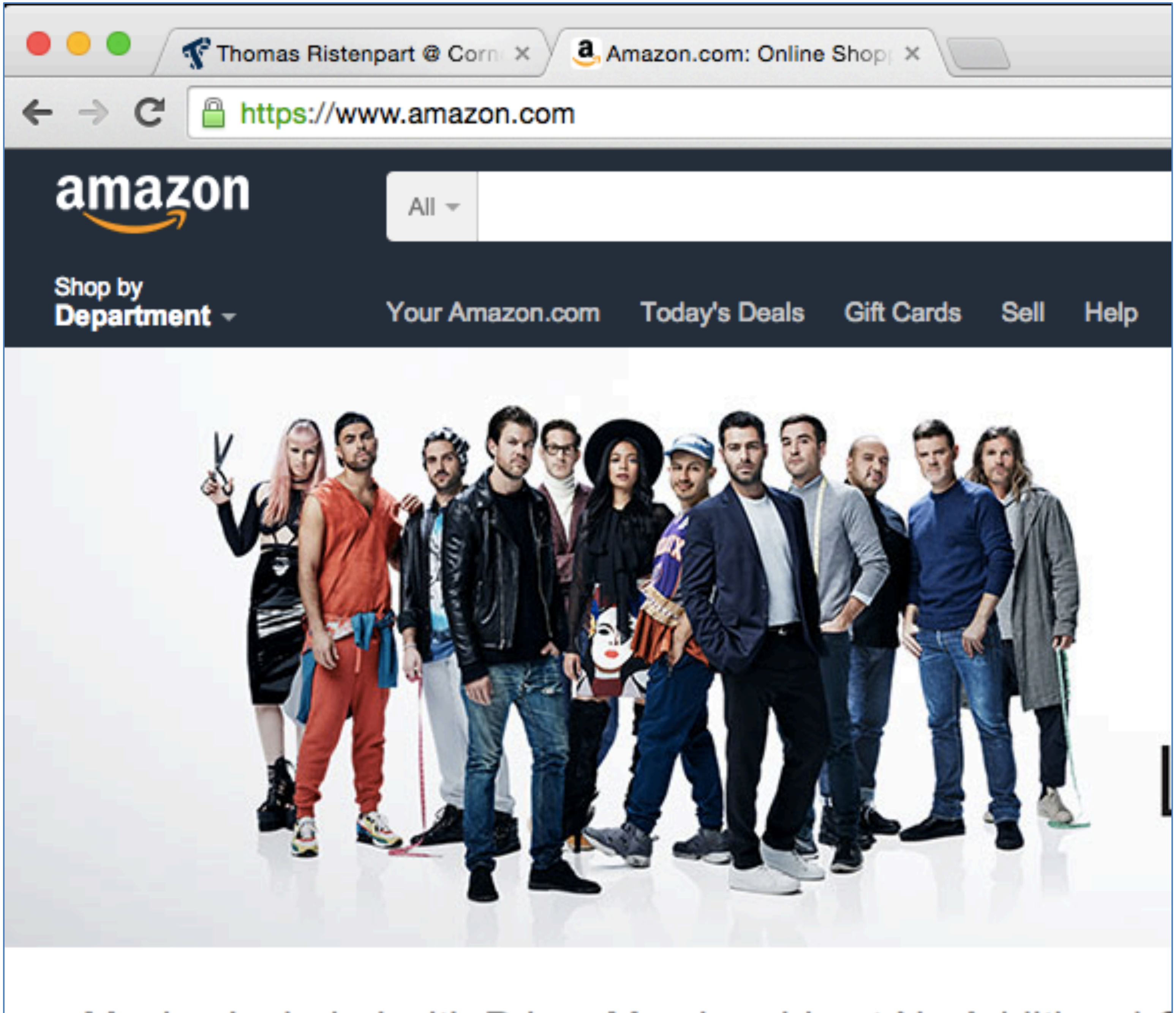
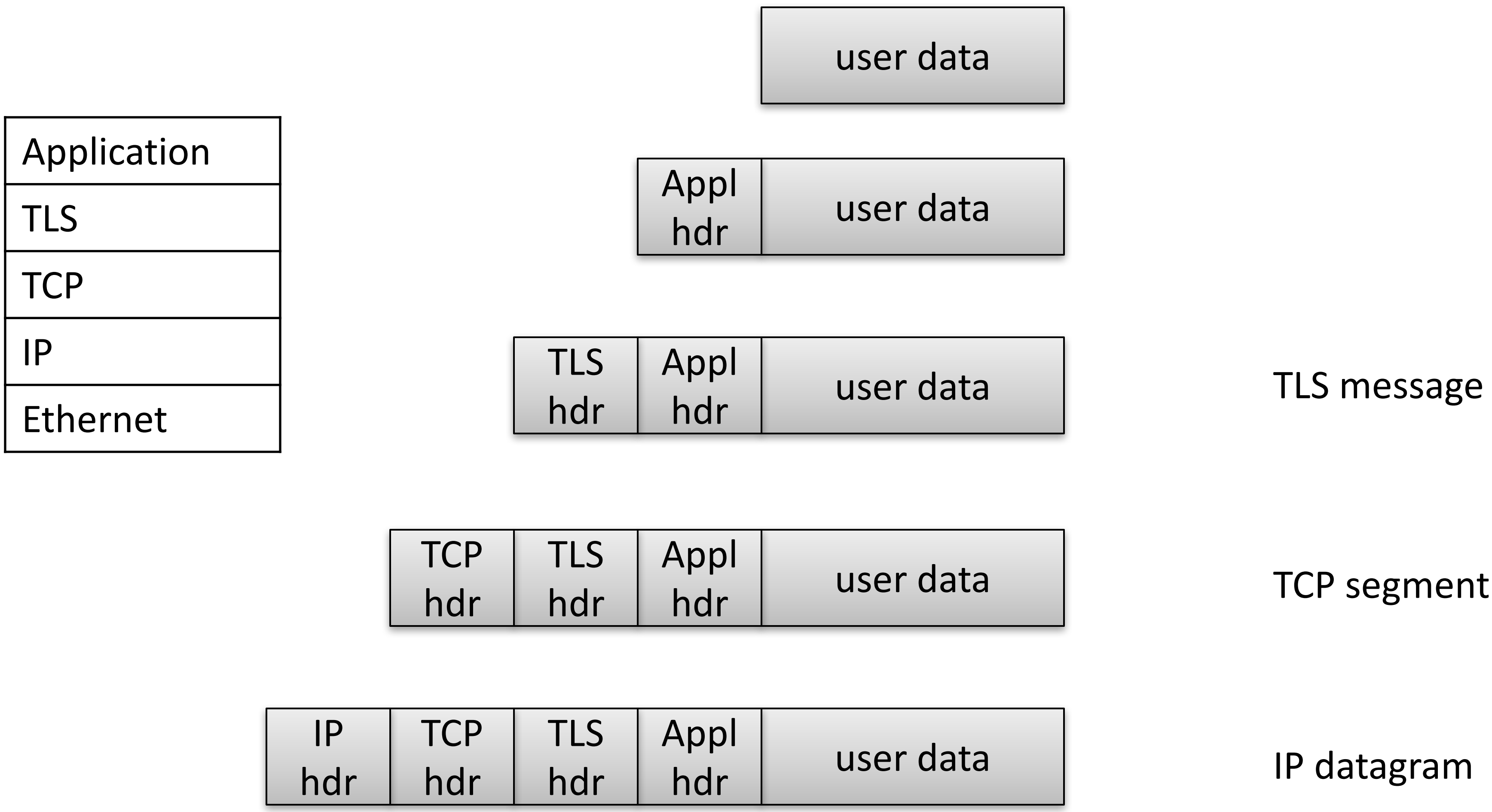
Today in Cryptography (5830)

TLS Overview refresh
TLS hand shakes
TLS record layer & attacks


Thomas Ristenpart @ Corn
Amazon.com: Online Shop
https://www.amazon.com
amazon
All
Shop by
Department
Your Amazon.com
Today's Deals
Gift Cards
Sell
Help

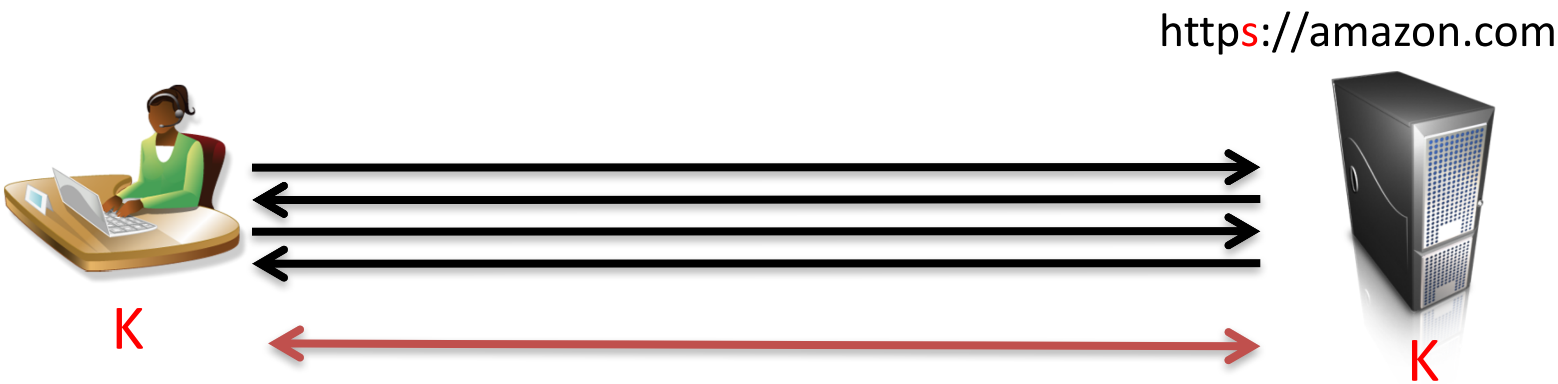
TLS sits between application and TCP



Places TLS is used

- HTTPS
  - HTTP messages but over TLS, not TCP
- Email connections
  - When getting information from your email server (not the email contents themselves)
- Virtual private networks (VPNs)
  - Tunnel other internet connections over a TLS connection

How does TLS work (high level)?



Step 1:
Key exchange protocol to share secret K

Step 2:
Send data via secure channel

The secure channel is implemented via our now familiar symmetric encryption primitives

Goals of handshake:

- Negotiate version
- Negotiate parameters (crypto to use)
- Authenticate server (Is server actually Amazon.com?)
    - Digital signatures and certificates
- Establish shared secret
    - Asymmetric encryption primitives

Client

TLS handshake for RSA transport

Server

Pick random Nc

ClientHello, MaxVer, Nc, Ciphers/CompMethods →

Pick random Ns

← ServerHello, Ver, Ns, SessionID, Cipher/CompMethod

Check CERT using CA public verification key

← CERT = (pk of bank, signature over it)

Pick random PMS
C <- E(pk,PMS)

C →

PMS <- D(sk,C)

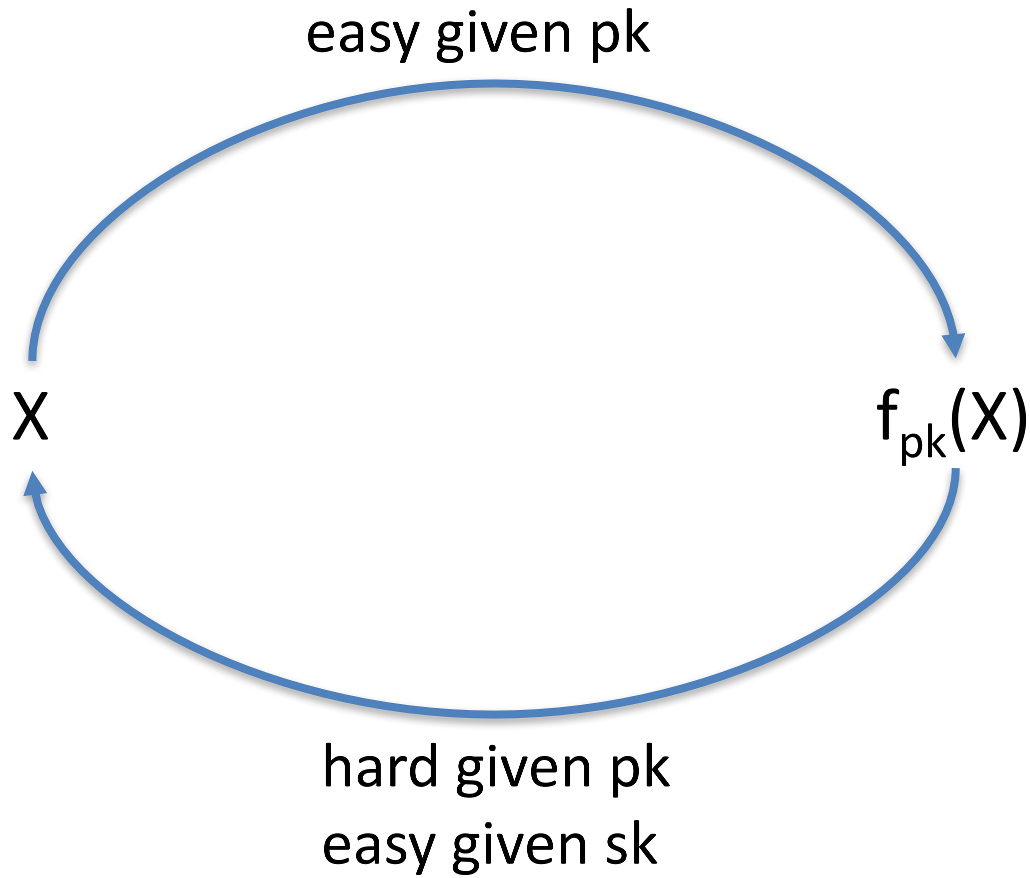
ChangeCipherSpec,
{ Finished, PRF(MS, “Client finished” || H(transcript)) } →

ChangeCipherSpec,
{ Finished, PRF(MS, “Server finished” || H(transcript’)) } ←

Bracket notation means contents encrypted

MS <- PRF(PMS, “master secret” || Nc || Ns)

Trapdoor functions



Client

TLS handshake for Diffie-Hellman Key Exchange

Server

Pick random Nc

ClientHello, MaxVer, Nc, Ciphers/CompMethods →

Pick random Ns

← ServerHello, Ver, Ns, SessionID, Cipher/CompMethod

Pick random x

$X = g^x$

← CERT = (pk_s , signature over it)

← p , g , X , σ = Sign(sk_s, p || g || X)

Check CERT using CA public verification key

Check σ

Pick random y

$Y = g^y$

PMS = g^{xy}

Y →

PMS = g^{xy}

ChangeCipherSpec,
{ Finished, PRF(MS, “Client finished” || H(transcript)) } →

ChangeCipherSpec,
← { Finished, PRF(MS, “Server finished” || H(transcript’)) }

Bracket notation means contents encrypted

MS <- PRF(PMS, “master secret” || Nc || Ns)

One-way functions

$$
\begin{tikzcd}
x \arrow[r, bend left, "\text{easy given } x"] & X = g^x \arrow[l, bend left, "\text{hard given } X"]
\end{tikzcd}
$$

TLS Key derivation & use

MS <- PRF(PMS, “master secret” || Nc || Ns)

K1,K2 <- PRF(MS, “key expansion” || Ns || Nc)

PRF(secret, seed) = HMAC-HASH(secret, A(1) + seed) +
HMAC-HASH(secret, A(2) + seed) +
HMAC-HASH(secret, A(3) + seed) + ...

Where A(0) = seed and A(i) = HMAC_hash(secret, A(i-1))

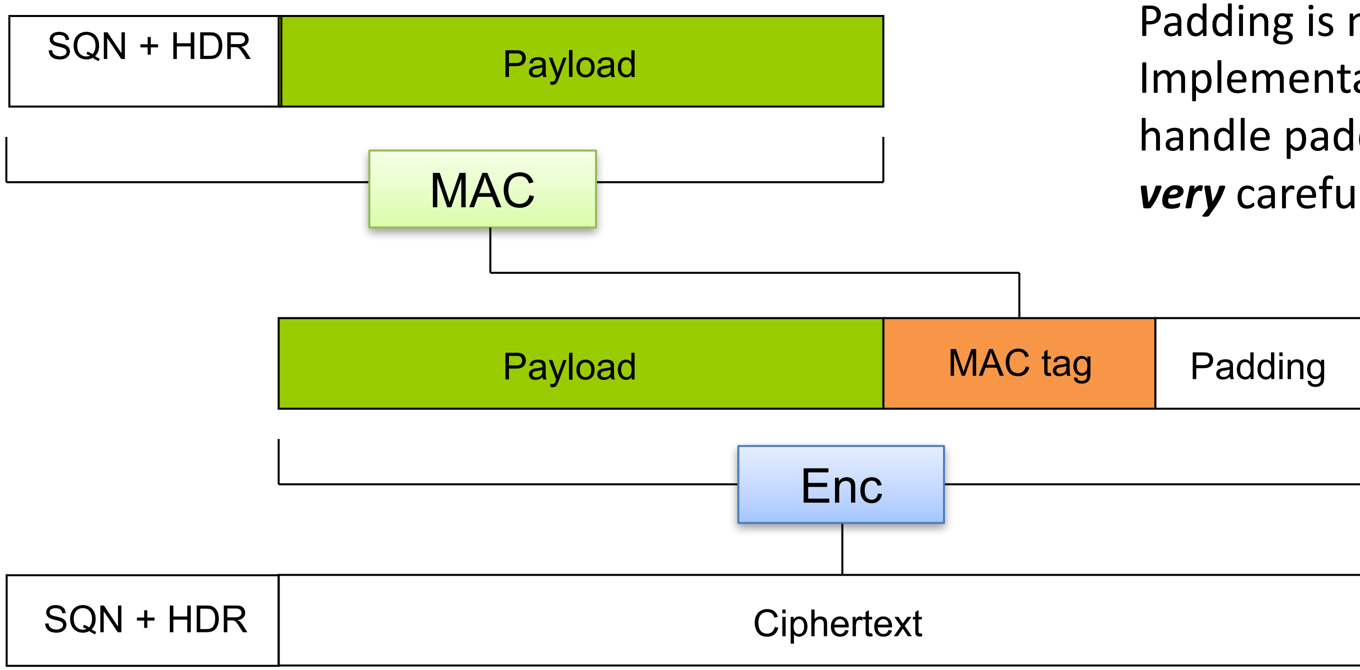
This mess replaced with HKDF in 1.3

C1 <- AEnc(K1,Message) $\xrightarrow{\text{C1}}$ Message <- ADec(K1,C1)

C2 <- AEnc(K2,Message’)

Message’ <- ADec(K2,C2) $\xleftarrow{\text{C2}}$

TLS 1.2 record protocol: MAC-Encode-Encrypt (MEE)



Padding is not MAC'd. Implementations must handle padding checks ***very*** carefully.

MAC HMAC-MD5, HMAC-SHA1, HMAC-SHA256

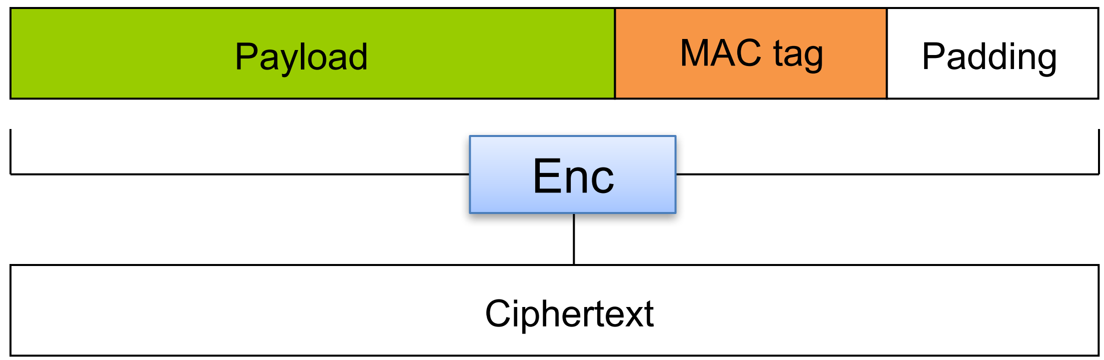
Encrypt CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

Record layer details

- Fragmentation
  - Maximum TLS ciphertext handles 2^{14} bytes of message data
  - Split longer message into multiple fragments
  - Encrypt each fragment separately
- Sequence numbers keep track of count of fragments sent in each direction
- Compression methods

Padding oracle vulnerabilities in TLS

CBC mode padding: 00 or 01 01 or 02 02 02 …



TLS 1.0: must check padding, and return ***decryption_failed*** error if it is incorrect
HMAC tag computation failure returns ***bad_record_mac*** error

TLS 1.1/1.2: must check padding, and return ***bad_record_mac*** error if it is incorrect
HMAC tag computation failure returns ***bad_record_mac*** error

https://www.imperialviolet.org/2013/02/04/luckythirteen.html

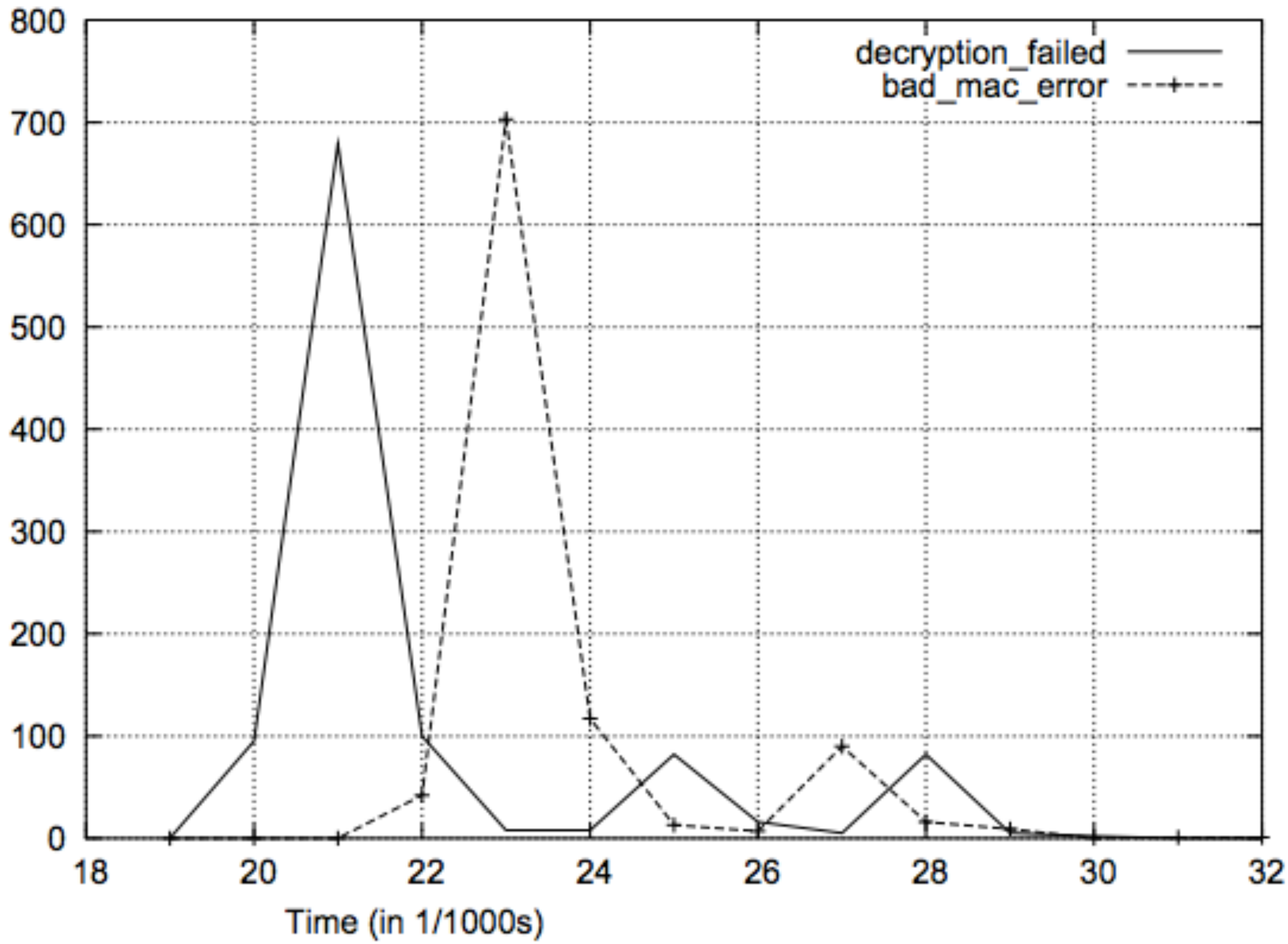
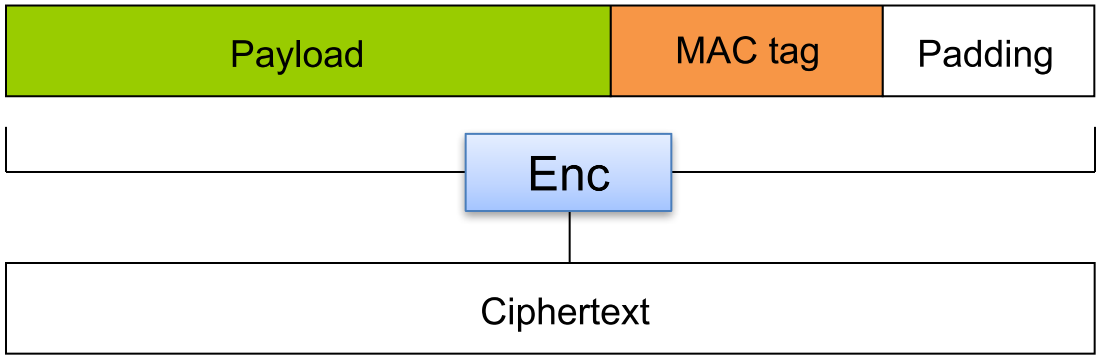


Fig. 3. Distribution of the number of `decryption_failed` and `bad_mac_error` error messages with respect to time.

From Canval et al. paper 2003

Padding oracle vulnerabilities in TLS

CBC mode padding: 00 or 01 01 or 02 02 02 …



TLS 1.0: must check padding, and return ***decryption_failed*** error if it is incorrect
HMAC tag computation failure returns ***bad_record_mac*** error

TLS 1.1/1.2: must check padding, and return ***bad_record_mac*** error if it is incorrect
HMAC tag computation failure returns ***bad_record_mac*** error

"implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct."

https://www.imperialviolet.org/2013/02/04/luckythirteen.html

From RFC 5246 (TLS 1.2)

In order to defend against this attack, implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct. In general, the best way to do this is to compute the MAC even if the padding is incorrect, and only then reject the packet. For instance, if the pad appears to be incorrect, the implementation might assume a zero-length pad and then compute the MAC. This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.

Lucky13 attack

Exploits timing differences in HMAC computations of different lengths to build padding oracle.
Assume SHA-1 (64 byte block, 20 byte output), AES (16 byte block)

IV and two other junk blocks

Offset used for search

$$HDR \mid\mid C_0 \mid\mid C_1 \mid\mid C_2 \mid\mid C' \oplus \Delta \mid\mid C^*$$

|HDR| = 13 bytes

Block preceding C*

Ciphertext block whose ptxt we want to learn

Three cases during decryption:

1. P_4 ends in 00 byte
2. P_4 ends in $\geq$2 valid padding bytes
3. P_4 ends in any other byte pattern

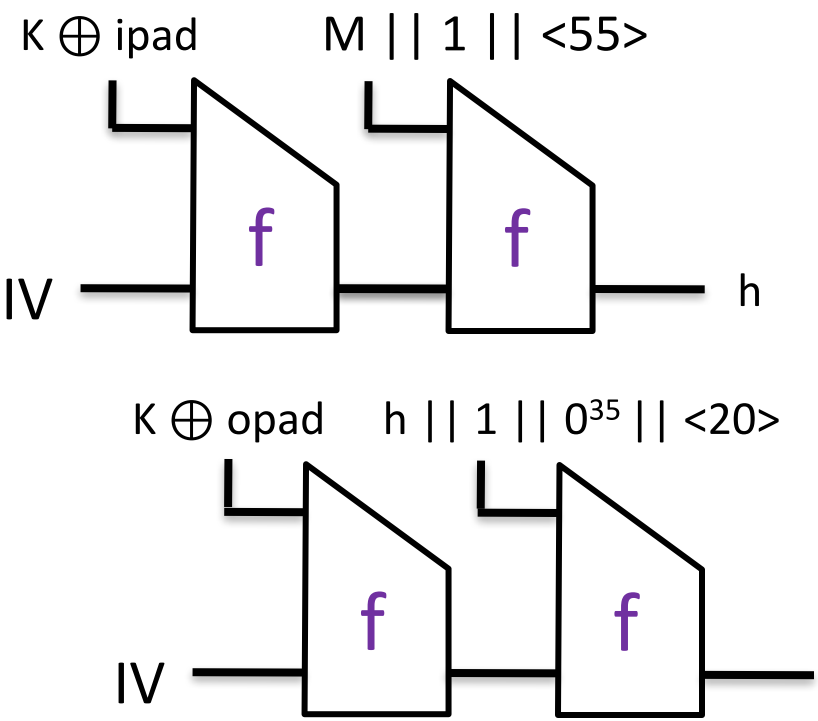
HMAC on 56 bytes
HMAC on 55 bytes
HMAC on 57 bytes

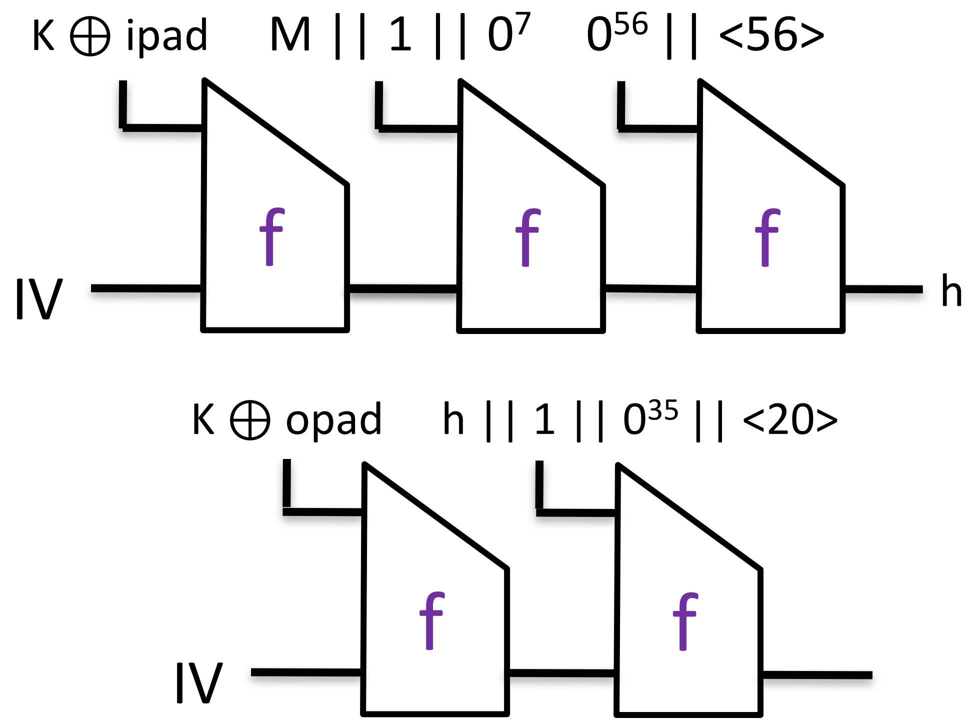
Slightly faster!

HMAC on 55 vs 56

55 byte message M
- Must be padded with 8-byte length field & at least 1 byte of padding
- Fits into single 64-byte block

56 byte message M
- Must be padded with 8-byte length field & at least 1 byte of padding
- Does ***not*** fit into single 64-byte block





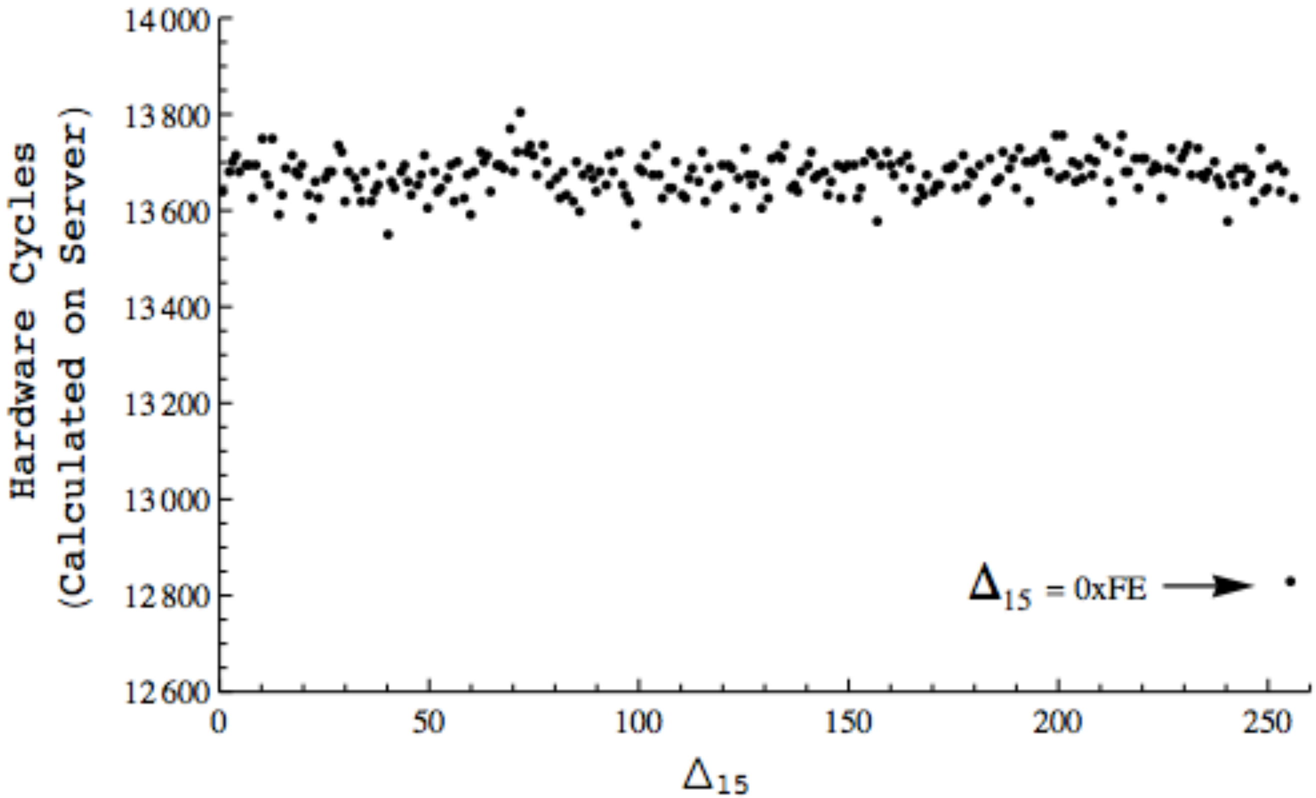


Figure 3: OpenSSL TLS median server timings (in hardware cycles) when $P^*_{14} = 0x01$ and $P^*_{15} = 0xFF$. As expected, $\Delta_{15} = 0xFE$ leads to faster processing time.

From [AlFardan and Paterson 2013]

TLS record layer attacks

Attack	Year	Vulnerability	Countermeasure
Vaudenay	2002	Padding oracle (theoretical)	---
Rogaway	2002	IV chaining (theoretical)	---
Kelsey	2002	Compression before encryption (theoretical)	---
Canval et al.	2003	Padding oracle via timing	Always compute HMAC
BEAST (Duong & Rizzo)	2011	IV chaining	Dedicated IVs
CRIME (Duong & Rizzo)	2012	TLS compression before encryption	Turn off TLS compression
Lucky13 (AlFardan & Paterson)	2013	Padding oracle via HMAC timing	Constant-time decryption attempted; move to RC4
RC4 attack (AlFardan et al.)	2013	RC4 cryptanalysis made practical	Move to CBC-based cipher suites
BREACH (Prado et al.)	2013	HTTP compression before encryption	Turn off HTTP compression (if possible)

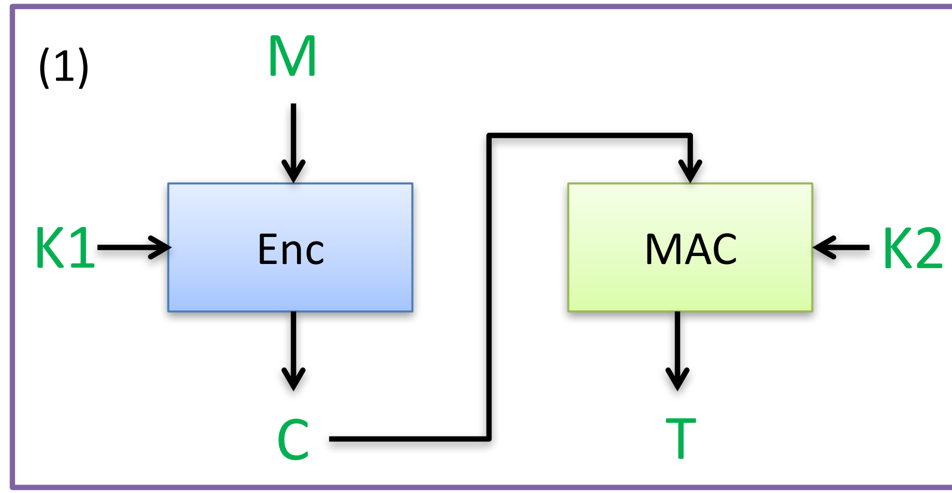
Encrypt-then-MAC in TLS: A poorly designed AE scheme

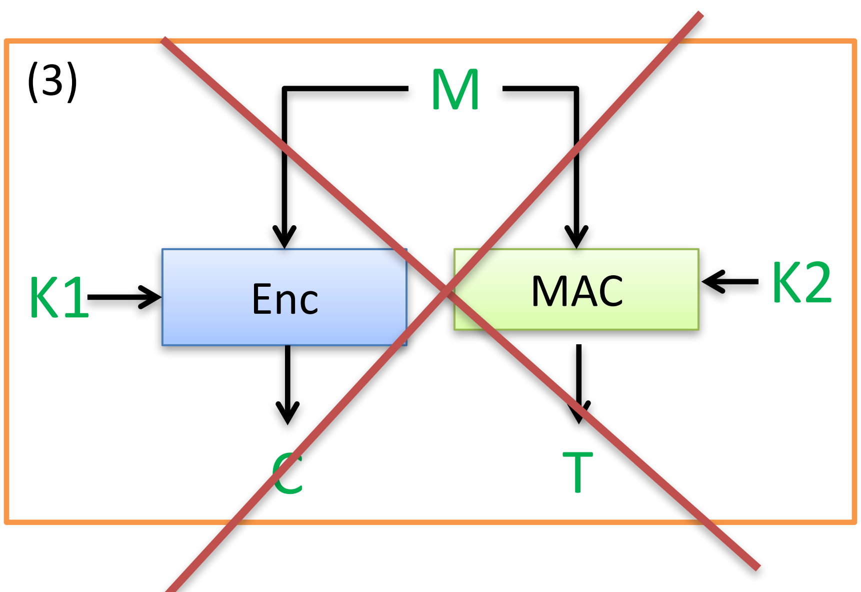
Several ways to combine:

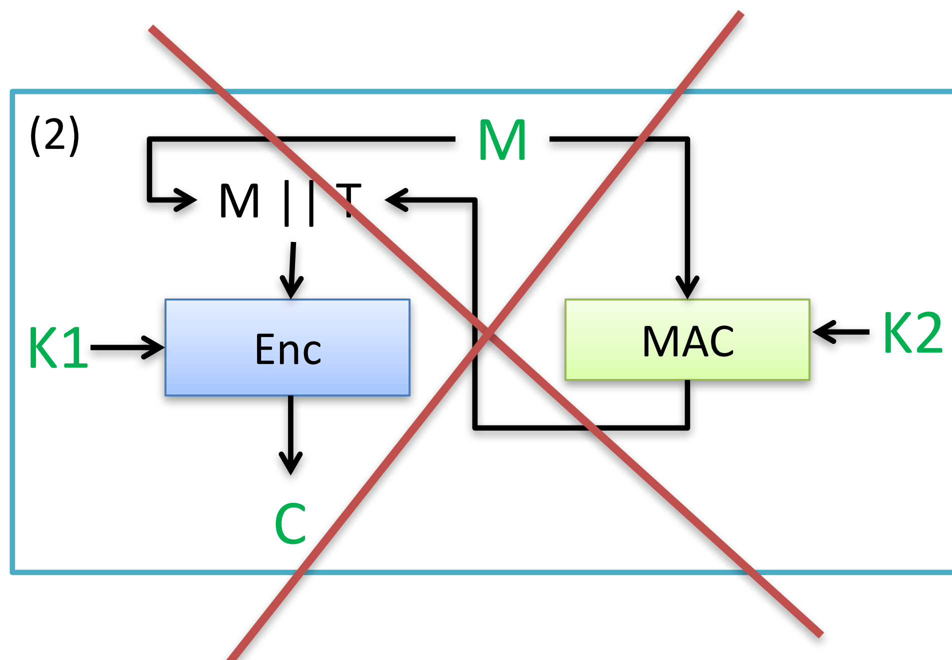
(1) encrypt-then-mac

(2) mac-then-encrypt

(3) encrypt-and-mac







Some other AE schemes

Attack	Inventors	Notes
OCB (Offset Codebook)	Rogaway	One-pass
GCM (Galios Counter Mode)	McGrew, Viega	CTR mode plus specialized MAC
CWC	Kohno, Viega, Whiting	CTR mode plus Carter-Wegman MAC
CCM	Housley, Ferguson, Whiting	CTR mode plus CBC-MAC
EAX	Wagner, Bellare, Rogaway	CTR mode plus OMAC

TLS 1.2 now supports GCM and AES-CBC-then-HMAC-SHA256

Summary

- TLS is one of the most widely used and studied protocols
- Record layer protocol handles authenticated-encryption of application-layer data
  - ~15 years of attack and hacky countermeasures
  - Padding oracles are almost impossible to get rid of with a bad AE algorithm
- We now finally have good AE schemes in TLS 1.2 and TLS 1.3

A short history of TLS

How many cryptographers involved?

SSL ver 2.0 designed by **Hickman** at Netscape	SSL ver 2	1994
Wagner, Goldberg break SSL ver 2		1995
Freier, Karlton, Kocher design SSL ver 3.0	SSL ver 3	
Bleichenbacher breaks RSA PKCS #1 encryption, used in SSL ver 3		1998
TLS ver 1 released as IETF standard, based on SSL 3, **many cryptographers** involved	TLS ver 1.0	1999
Vaudenay, Klima et al. padding attacks		2001
Rogaway IV re-use insecurity		2002
Brumley, Boneh remote timing attacks		2003
TLS ver 1.1 released as standard	TLS ver 1.1	2006

⋮

(more attacks and fixes)