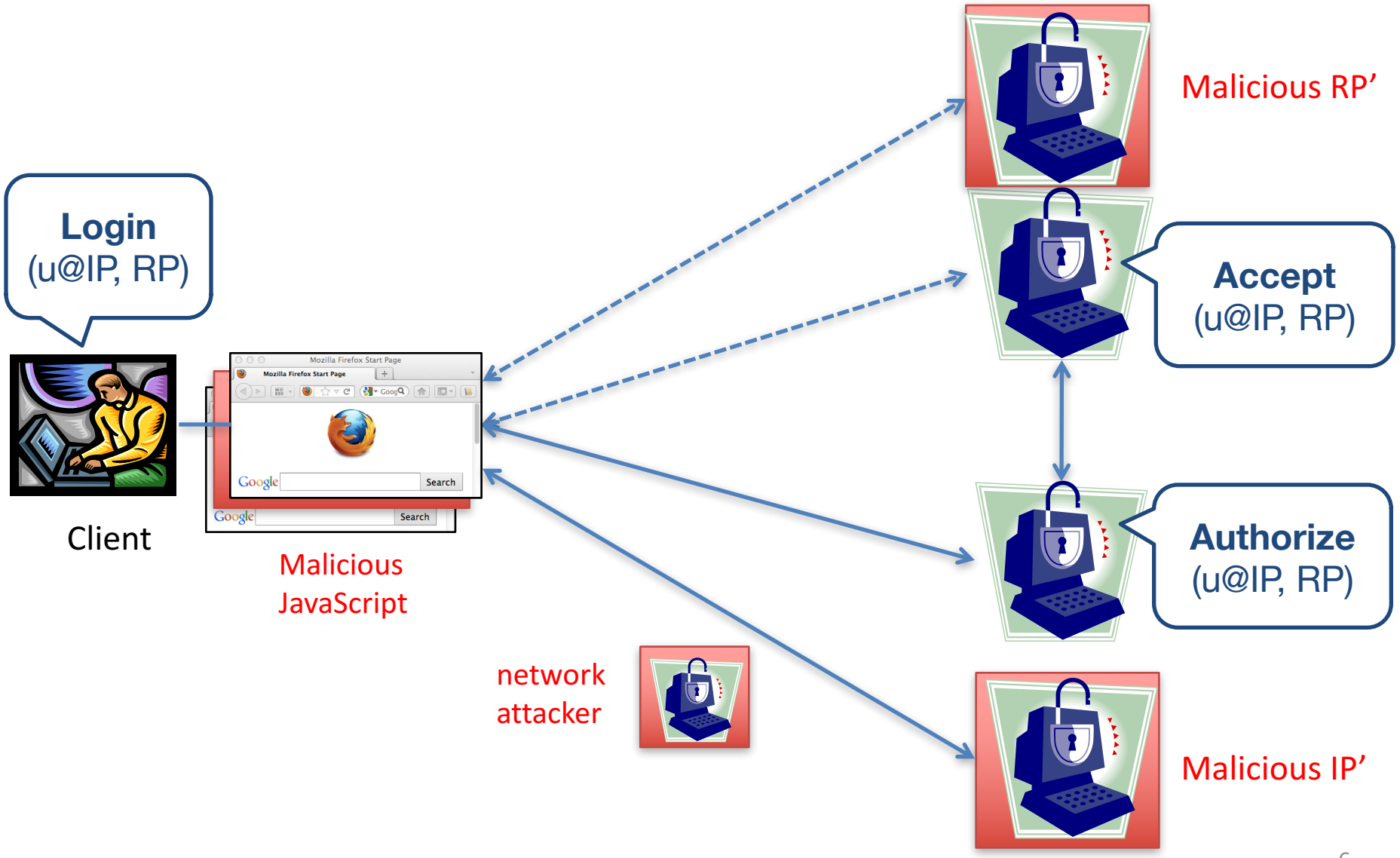
Classic 3-Party Authentication



Web Attacker Model



Web 3-Party Authentication



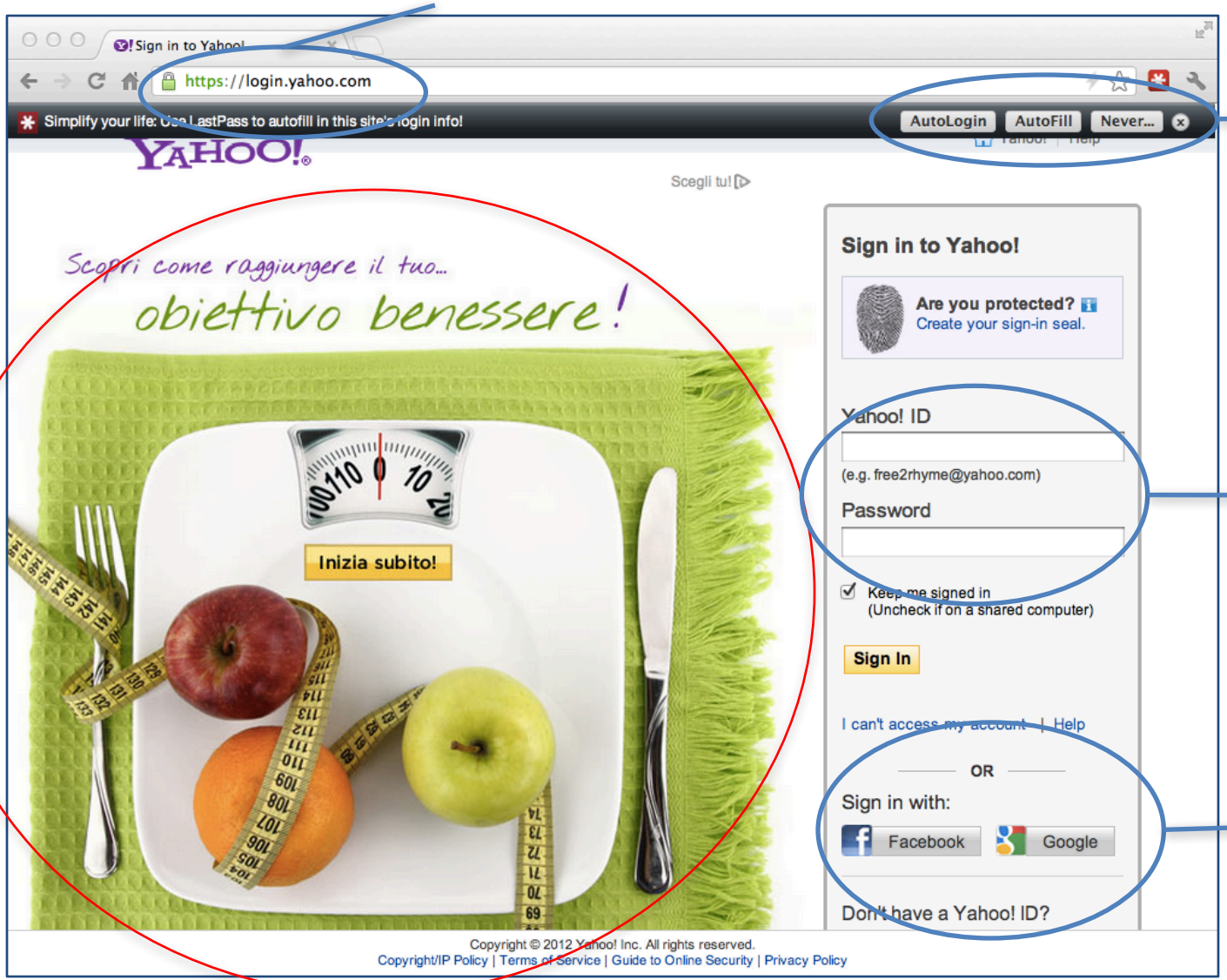
Robustness Against Web Attackers

- Problem: network attackers
  - Solution: HTTPS
- Problem: malicious websites
  - Solution: same-origin policy
- Problem: buggy web pages
  - Solution: use best practices

- *Challenge*: Can we formally prove that our deployed protocol is *robust* against such attacks?
  - Are we making the right assumptions about HTTPS?
  - Are we using SOP correctly?
  - Can we use techniques from protocol analysis?

Modern Website Login

HTTPS Protocol *(HTTP over TLS)*



Advert *(Untrusted)*

Password Manager *(Encrypted Cloud Storage)*

Login Form *(password)*

Single Sign-On *(OAuth, OpenID)*

Facebook Login at Yahoo (OAuth)

a is the browser, y is Yahoo, f is Facebook
Assume a is logged into f
All messages are over TLS

1. $a \rightarrow y$	Request(LoginWithFacebook())
2. $y \rightarrow a$	Redirect(`https://f/`TokenRequest(y,perms,return))
3. $a \rightarrow f$	Request[$\text{sid}_{u,f}$] (TokenRequest(y,perms,return))
4. $f \rightarrow a$	Response[$\text{sid}_{u,f}$](AuthorizationForm(y,perms))
5. $a \rightarrow f$	Request[$\text{sid}_{u,f}$](Authorize(y,perms))
6. $f \rightarrow a$	Redirect[$\text{sid}_{u,f}$](`https://y/`return`?access_token=XXX`)
7. $a \rightarrow y$	Request(return,`access_token=XXX`)
8. $y \rightarrow f$	Request(API.GetId(`XXX`))
9. $f \rightarrow y$	Response($u@f$)
10. $y \rightarrow a$	Response[$\text{sid}_{u@f,y}$](SocialLoginSuccess())

Simplest user-agent flow: 8 HTTPS exchanges (including login)

Double redirection protocol: Yahoo to Facebook and back

Main Security Goals for FB Login

- *User Authentication:*
  If Yahoo logs in Alice@fb.com then
  - Alice tried to login to Yahoo through Facebook, and
  - Facebook authenticated Alice, and
  - Alice authorized Yahoo to get her Facebook Id

- *API Authorization:*

  If Facebook accepts an API call for Alice's data
  - Alice must have authorized some website, and
  - that website must have made this API call

- *Both goals depend upon the secrecy of access tokens*

Stealing OAuth Access Tokens

- Dozens of ways of stealing OAuth tokens
  - Applies to websites, smartphone apps, …
  - See Andrey's talk tomorrow
- Stealing a user's access token is valuable
  - You can impersonate the user (e.g. to vote on Helios)
  - You can steal user's data (e.g. her Facebook profile)

- Let's focus on secure token delivery from IP to RP
  - IP redirects user's browser to https://RP.com/login/?access_token=XXX&…
  - How do we protect the token XXX from the attacker?

Stealing OAuth Access Tokens

- Let's focus on secure token delivery from IP to RP
  - IP redirects user's browser to https://RP.com/login/?access_token=XXX&…
  - How do we protect the token XXX from the attacker?

- We rely on many strong assumptions
  - TLS provides a secure channel to RP.com:443
  - HTTP routes requests correctly to /login
  - SOP protects token from other malicious websites
  - RP does not leak token by web vulnerabilities
  - Are these assumptions reasonable?

Transport Layer Security (1994—)

The default secure channel protocol?

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

20 years of attacks, fixes, and extensions

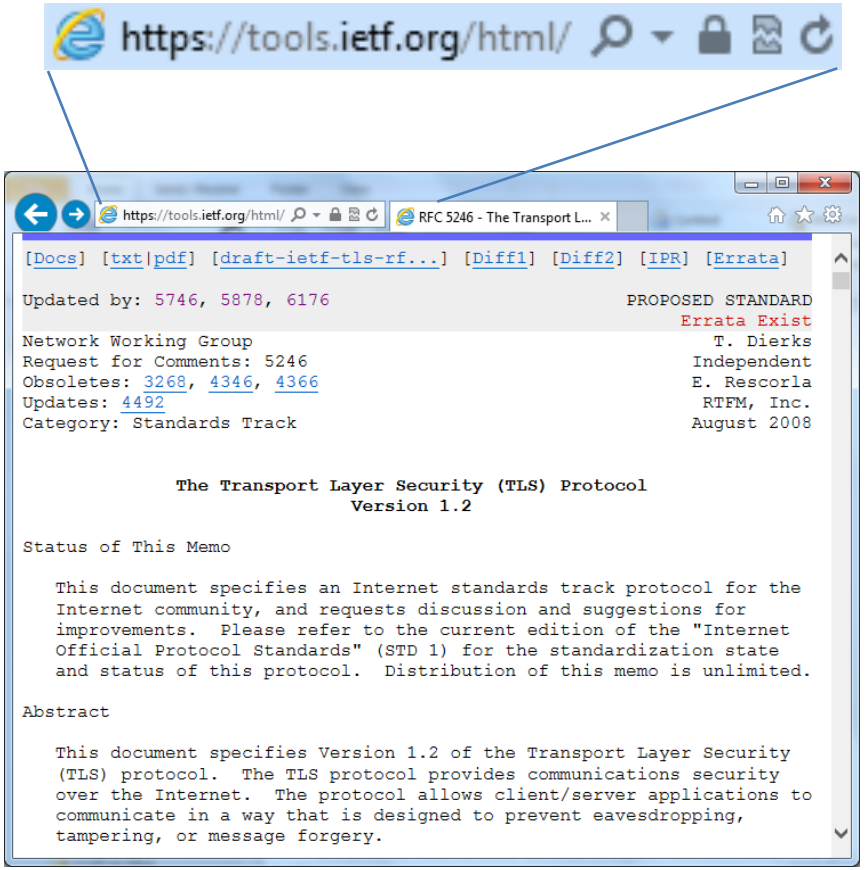
1994	Netscape's Secure Sockets Layer
1996	SSL3
1999	TLS1.0 (RFC2246)
2006	TLS1.1 (RFC4346)
2008	TLS1.2 (RFC5246)
2015	TLS1.3?

Many implementations

OpenSSL, SecureTransport, NSS,
SChannel, GnuTLS, JSSE, PolarSSL, …
many bugs, attacks, patches every year

Many papers

Well-understood, detailed specs
many security theorems…
mostly for small simplified models of TLS



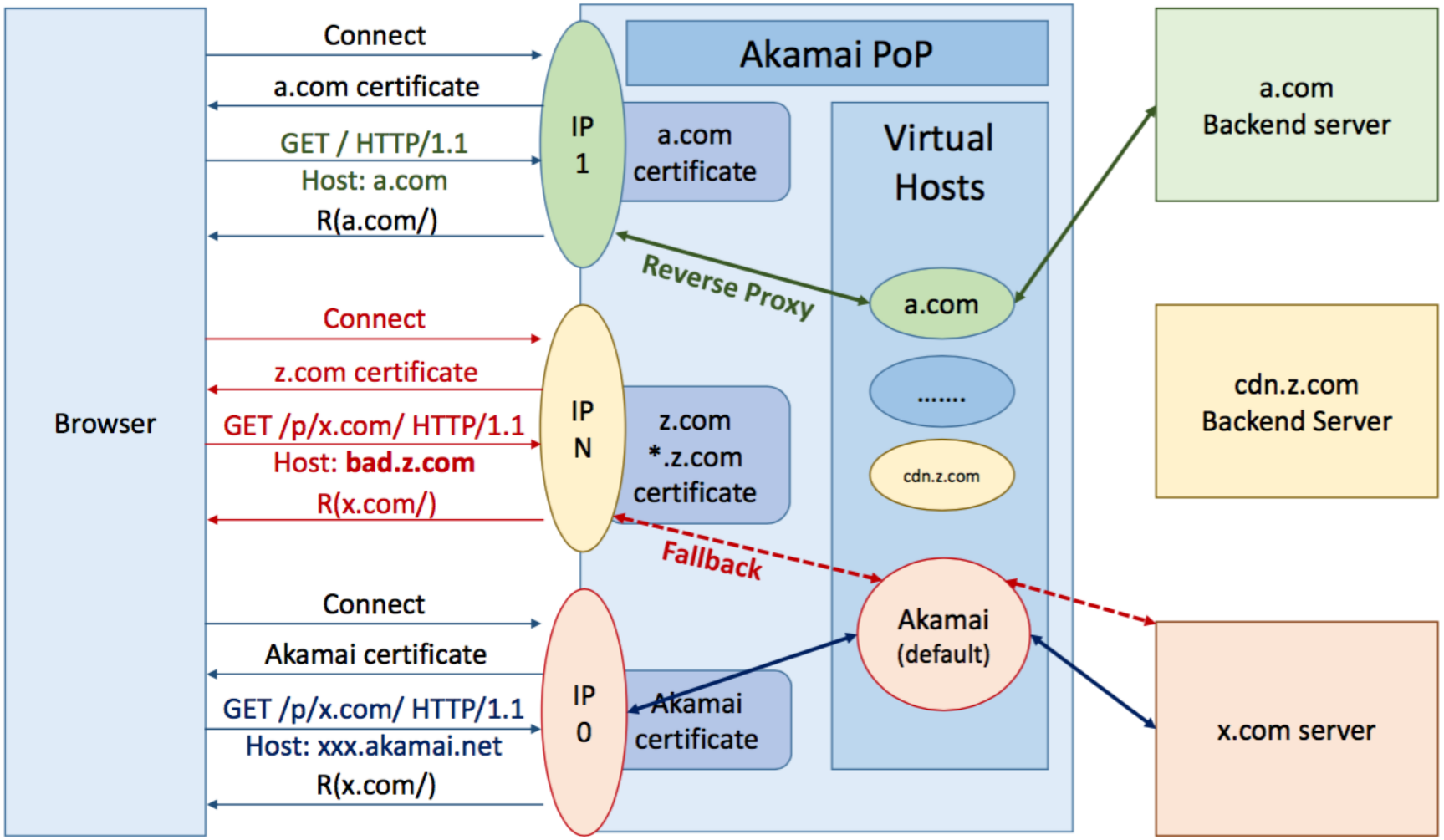
Attacks on TLS

- TLS provides a secure channel to RP.com:443
  - https://RP.com/login/?access_token=XXX&…
- But TLS deployments are easy to get wrong
  - POODLE — TLS 1.2 → SSLv3 — [Dec'14]
  - FREAK — RSA-2048 → RSA-512 — [Mar'15]
  - LOGJAM — DH-2048 → DH-512 — [May'15]
  - SLOTH — RSA-SHA256 → RSA-MD5 — [Jan'16]
  - DROWN — TLS 1.2 -> SSLv2 — [Mar'16]
- FREAK broke 25% of websites [IEEE S&P 2015]
  - Including https://connect.facebook.net/en_us/all.js
  - FREAK was found by systematic protocol analysis

HTTP virtual host confusion

- HTTP routes requests correctly to /login
  - https://oauth.RP.com/login/?access_token=XXX&…
- Virtual Host Confusion attacks [WWW 2015]
  - Suppose RP.com is hosted by a CDN alongside other (possibly malicious) websites
  - A network attacker confuse the web server into delivering the token to the wrong virtual host
  - General attack vector that breaks many origin-based protections
  - Broke OAuth for all Akamai-hosted websites, e.g. https://www.linkedin.com

HTTP virtual host confusion



Benign HTTP Redirectors on RP

- An search (open) redirector on Yahoo
  - url =http://search.yahoo.com/redirect/attacker.com
  - Requests to url?params are redirected to http://attacker.com/?params
- An shortened URL on Bitly (for attacker.com)
  - url = http://bitly.com/xyzwiu
  - Requests to url?params are redirected to http://attacker.com/?params
- A hosted website on Wordpress
  - url = http://attacker.wordpress.com
  - Requests to url?params can be redirected to http://attacker.com/?params

Triple Redirection Attacks on OAuth

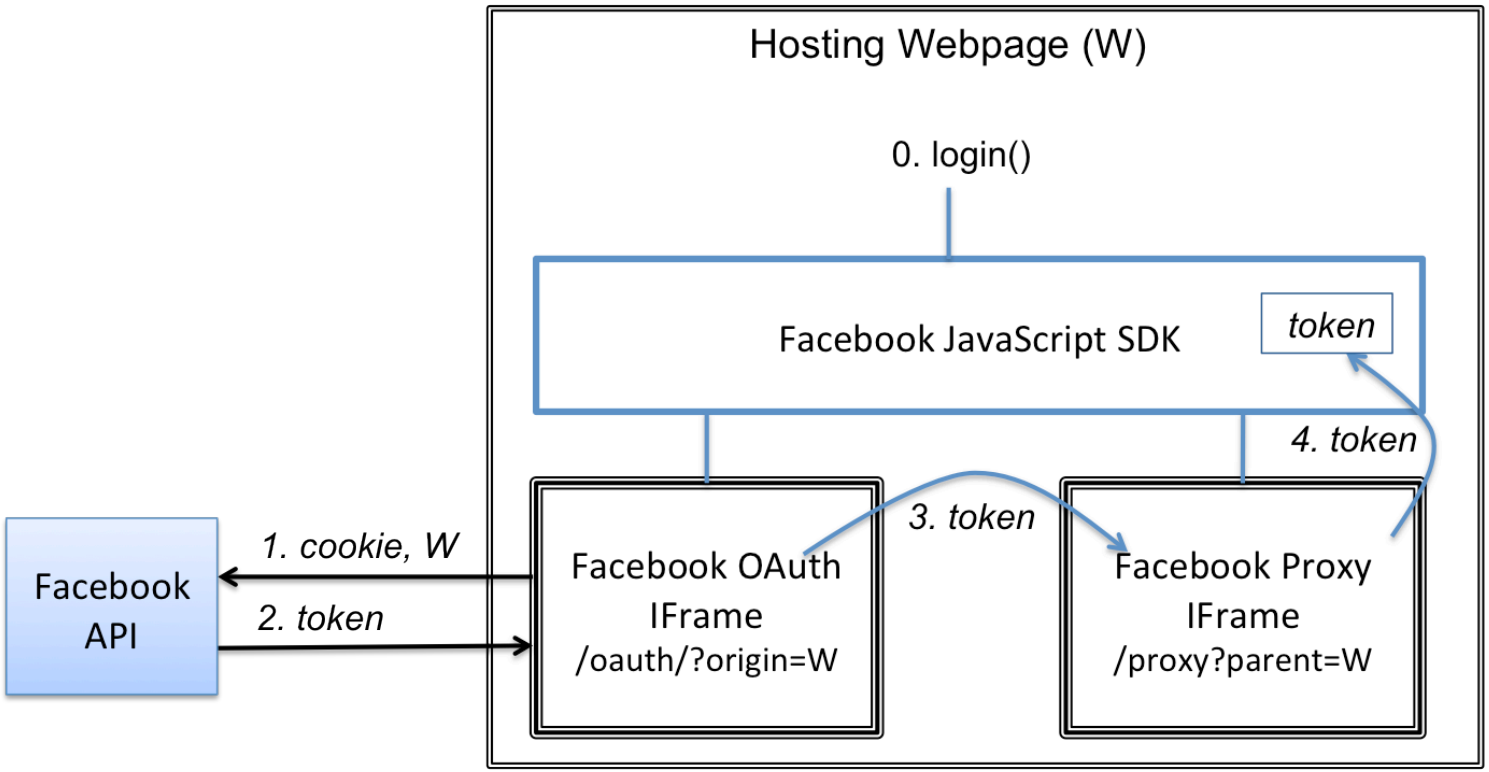
- Suppose a malicious website redirects a user to
  - https://facebook.com/oauth?
    app_id=(Yahoo) &
    perms=email,name,… &
    redirect_uri=search.yahoo.com/redirect/attacker.com
- Facebook will then redirect the browser to
  - http://search.yahoo.com/redirect/attacker.com/?
    access_token=XXX
- Yahoo Search will then redirect the browser to
  - http://attacker.com/?access_token=XXX

- Hence, the attacker obtains the token XXX

- Found by formal analysis in ProVerif [IEEE CSF 2012]

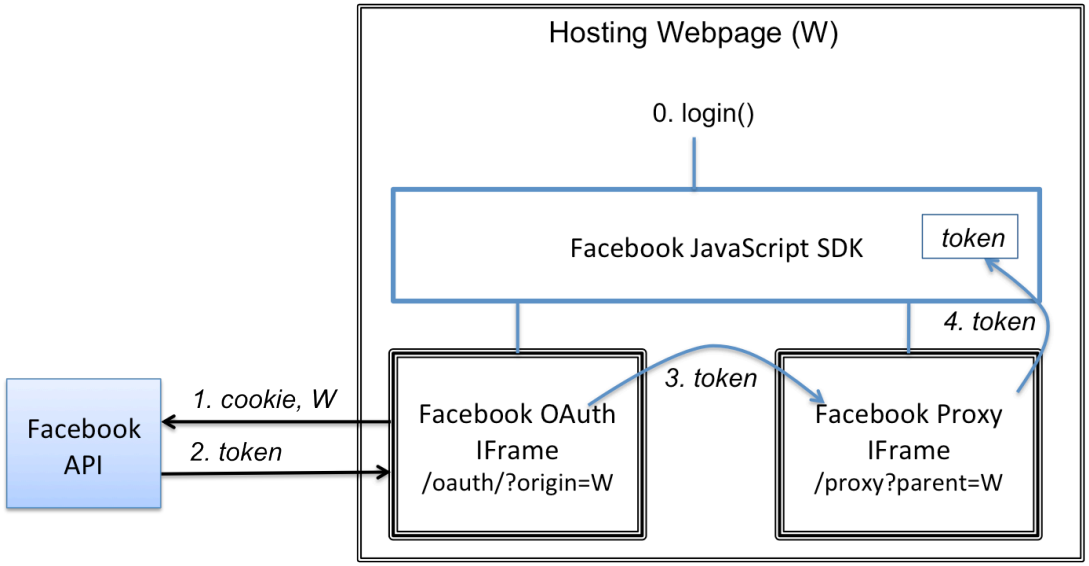
Delivering Tokens to JavaScript

- Facebook's JavaScript SDK implements OAuth 2.0
  - Load SDK, call FB.login(), call FB.api()
  - Creates two frames: OAuth gets token, Proxy delivers token
  - Google, Live SDKs do similar things



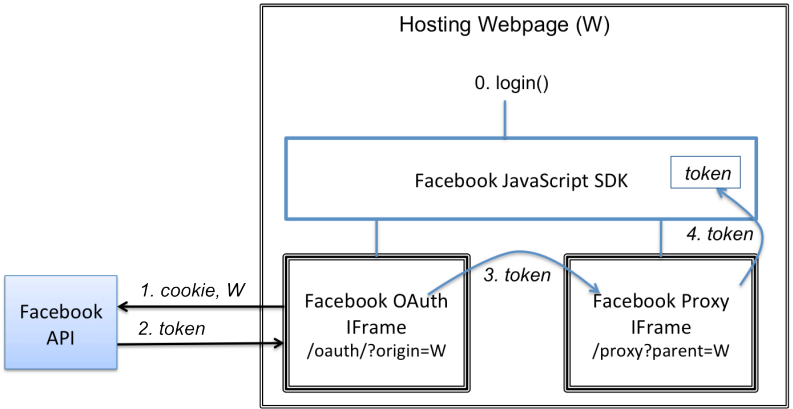
Relying on the Same Origin Policy

- Four separate instances of the Same Origin Policy
  - *iframe*: W cannot access content of OAuth or Proxy
  - *redirection*: OAuth token redirection is invisible to W
  - *AJAX*: W cannot directly access Facebook API
  - *postMessage*: W cannot read token sent to Yahoo



Origin Spoofing Attacks

- Malicious website W can steal access tokens for Yahoo
  - OAuth iframe started with URL parameter **origin=Yahoo**
  - Proxy iframe started with URL parameter **parent=W**
  - OAuth frame retrieves the token XXX for **Yahoo**, passes the token to Proxy, which sends it to **W**
- Attack possible due to many bugs [Usenix Security 2013]
  - Bug #1: OAuth and Proxy do not compare origin == parent
  - Bug #2: Proxy does not parse its parent URI correctly
  - Bug #3: OAuth does not parse multiple params in origin correctly



Many Attacks on OAuth Tokens

Active area of research

- *Discovering Concrete Attacks on Website Authorization by Formal Analysis*, C. Bansal, K. Bhargavan, S. Maffeis. CSF 2012.
- *Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services*, R. Wang, S. Chen, and X. Wang, *IEEE S&P 2012*
- *….. Many many papers in ACM CCS, IEEE S&P, Usenix Security*

Attacks appear at all levels

- TLS configuration, HTTP redirectors, SOP misuse
- Most of these threats are well documented in the standards, but developers still fall for them

Protecting Bearer Tokens over TLS

- Protocols like OAuth rely on secret bearer tokens
  - Tokens may be stolen and reused by attacker
  - Protecting tokens requires a concerted effort by web developers following best practices to design websites
  - Need to keep up with latest attacks on TLS, SOP, etc.

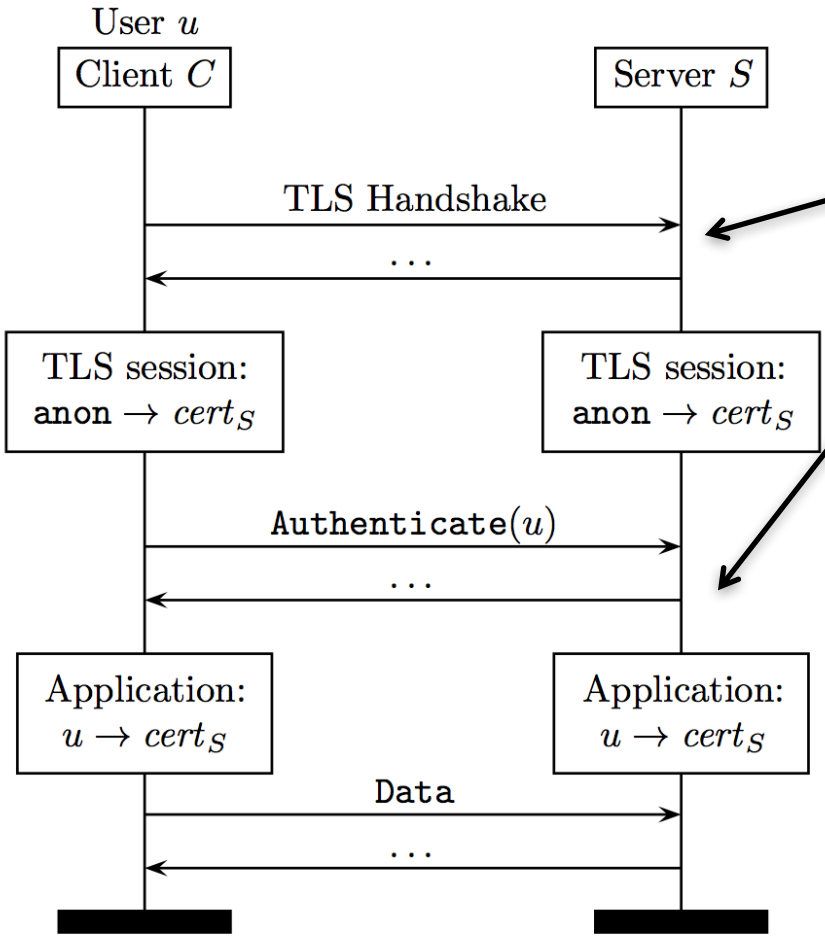
- Formal methods can help evaluate deployments
  - Needs a “comprehensive model of the web”
  - See Daniel, Ralf, and Guido’s work
  - Can we get protocol designers to use such tools?
  - Can we design protocols that are easier to analyze?

Making Tokens Less Valuable

- Protocols like OAuth rely on secret bearer tokens
  - Can we redesign them so that stolen bearer tokens cannot be reused?

- Compound authentication via Token Binding
  - Each authentication token is bound to a TLS channel
  - Even if stolen, it can only be used on that channel
  - Prevents a general class of credential forwarding attacks
  - Bit it requires additional guarantees from TLS

User authentication over TLS



- Common Pattern
  - *Outer*: server-authenticated TLS
  - *Inner*: user authentication protocol
- Many examples
  - SASL, GSSAPI, EAP, …
  - TLS Renegotiation with client certificate
- Inner authentication *blesses* outer unauthenticated channel

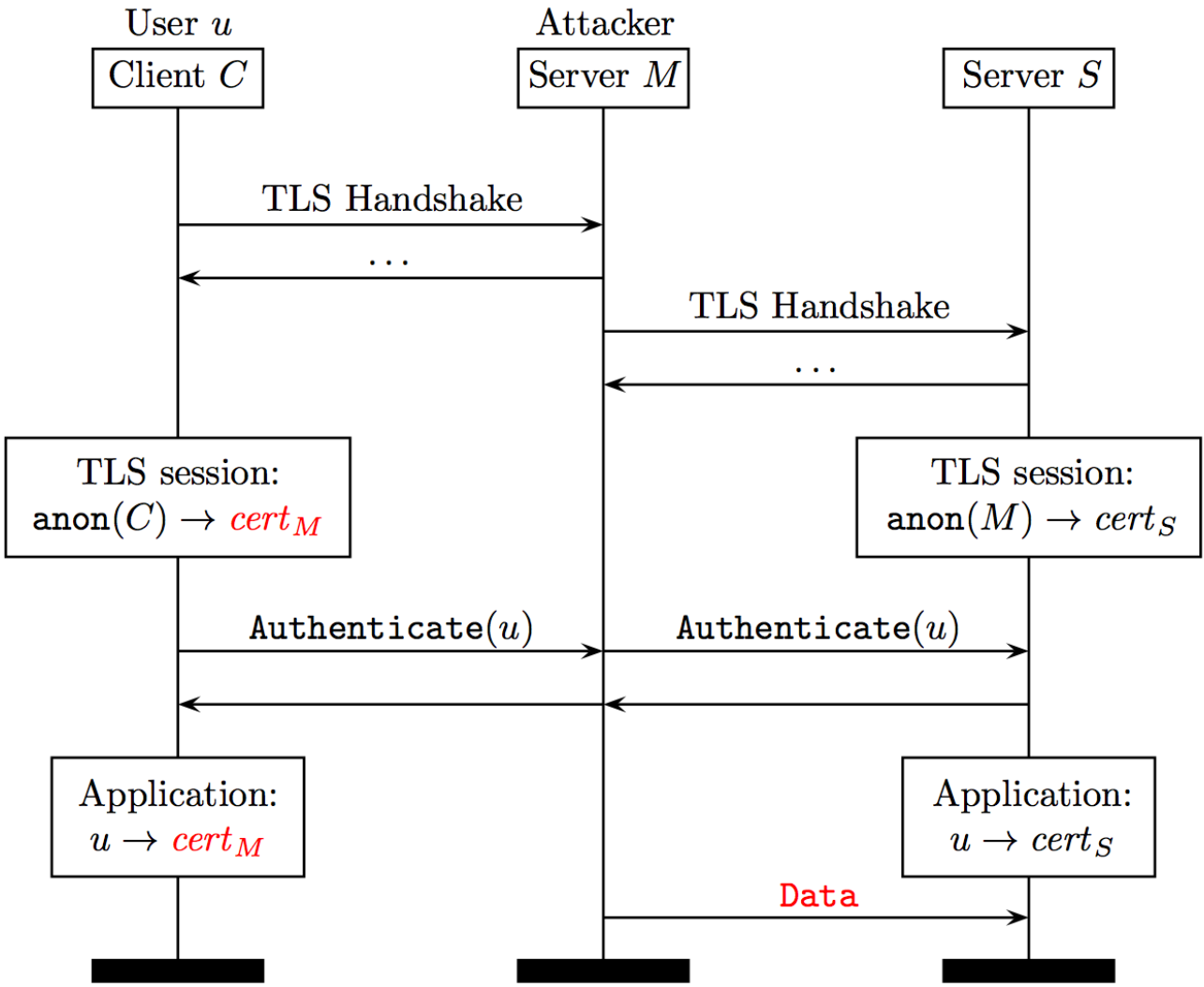
  Need to strongly bind the two protocol layers together!

Generic credential forwarding attack

Simplified version of [Asokan, Niemi, Nyberg'02]

- Suppose *u* uses same authentication credential at both *M* and *S*
- *M* forwards *S's* authentication challenge to *C*
- *M* forwards *C's* response to *S*

- *M can* log in as *u* at *S*!

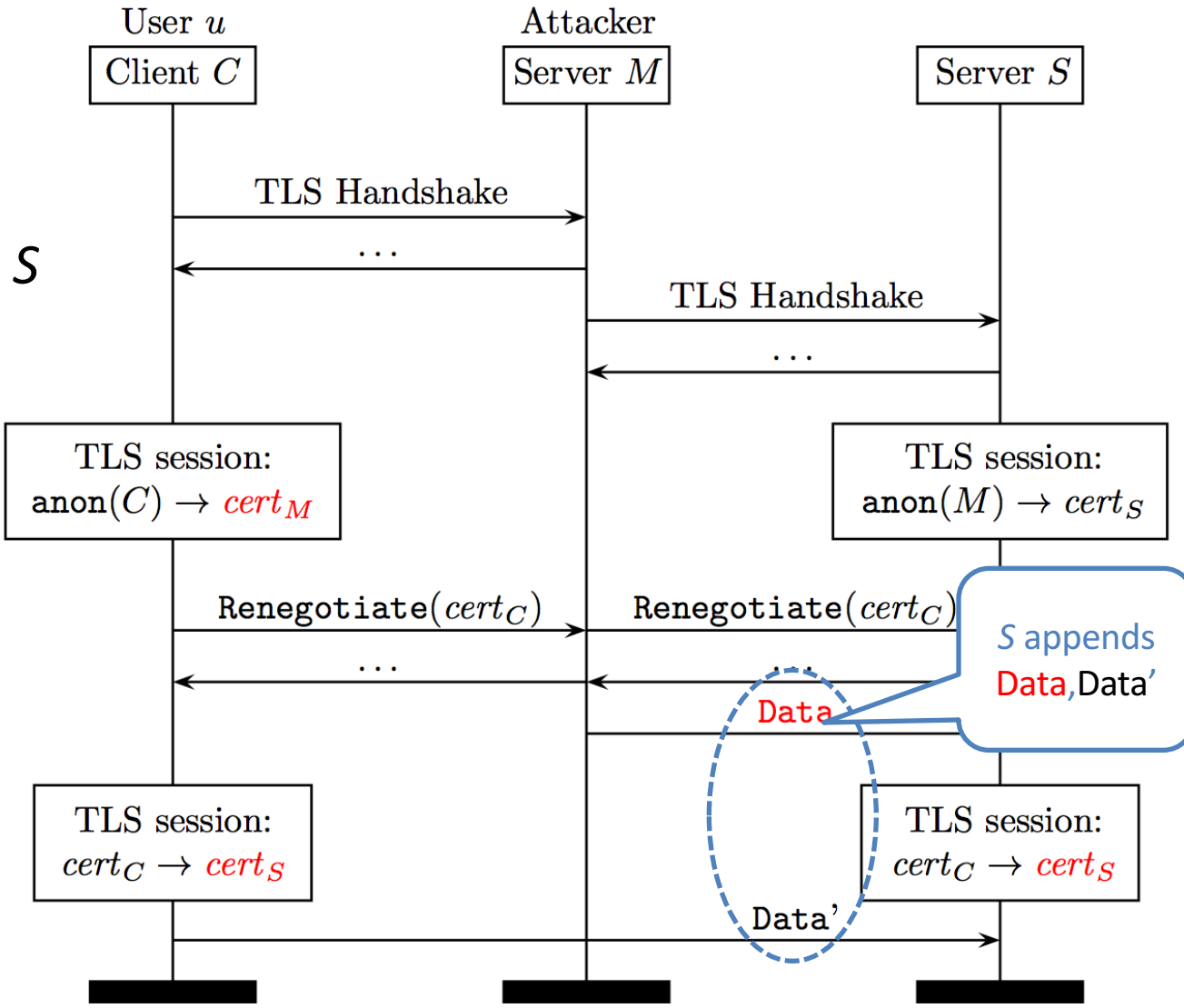


TLS renegotiation attack [2009]

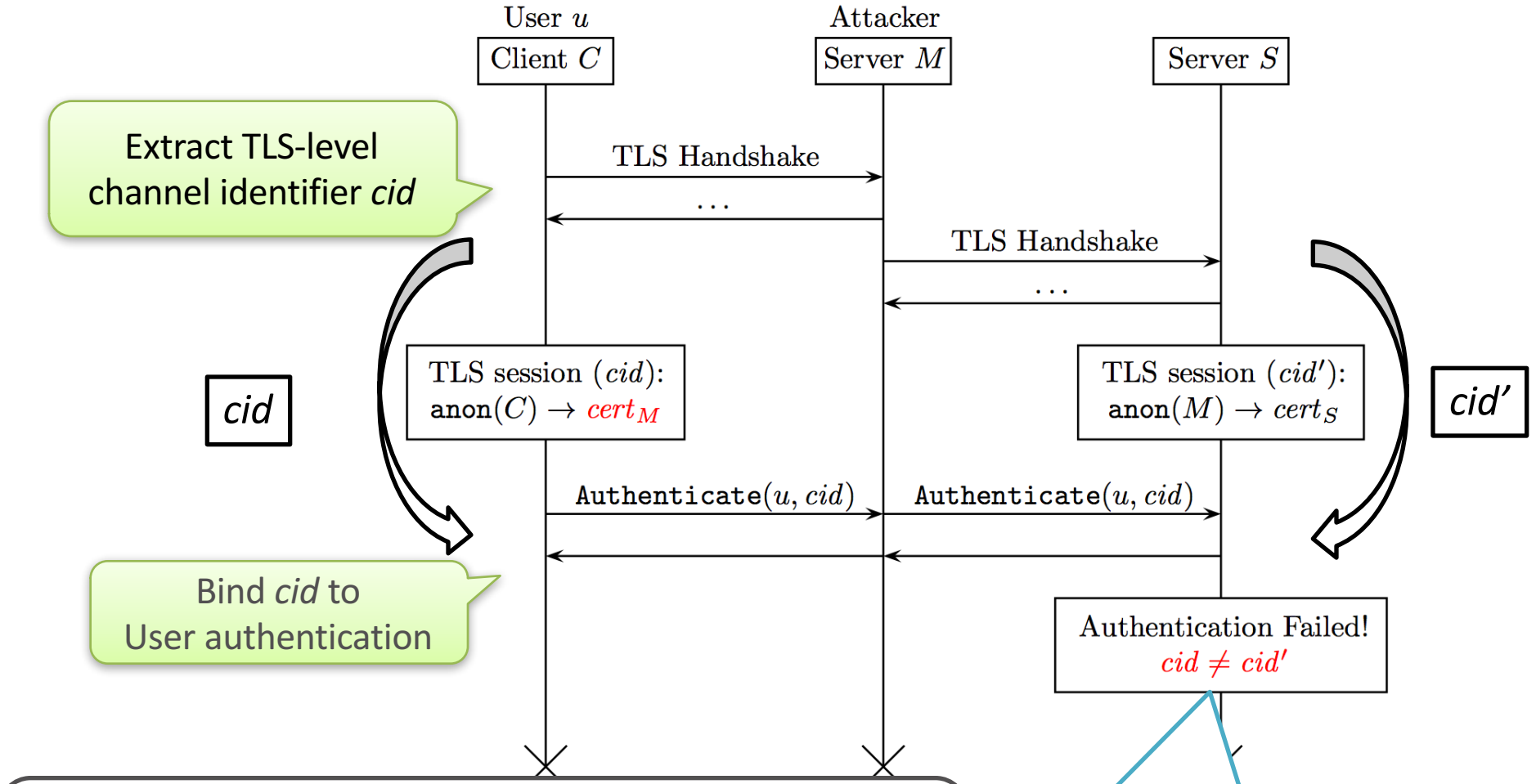
Martin Rex's Version

- Suppose *u* uses same client cert to log in to both *M* and *S*
- *M* forwards *S's* renegotiation request to *C*
- *M* forwards renego handshake between *C and S*

- *S concatenates data sent by M to data sent by u*!



Binding user auth to TLS channels

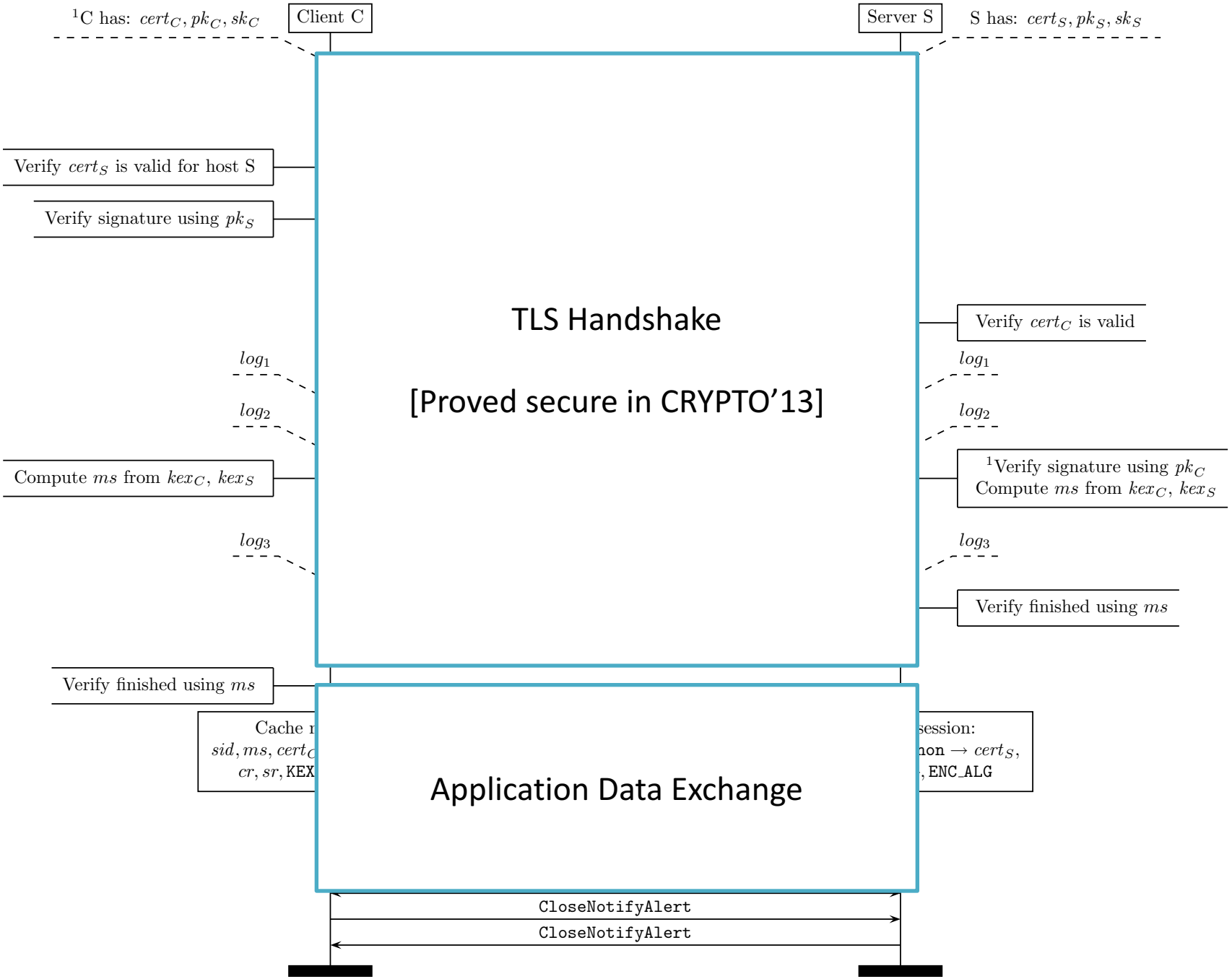


Computing a channel identifier (*cid*):

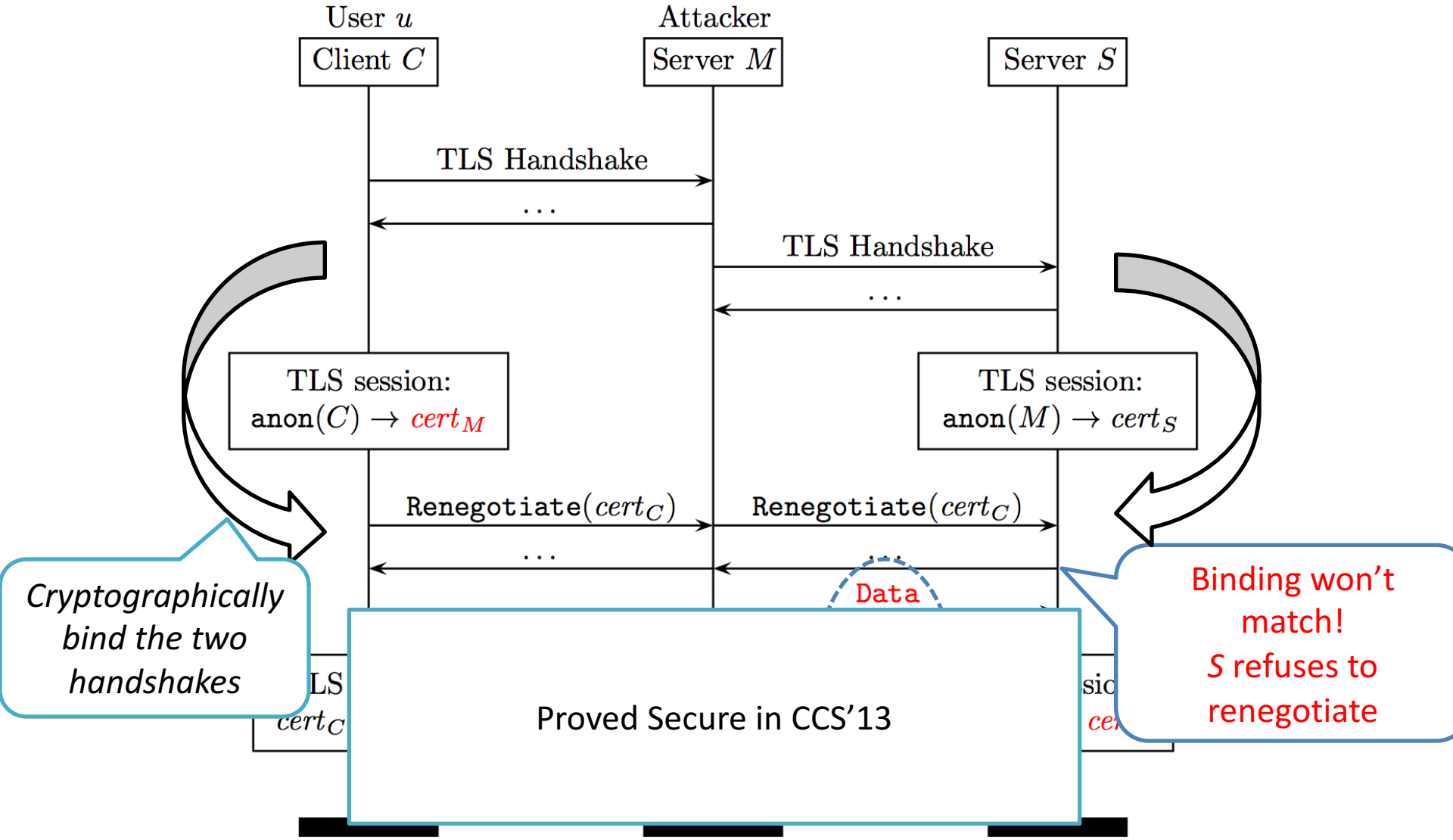
- *f(master secret)* (EAP)
- *f(handshake log)* (Renegotiation Indication, SASL)

S detects MitM attack because cid ≠ cid'

Is it secure?


[1]C has: $cert_C, pk_C, sk_C$
Client C
Server S
S has: $cert_S, pk_S, sk_S$
Verify $cert_S$ is valid for host S
Verify signature using pk_S
TLS Handshake
[Proved secure in CRYPTO'13]
Verify $cert_C$ is valid
log_1
log_2
Compute ms from kex_C, kex_S
[1]Verify signature using pk_C
Compute ms from kex_C, kex_S
log_3
Verify finished using ms
Verify finished using ms
Application Data Exchange
CloseNotifyAlert
CloseNotifyAlert

TLS Renegotiation Fix [2009]



Triple Handshake Attack [IEEE S&P 2014]

- None of the existing channel identifiers quite work
  - Tokens bound to master_secret
    are broken after one handshake (PEAP)
  - Tokens bound to verify_data
    are broken after two handshakes (SASL)
  - TLS Renegotiation is broken after three handshakes

- Why is this a concern for Token Binding?
  - A channel-bound token can still be reused by attacker!
  - TLS does not provide a good channel identifier,
    but it can and it should!

Fixing the TLS Standard

Compute a session hash for every full handshake

session_hash = Hash(handshake log)

Add session hash to master secret derivation

master_secret = PRF(pre_master_secret,
"extended master secret”,
session_hash) [0..47];

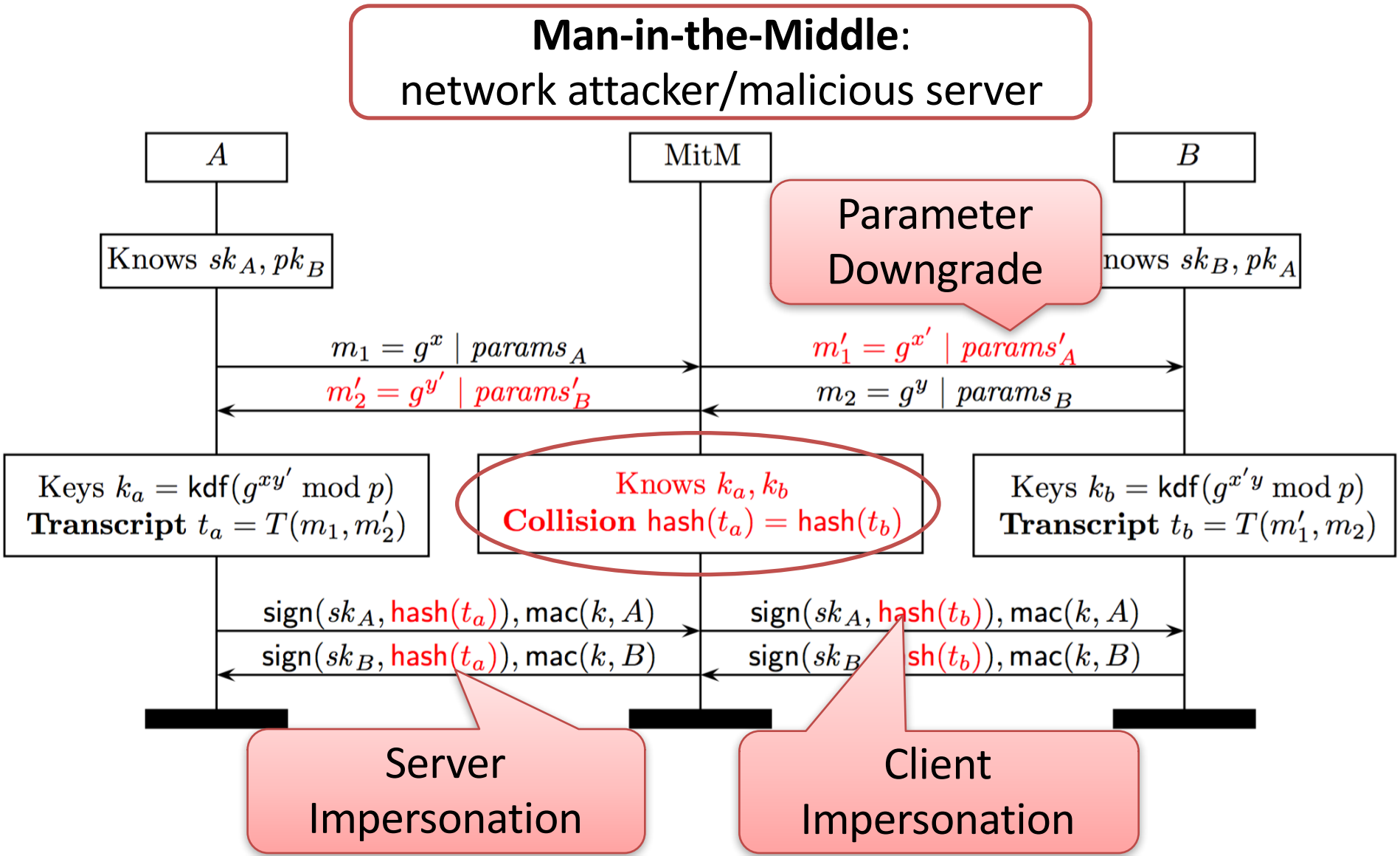
New protocol extension: RFC7627

- Implemented in miTLS, OpenSSL, NSS, PolarSSL, …
- Saves master_secret, tls-unique, renegotiation from attacks
- Construction built in to TLS 1.3

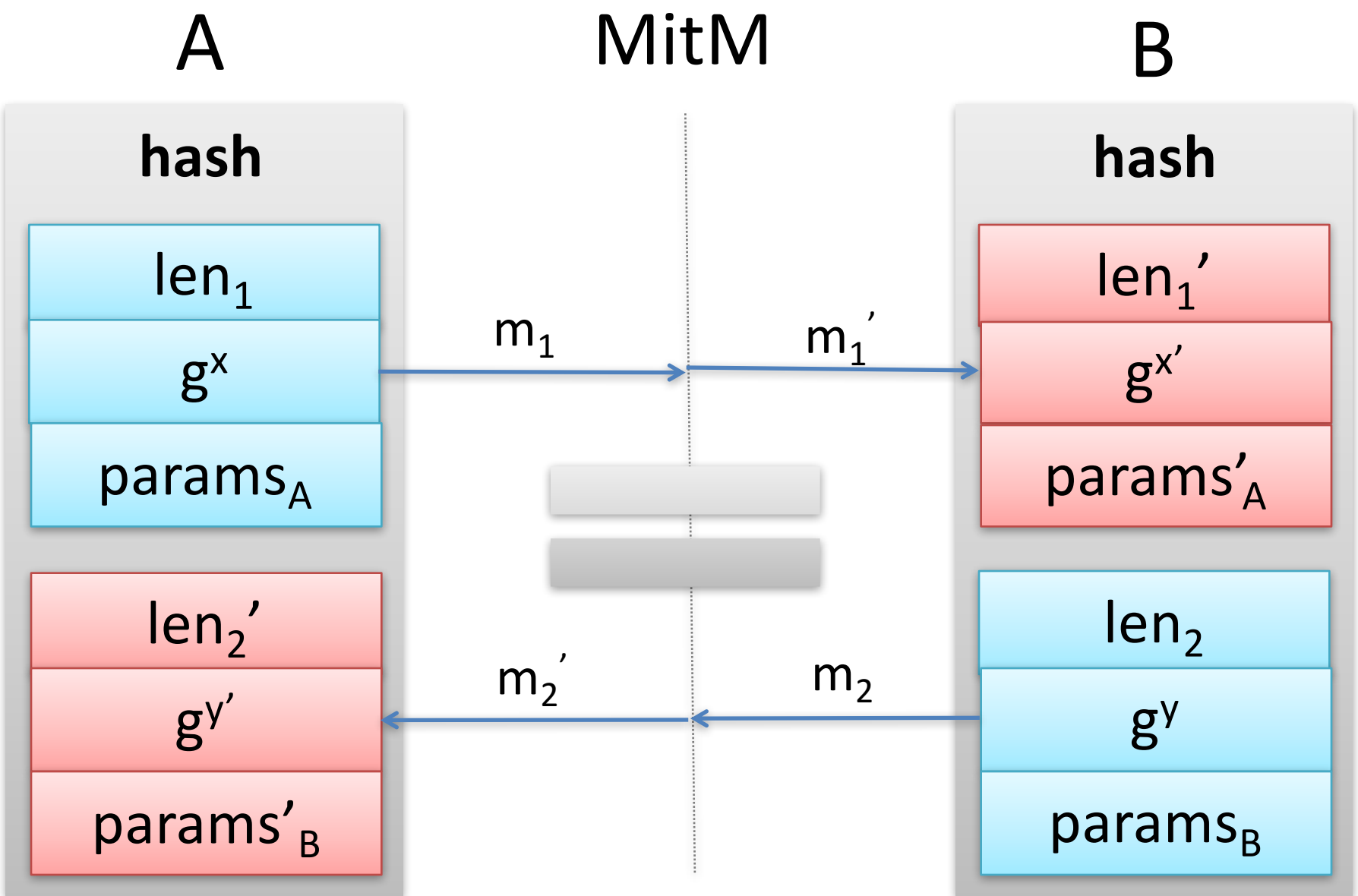
Problem Solved?

- Let's bind our tokens to "tls-unique"
  - Used in Token Binding, FIDO, SASL
    tls-unique = first 12 bytes of:
    HMAC-SHA256(master_secret,
    handshake transcript);
  - No two TLS connections should have the same tls-unique value
    - Relies on RFC7627 to prevent Triple Handshake
    - What if attacker can cause collisions in tls-unique?
    - 12 bytes = 96 bits is too short

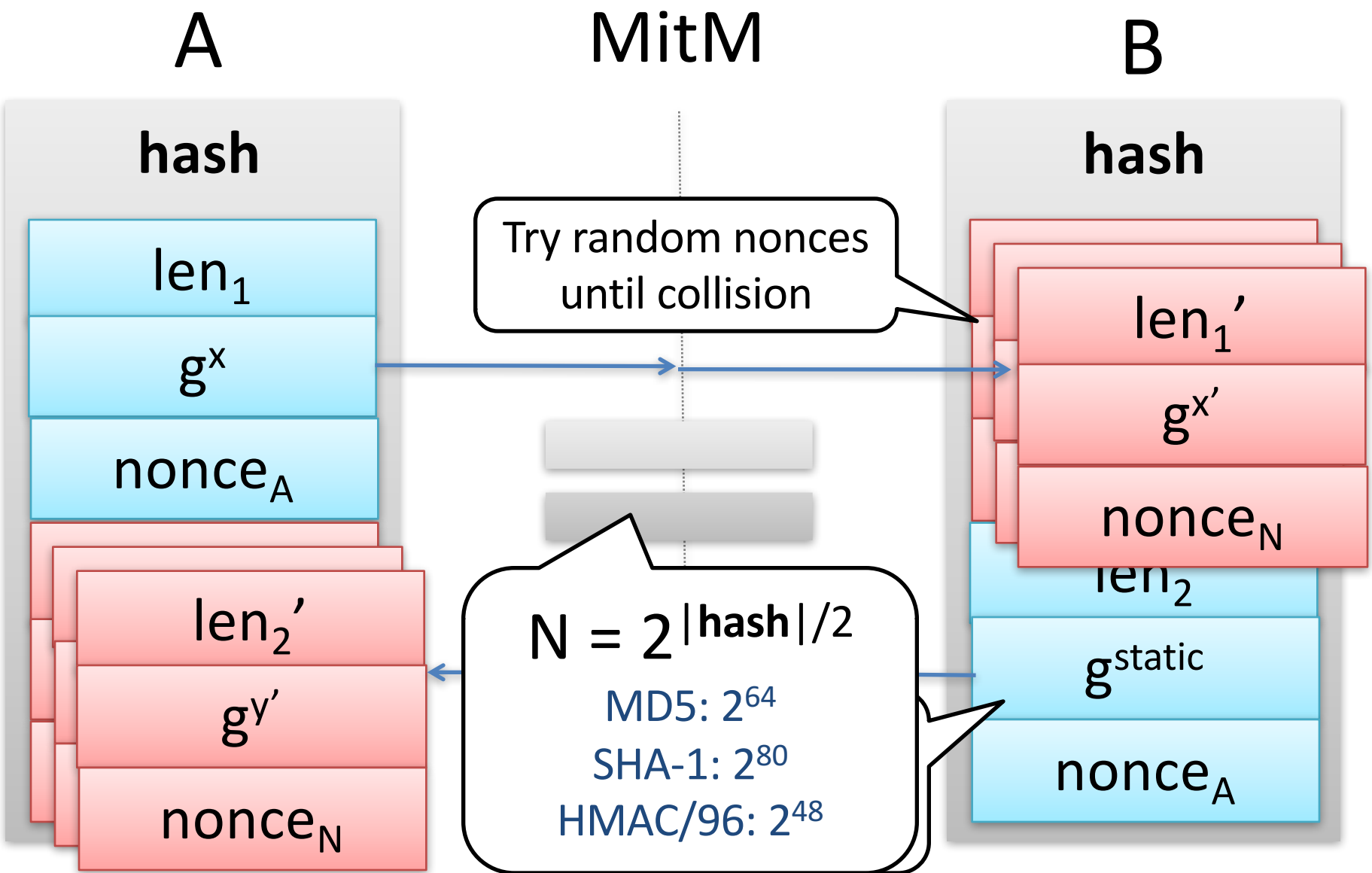
SLOTH: Transcript Collision Attacks



Computing Transcript Collisions



Generic Transcript Collisions



Preventing Transcript Collisions

- In TLS 1.2, tls-unique not a good identifier
  - Requires 2^{48} HMAC computations to break
  - Full SHA256 hash would be a better identifier
  - tls-unique now deprecated from use.

- In TLS 1.3, use exporter master secret
  - ems = HKDF(master_secret, handshake log)
  - Prevents SLOTH, triple handshake
  - Good channel identifier for token binding

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

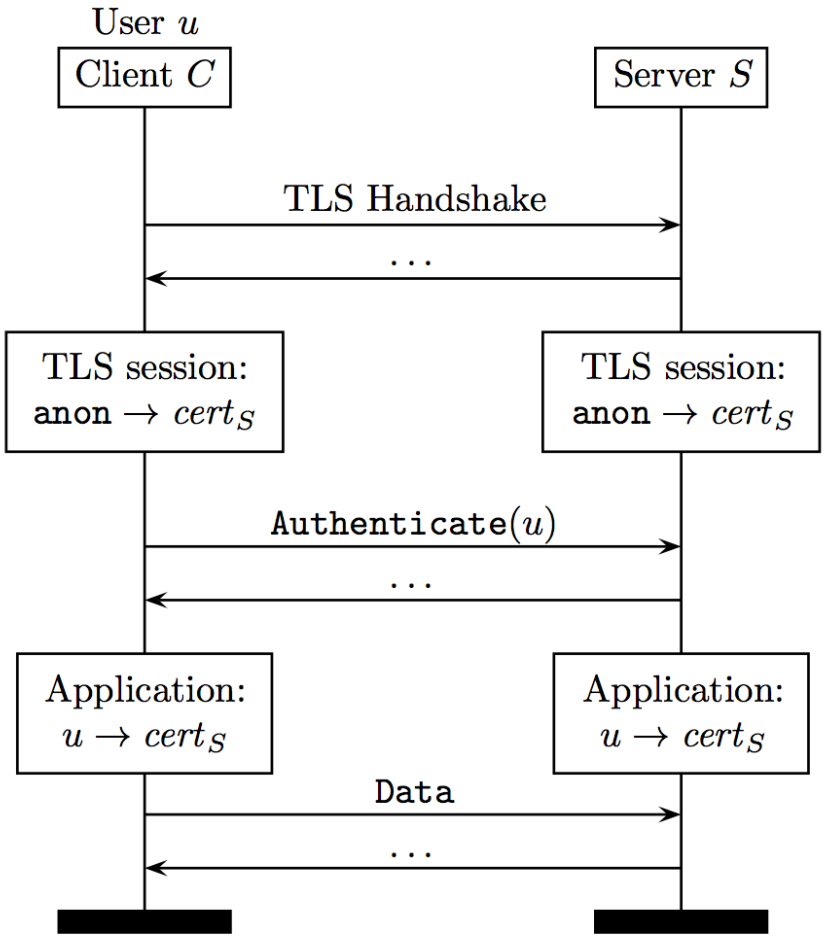
Faster: lower latency with 1 round-trip

- 0-round trip mode also available

Crypto proofs built side-by-side with standardization

- Active participation by a large group of researchers
- Proofs in multiple symbolic and computational models
- Attacks found and fixed before standardization

TLS 1.3 + Token Binding + OAuth



- Browser holds a private key sk_{RP} for each RP
- It extracts ems from TLS 1.3
- It signs ems with sk_{RP} to authenticate TLS channel
- pk_{RP} identifies user's browser

- IP binds token to pk_{RP}
- RP verifies that token was received on a TLS channel authenticated with sk_{RP}

- Stolen tokens cannot be used by other clients, without sk_{RP}

Final Thoughts

- OAuth security relies on many other mechanisms
  - TLS, HTTP, Same Origin Policy, etc.
  - Careful: they may not guarantee what you think they do
- Formal methods provide a framework for systematically evaluating security protocols against a class of attacks
  - Too many things can go wrong
  - So prove it secure instead of waiting for attacks
  - Need a comprehensive model of the web + automated tools
- Protocol design can make analysis simpler
  - TLS 1.3 designed in collaboration with academics
  - The process was hard but fruitful: new attacks and proofs
  - Open problems: TLS 1.3, Token Binding, OpenID Connect

Questions?

- http://prosecco.inria.fr

- *Discovering Concrete Attacks on Website Authorization by Formal Analysis*
  (C Bansal, K Bhargavan, S Maffeis)
  In IEEE Computer Security Foundations Symposium, 2012.
- *Language-Based Defenses Against Untrusted Browser Origins*
  (K Bhargavan, A Delignat-Lavaud, S Maffeis)
  In USENIX Security Symposium, 2013
- *Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS*
  (K Bhargavan, A Delignat-Lavaud, C Fournet, A Pironti, P-Y Strub)
  In IEEE Symposium on Security & Privacy, 2014
- *Verified Contributive Channel Bindings for Compound Authentication*
  (K Bhargavan, A Delignat-Lavaud, A Pironti)
  In Network and Distributed System Security Symposium, 2015
- *Network-based Origin Confusion Attacks against HTTPS Virtual Hosting*
  (A Delignat-Lavaud, K Bhargavan)
  In International Conference on World Wide Web, 2015