Cryptographic Analysis of TLS

Kenny Paterson

FOSAD 2013

Royal Holloway
University of London

Information Security Group

Outline

- TLS overview
- TLS Record Protocol
  - Theory
  - Attacks
  - Security analysis
- TLS Handshake Protocol
  - Security analysis
- Discussion

Philosophy

- TLS is important.
- There has been lots of research on TLS lately.

- I'll try to convince you that bits and bytes are interesting and really matter.
  - But you'll have to leave your symbolic intuition behind!

- I'll try to teach some lessons about how hard cryptography is in the real world.
- And about the interplay between theoretical progress and practical application.

Outline

- **TLS overview**
- TLS Record Protocol
  - Theory
  - Attacks
  - Security analysis
- TLS Handshake Protocol
  - Security analysis
- Discussion

TLS Overview

- SSL = Secure Sockets Layer.
  - Developed by Netscape in mid 1990s.
  - SSLv1 broken at birth.
  - SSLv2 flawed, now IETF-deprecated (RFC 6176).
  - SSLv3 still widely supported.

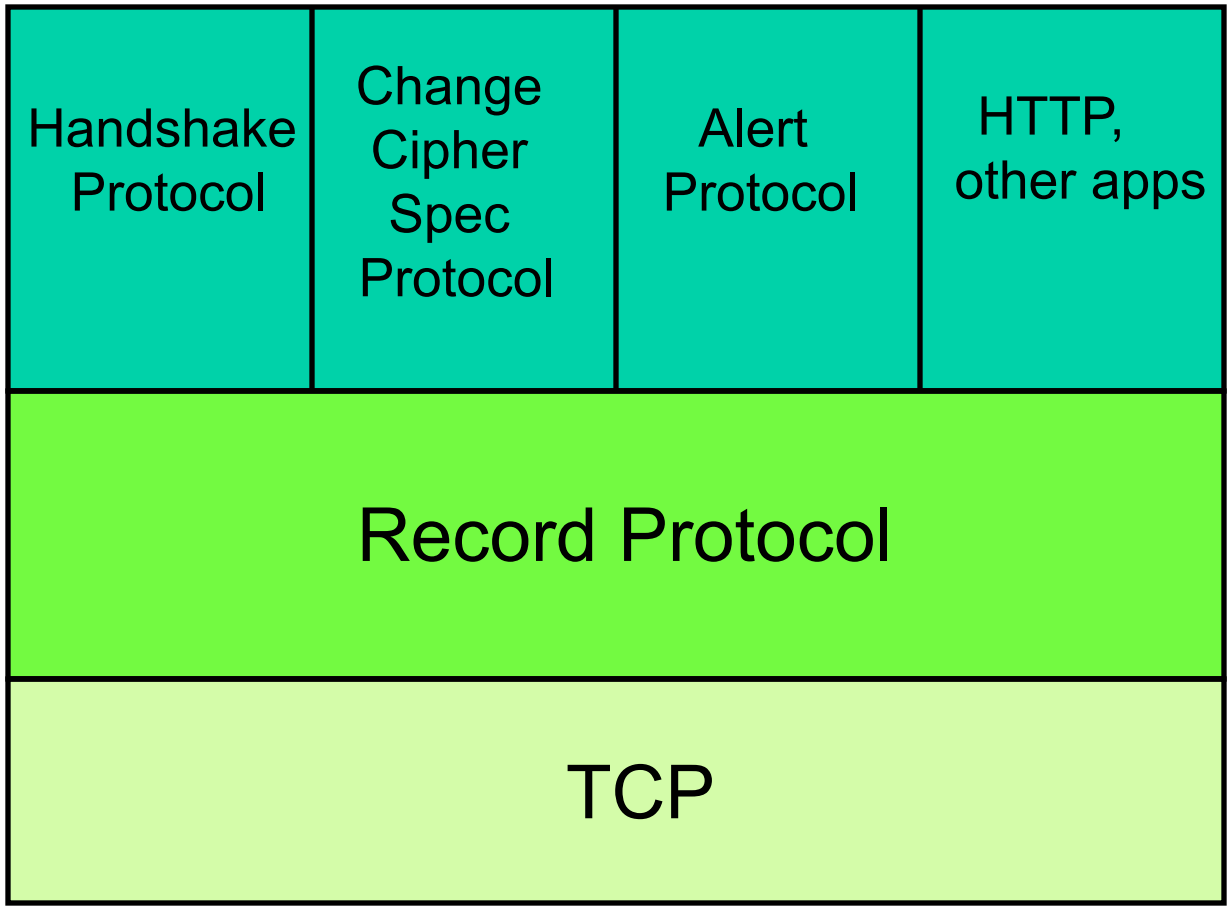
- TLS = Transport Layer Security.
  - IETF-standardised version of SSL.
  - TLS 1.0 in RFC 2246 (1999).
  - TLS 1.1 in RFC 4346 (2006).
  - TLS 1.2 in RFC 5246 (2008).

Importance of TLS

- Originally designed for secure e-commerce, now used much more widely.
  - Retail customer access to online banking facilities.
  - Access to gmail, facebook, Yahoo, etc.
  - Mobile applications, including banking apps.
  - Payment infrastructures.

- TLS has become the *de facto* secure protocol of choice.
  - Used by hundreds of millions of people and devices every day.
  - A serious attack could be catastrophic, both in real terms and in terms of perception/confidence.

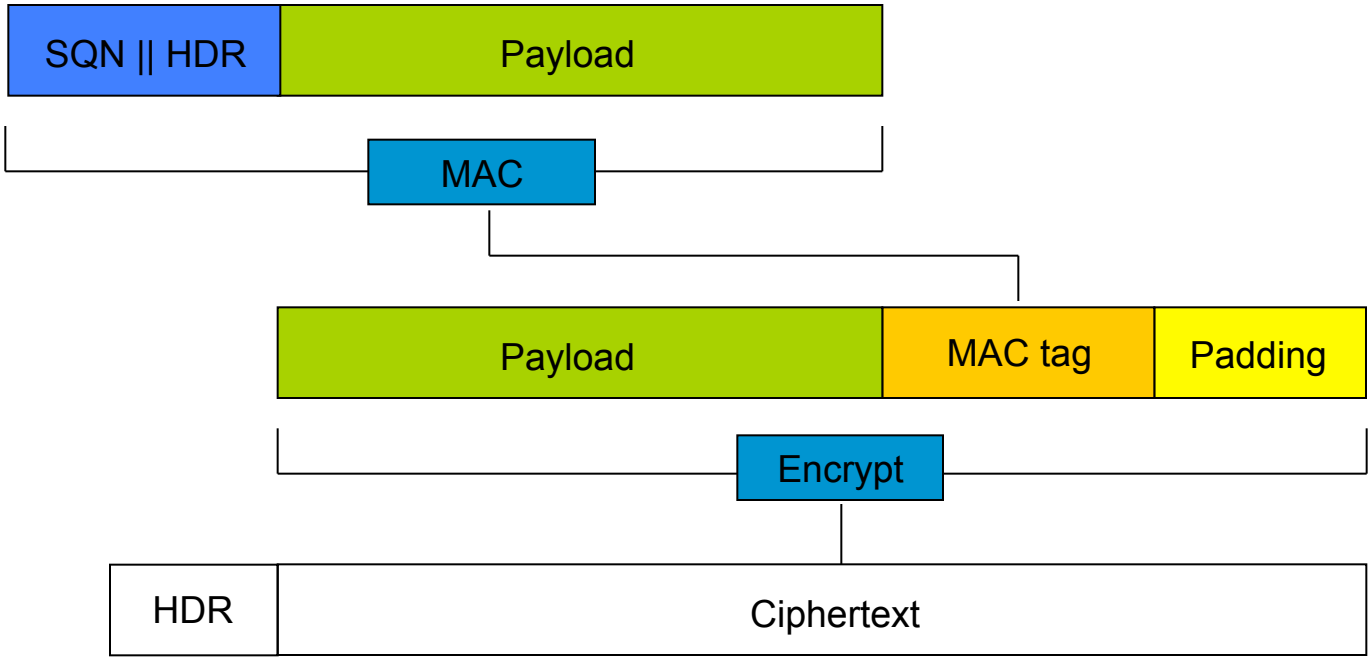
TLS Protocol Architecture



TLS Record Protocol

- TLS Record Protocol provides:
  - Data origin authentication, integrity using a MAC.
  - Confidentiality using a symmetric encryption algorithm.
  - Anti-replay using sequence numbers protected by the MAC.
  - Optional compression.
  - Fragmentation of application layer messages.

TLS Record Protocol: MAC-Encode-Encrypt (MEE)



MAC	HMAC-MD5, HMAC-SHA1, HMAC-SHA256
Encrypt	CBC-AES128, CBC-AES256, CBC-3DES, RC4-128
Padding	“00” or “01 01” or “02 02 02” or …. or “FF FF….FF”

Operation of TLS Record Protocol

- Data from layer above is received and partitioned into fragments (max size 2^{14} bytes).
- Optional data compression.
  - Default option is no compression.
- Calculate MAC on sequence number, header fields, and data, and append MAC to data.
- Pad (if needed by encryption mode), then encrypt.
- Prepend 5-byte header, containing:
  - Content type (1 byte, indicating content of record, e.g. handshake message, application message, etc),
  - SSL/TLS version (2 bytes),
  - Length of fragment (2 bytes).
- Submit to TCP.

Operation of TLS Record Protocol

In-bound processing steps reverses these steps:

1. Receive message, of length specified in HDR.
2. Decrypt.
3. Remove padding.
4. Check MAC.
5. (Decompress payload.)
6. Pass payload to upper layer (no defragmentation).

Errors can arise from any of decryption, padding removal or MAC checking steps.

All of these are *fatal* errors.

AE and TLS Record Protocol

- Dedicated Authenticated Encryption (AE) algorithms are supported in TLS 1.2 in addition to MEE.
  - Need not conform to MEE template.
  - AES-GCM specified in RFC 5288.
  - AES-CCM specified in RFC 6655.

- But TLS 1.2 is not yet widely supported.
  - Most browsers support SSLv3 and TLS 1.0 only.
  - Currently, roughly 50/50 usage split between CBC mode and RC4.
  - Less than 17% of servers support TLS 1.1 or higher (source: SSL Pulse), increasing at 1-2% per month.
  - [27% of servers still support SSLv2!]

- We will focus on CBC mode and RC4.

TLS Record Protocol Sequence Numbers

- Sequence number is 64 bits in size and is incremented for each new message.
- Sequence number not transmitted as part of message.
  - Each end of connection maintains its own view of the current value of the sequence number.
  - TLS is reliant on TCP to deliver messages in order.
- Wrong sequence number leads failure of MAC verification
  - A fatal error leading to TLS connection termination.
- Creates stateful encryption scheme.
  - Preventing replay, insertion, reordering attacks,…

TLS Handshake Protocol

- TLS consumes symmetric keys:
  - MAC and encryption algorithms in Record Protocol.
  - Different keys in each direction.
- TLS also needs initialization vectors (IVs) for some encryption algorithms.
- These keys and IVs are established by the Handshake Protocol and subsequent key derivation.
- The TLS Handshake Protocol is itself a complex protocol with many options…

TLS Handshake Protocol Security Goals

- Entity authentication of participating parties.
  - Participants are called ‘client’ and ‘server’.
    - Reflects typical usage in e-commerce.
  - Server nearly always authenticated, client more rarely.
  - Appropriate for most e-commerce applications.

- Establishment of a fresh, shared secret.
  - Shared secret used to derive further keys.
  - For confidentiality and authentication/integrity in SSL Record Protocol.

- Secure negotiation of all cryptographic parameters.
  - SSL/TLS version number.
  - Encryption and hash algorithms.
  - Authentication and key establishment methods.
  - To prevent version rollback and ciphersuite downgrade attacks.

TLS Handshake Protocol – Key Establishment

- TLS supports several key establishment mechanisms.

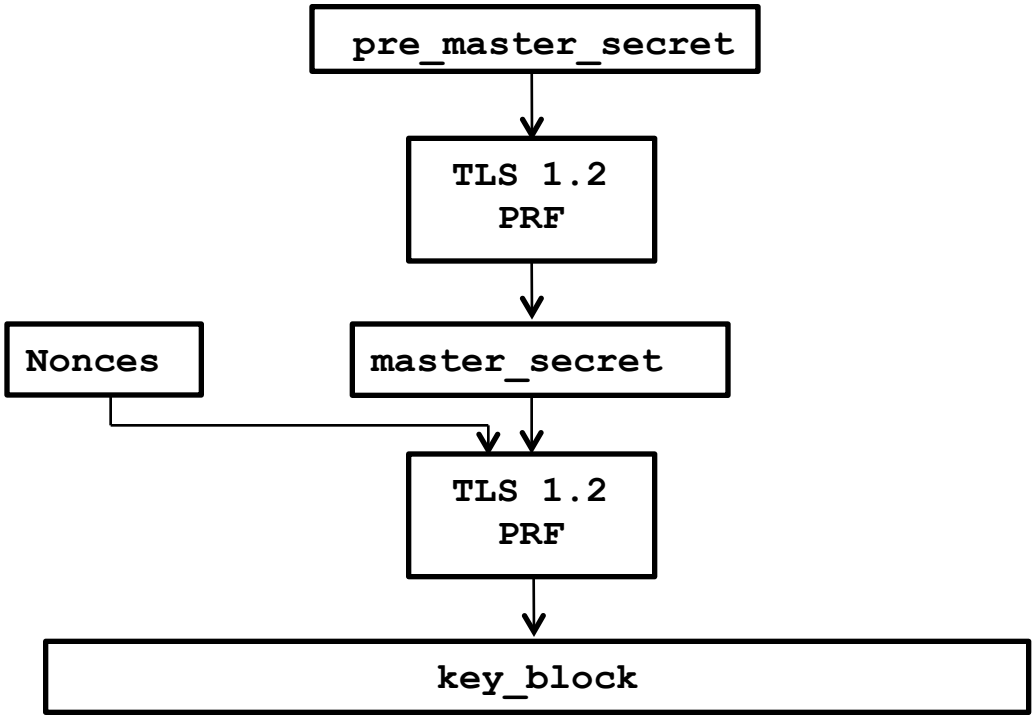
- Method used is negotiated during the Handshake Protocol itself.
  - Client sends list of *ciphersuites* it supports in `ClientHello`; server selects one and tells client in `ServerHello`.
  - e.g. TLS_RSA_WITH_AES_256_CBC_SHA256

- Very common choice is RSA encryption.
  - Client chooses `pre_master_secret`, encrypts using public RSA key of server, sends to server.
  - RSA encryption based on PKCS#1v1.5 padding method.

TLS Handshake Protocol – Key Establishment

- Can also create `pre_master_secret` from:
  - Static Diffie-Hellman
    - Server certificate contains DH parameters (group, generator *g*) and static DH value g^x.
    - Client chooses *y*, computes g^y and sends to server.
    - `pre_master_secret` = g^{xy}.
  - Ephemeral Diffie-Hellman
    - Server and Client exchange fresh Diffie-Hellman components in group chosen by server.
    - Signed (usually only by server) to provide authentication.
  - Anonymous Diffie-Hellman
    - Each side sends Diffie-Hellman values in group chosen by server, but no authentication of these values.
    - Vulnerable to man-in-middle attacks.

TLS Key Derivation



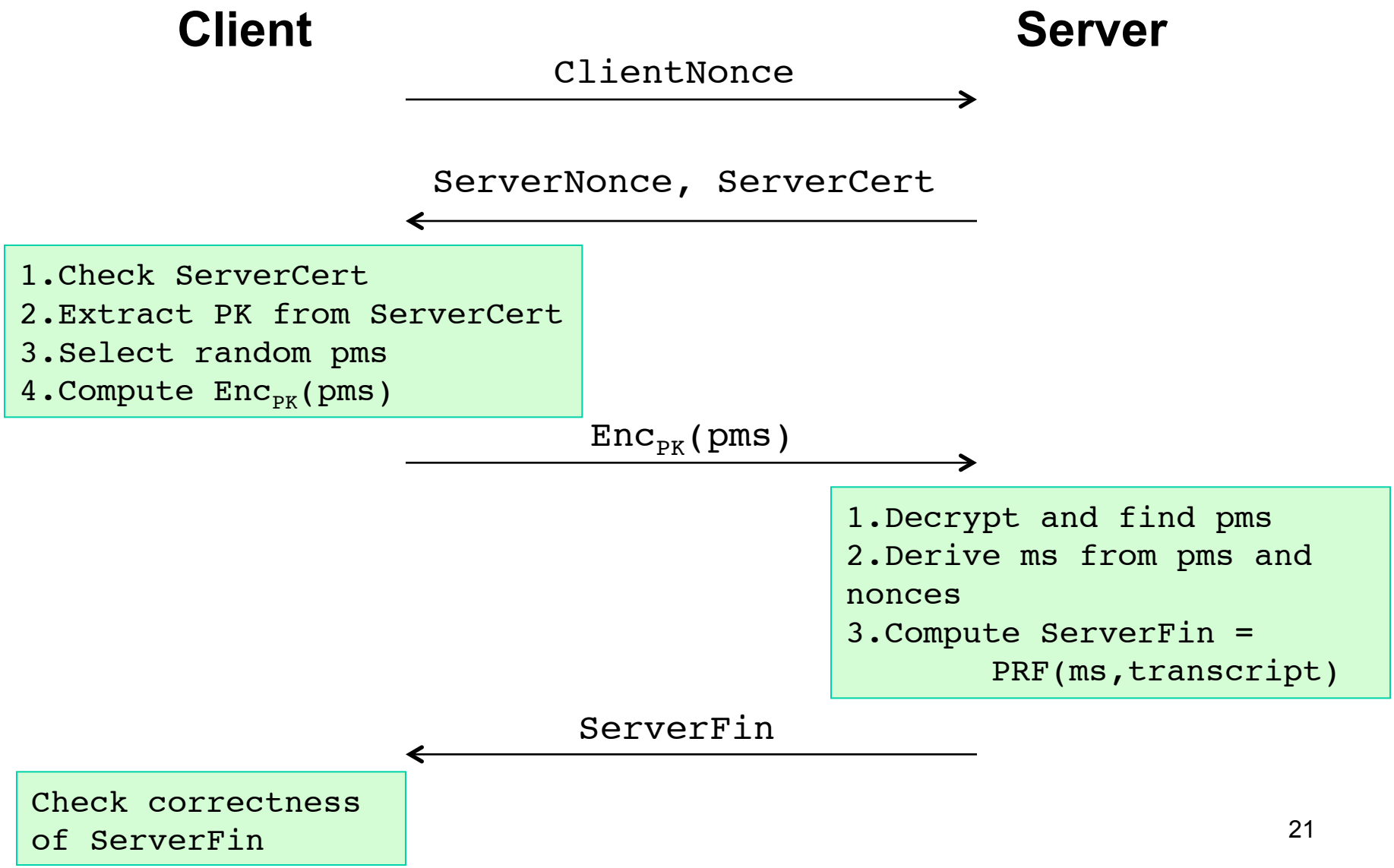
TLS Key Derivation

- Keys used by MAC and encryption algorithms in the Record Protocol are derived from `pre_master_secret`:
  - Derive `master_secret` from `pre_master_secret` using TLS Pseudo-Random Function (PRF).
    - PRF used can be negotiated during the Handshake Protocol (TLS1.2).
    - Default PRF for TLS1.2 is built by iterating HMAC-SHA256 in a specified way
  - Derive `key_block` from `master_secret` and client/server nonces exchanged during Handshake Protocol.
    - Again using the TLS PRF in TLS1.2.
  - Split up `key_block` into MAC keys, encryption keys and IVs for use in Record Protocol as needed.

TLS Handshake Protocol – Entity Authentication

- TLS supports several different entity authentication mechanisms for clients and servers.
- Method used is negotiated along with key exchange method during the Handshake Protocol itself.
  - Specified in ciphersuites.
- Most common server authentication method is based on RSA.
  - Ability of server to decrypt `pre_master_secret` using its private key and then generate correct PRF value in `finished` message using key derived from `pre_master_secret` authenticates server to client.

Authentication Based on RSA Encryption (simplified)



TLS Handshake Protocol – Entity Authentication

- Less common authentication methods:
  - Ability of server to derive key from server's static (private) DH value (in server certificate) and client's ephemeral (public) DH value.
  - ECDSA, DSA or RSA signatures on nonces (and other fields, e.g. Diffie-Hellman values).
  - Pre-shared key.
  - Shared password.

TLS Handshake Protocol Overview

M1: C → S: `ClientHello`

M2: S → C: `ServerHello, [Certificate, ServerKeyExchange, CertificateRequest,] ServerHelloDone`

M3: C → S: [`Certificate,`] `ClientKeyExchange, [CertificateVerify,] ChangeCipherSpec, ClientFinished`

M4: S → C: `ChangeCipherSpec,ServerFinished`

- `[]` denotes optional/situation-dependent field.
- `ChangeCipherSpec` messages not part of Handshake.

TLS Handshake Protocol – Additional Features

- TLS Handshake Protocol supports *ciphersuite renegotiation* and *session resumption*.
  - Ciphersuite renegotiation allows re-keying and change of ciphersuite during a session.
    - E.g., to force strong client-side authentication before access to a particular resource on the server is allowed.
    - Initiated by client sending `ClientHello` or server sending `ServerHelloRequest` over existing Record Protocol.
    - Followed by full Handshake Protocol.
  - Session resumption allows authentication and shared secrets to be reused across multiple *connections* in a single session.
    - E.g., allows fetching next web-page from same website without re-doing full, expensive Handshake Protocol.

TLS Handshake Protocol – Session Resumption

Client and server run lightweight version of Handshake Protocol:

1. C → S: `ClientHello`
   (quoting existing `SessionID`, new **`ClientNonce`** and list of ciphersuites).

2. S → C: `ServerHello`
   (repeating `SessionID`, new **`ServerNonce`** and selected ciphersuite),
   `ChangeCipherSpec`, `Finished`.

3. C→ S: `ChangeCipherSpec`, `Finished`.

- New `key_block` is derived by both sides.
  - New keys and IVs derived from new nonces and existing `master_secret`.
- Exchange protected by existing Record Protocol.
- No public key operations involved in session resumption.

TLS Sessions and Connections

- Session concept:
  - Sessions are created by the Handshake Protocol.
  - Session state defined by session ID and set of cryptographic parameters (encryption and hash algorithm, master secret, certificates) negotiated in Handshake Protocol.
  - Each session can carry multiple **sequential** *connections*.

- Connection concept:
  - Keys for multiple connections are derived from a single `master_secret` created during one run of the full Handshake Protocol.
  - Session resumption Handshake Protocol runs exchange new nonces.
  - These nonces are combined with existing `master_secret` to derive keys for each new connection.
  - Avoids repeated use of expensive Handshake Protocol.

Other TLS Protocols

- Alert protocol.
  - Management of SSL/TLS connections and sessions, error messages.
  - Fatal errors and warnings.
  - Defined actions to ensure clean session termination by both client and server.
- Change cipher spec protocol.
  - Technically not part of Handshake Protocol.
  - Used to indicate that entity is changing to recently agreed ciphersuite.
- Both protocols run over Record Protocol (so are peers of Handshake Protocol).

SSL and TLS

TLS 1.0 = SSLv3.0 with minor differences, including:

- TLS signalled by version number 3.1.
- Use of HMAC for MAC algorithm in TLS 1.0.
- Different method for deriving keying material (`master_secret` and `key_block`).
  - TLS 1.0 uses PRF based on HMAC with MD5 and SHA-1 operating in combination.
- Additional alert codes.
- More client certificate types.
- Variable length padding.
  - Can be used to hide lengths of short messages and so limit traffic analysis.

Evolution of TLS

- TLS continues to evolve.
- TLS 1.1 (RFC 4346, 2006) obsoletes TLS 1.0 (RFC 2246).
  - Uses explicit IVs instead of IV chaining to prevent attacks based on predictable IVs (see later).
  - Attempts to protect against padding oracle attacks (see later).

Evolution of TLS

- TLS 1.2 (RFC 5246) published in 2008 obsoletes TLS 1.1 (RFC 4346).
  - Removal of dependence on MD-5 and SHA-1 hash algorithms for PRFs.
    - Now negotiable in Handshake Protocol, but specific PRF based on HMAC-SHA256 as standard.
  - Support for authenticated encryption modes.
  - Removed support for some cipher suites.
- Adoption of TLS 1.1 and 1.2 not yet widespread.
  - But this is expected to change soon because of recent high-profile attacks.
  - ☺

TLS Extensions and DTLS

- Many extensions to TLS exist.
- Allows extended capabilities and security features.
- Examples:
  - Renegotiation Indicator Extension (RIE), RFC 5746.
  - Application layer protocol negotiation (ALPN), draft RFC.
  - Authorization Extension, RFC 5878.
  - Server Name Indication, Maximum Fragment Length Negotiation, Truncated HMAC, etc, RFC 6066.

- DTLS is effectively “TLS over UDP”
  - DTLS 1.0 aligns with TLS 1.1, and DTLS 1.2 with TLS 1.2.
  - UDP provides unreliable transport, so DTLS must be error tolerant, necessitating changes to Handshake Protocol and error management.

Outline

- TLS overview
- **TLS Record Protocol**
  - **Theory**
  - Attacks
  - Security analysis
- TLS Handshake Protocol
  - Security analysis
- Discussion

Theory for TLS Record Protocol

- Security models for symmetric encryption as used in TLS Record Protocol are well established.

- Syntax: SE = (KGen,Enc,Dec)
  - Probablistic KGen(1^k), outputs key *K.*
  - Probablistic Enc: $c \leftarrow \mathrm{Enc}_K(m)$.
  - Deterministic Dec, outputs message *m* or failure symbol $\perp$:

$$m / \perp \leftarrow \mathrm{Dec}_K(m).$$

  - Correctness requirement:

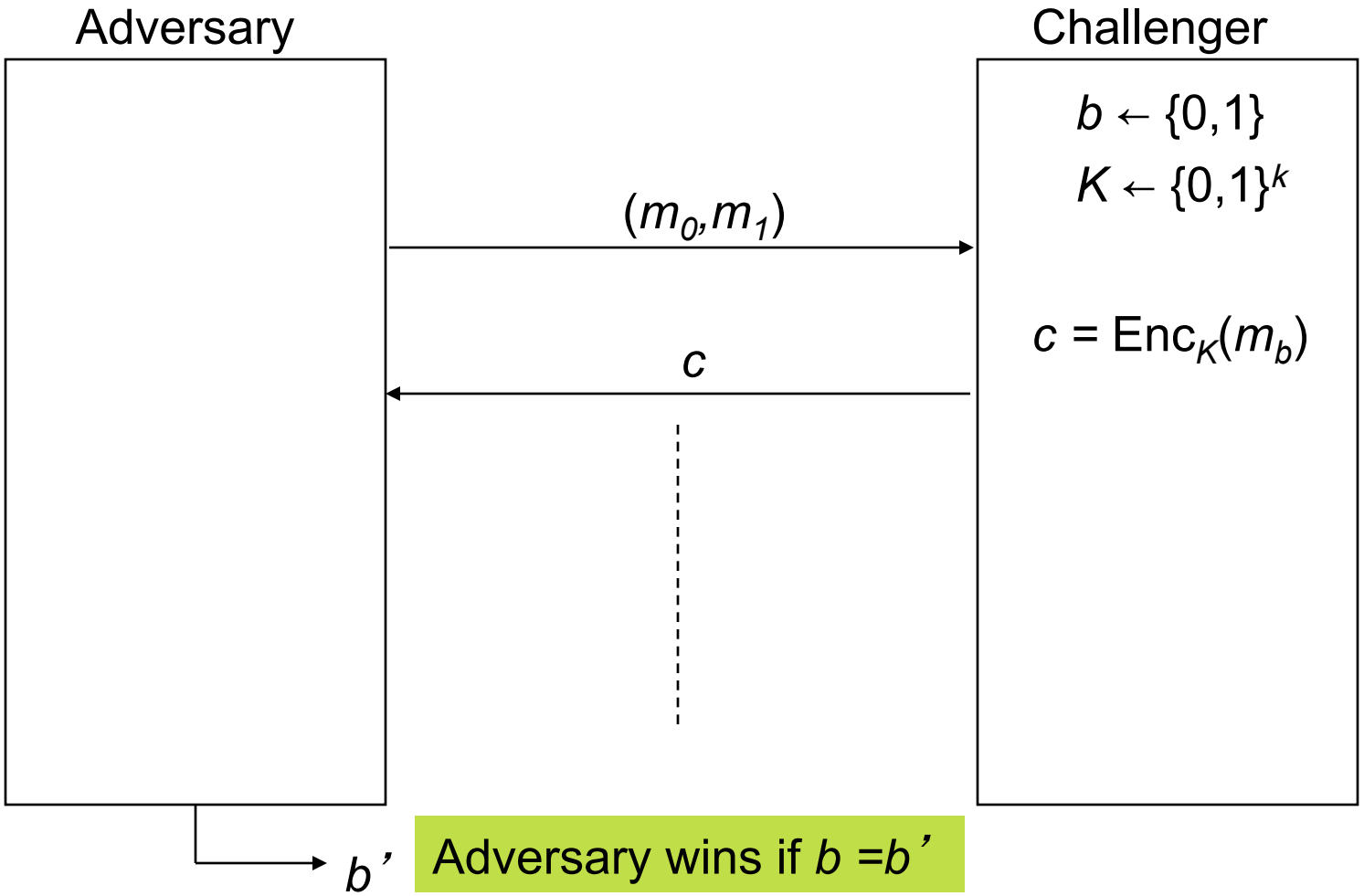
$$\mathrm{Dec}_K(\mathrm{Enc}_K(m)) = m.$$

Theory for TLS Record Protocol

- IND-CPA security:
  - Adversary has repeated access to *Left-or-Right (LoR)* encryption oracle.
  - Adversary submits pairs of equal length messages (m_0,m_1) to the oracle.
  - Receives *c*, an encryption of either m_0 or encryption of m_1.
  - Adversary has decide which message c encrypts.
  - Adversary wins if it decides correctly.

- Formalised as a security game between the adversary and a challenger.

IND-CPA Security



IND-CPA Security

- Adversary's advantage is defined to be:

$$|\Pr(b=b') - 1/2|.$$

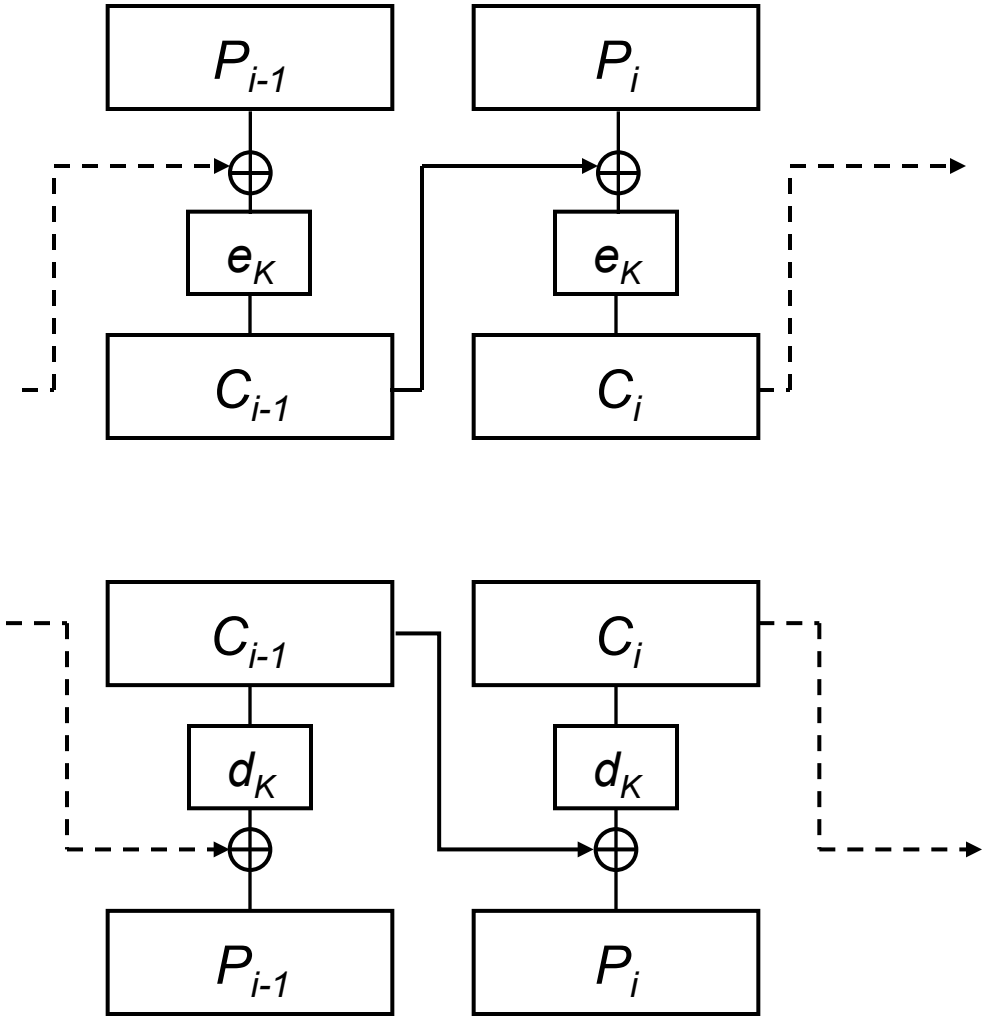
- A scheme SE is said to be IND-CPA secure if advantage is "small" for any adversary using "reasonable" resources.
  - Concepts of "small" and "reasonable" can be formalised using either an asymptotic approach or a concrete approach.
  - Meeting the definition requires non-deterministic (randomised) encryption.
  - IND = Indistinguishable.
  - CPA = Chosen Plaintext Attack.

IND-CPA Security

- Informally, IND-CPA is a computational version of perfect security.
  - Ciphertext leaks nothing about the plaintext.
  - Stronger notion than requiring the adversary to recover plaintext.

- Easy to achieve using suitable mode of operation of block cipher:
  - Block cipher in CBC mode with random IVs.
  - Block cipher in CTR mode.
  - See [BDJR97] for analysis.
  - Requires modelling of block cipher as PRF.

CBC Mode



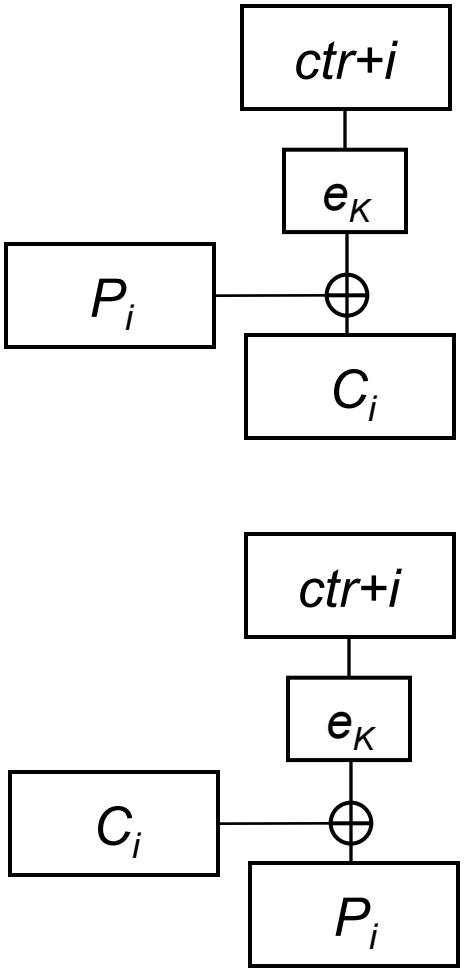
Initialisation Vector (IV):

- Defines C_0 for processing first block.
- IV often taken as random;
- Chained IVs also common

CBC mode needs some form of padding if plaintext lengths are not multiple of block length.

- More on padding later.

CTR Mode



- CTR mode uses a block cipher to build a stream cipher.
- Random initial value chosen for *ctr* (or maintained as state).
- Encrypt blocks

  ctr, ctr+1, ctr+2, …

  to create a sequence of ciphertext blocks.
- Use this sequence as keystream.
- Same process to decrypt.
- IND-CPA secure assuming block cipher is a PRF.

Motivating Stronger Security

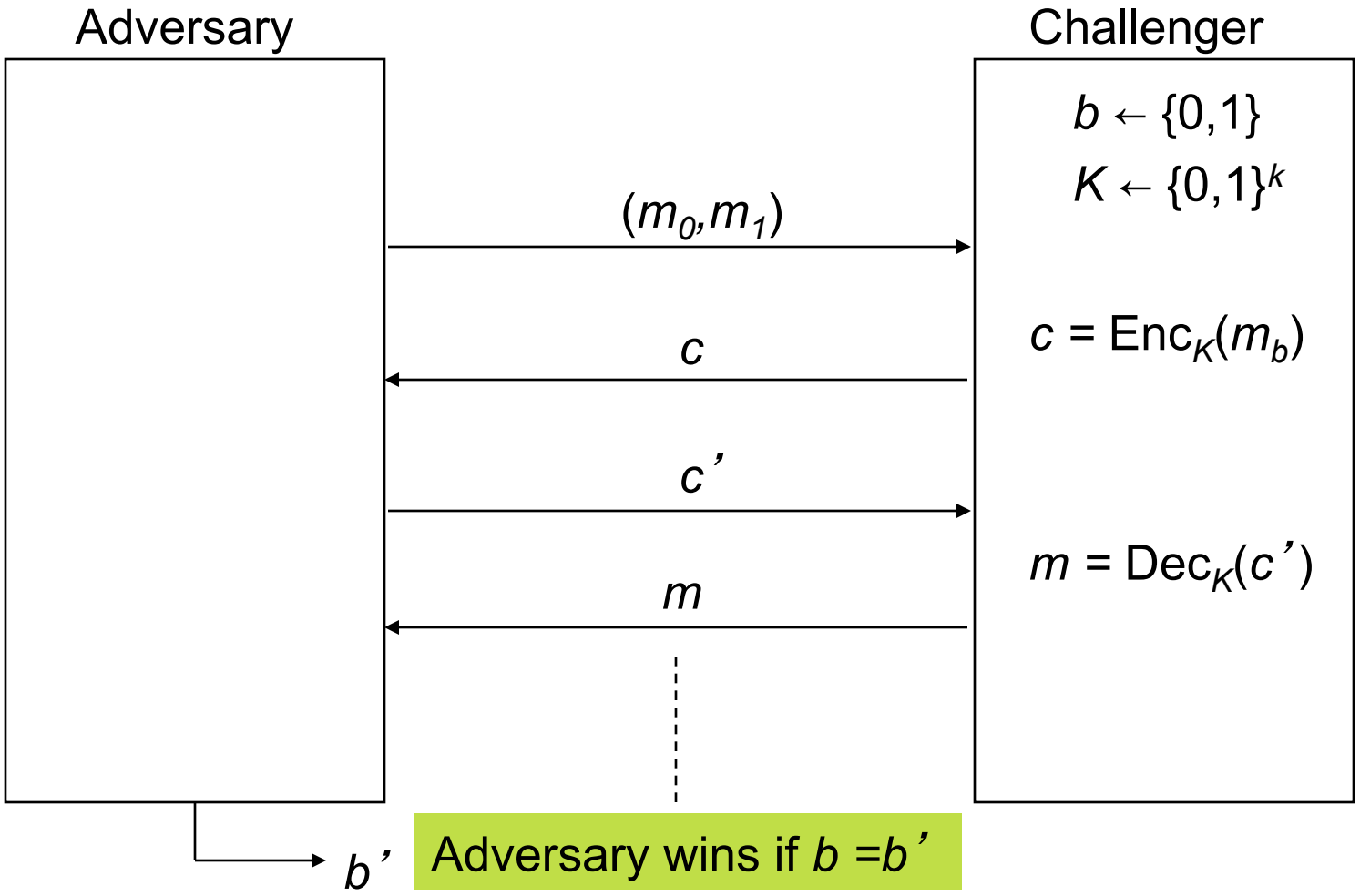
- With CBC mode and CTR mode, an active adversary can manipulate ciphertexts.
  - Modify *c* to *c'* and change the underlying plaintext from *p* to *p'.*
  - Or create completely new ciphertexts.
  - Does not break IND-CPA security, but is clearly undesirable for building secure channels.
  - We really want *non-malleable* encryption, guaranteeing integrity as well as confidentiality.

- We may also want to consider chosen-ciphertext attacks, where adversary can get ciphertexts of his choice decrypted.
  - This may arise in practice depending on application.

IND-CCA Security

- IND-CCA security:
  - Attacker now has repeated access to LoR encryption oracle *and to a decryption oracle*.
  - LoR encryption oracle as before.
  - Decryption oracle takes any c as input, and outputs either $\mathrm{Dec}_K(c)$, which is either a message m or a failure symbol $\perp$.
  - Adversary not permitted to submit output of LoR encryption oracle to its decryption oracle.
  - To prevent trivial win.
- All basic modes of operation are insecure in this model!
  - Trivial for CTR mode.

IND-CCA Security



INT-PTXT Security

- INT-PTXT security:
  - Attacker has repeated access to *an encryption oracle and a "try" oracle.*
  - Encryption oracle takes any m as input, and outputs $\mathrm{Enc}_K(m)$.
  - Adversary's task is to submit c^* to its *try* oracle such that $\mathrm{Dec}_K(c^*)$ decrypts to message m^* that is distinct from all m queried to encryption oracle.
  - Hence win if adversary creates "plaintext forgery".
  - Scheme is INT-PTXT secure if no such efficient adversary exists.
  - Clearly a desirable property of encryption scheme used for building a secure channel, as it prevents (plaintext) message injection.

INT-CTXT Security

- INT-CTXT security:
  - As INT-PTXT, but only requirement is that *c** be valid ciphertext (could be another encryption of some *m*).
  - Hence win if adversary creates “ciphertext forgery”.
  - (Application to secure channel not immediately clear.)
- INT-CTXT security implies INT-PTXT security.
- IND-CPA + INT-CTXT → IND-CCA [BN00]

Authenticated Encryption Security

- AE := IND-CPA + INT-CTXT → IND-CCA
- More elegant “all-in-one definition” possible [RS06].
  - One version will be given later.
- Often easier to prove IND-CPA and INT-CTXT separately than to prove IND-CCA directly.
- AE security has become the accepted security target for encryption schemes.
- AEAD:
  - AEAD = “AE with Additional Data”.
  - Extension to AE allowing some data to be encrypted and remainder to be authenticated/integrity protected.
  - Sample application is TLS Record Protocol data: header is integrity protected, rest of payload is encrypted and integrity protected.

Stateful Security for Symmetric Encryption

- [BKN02] developed *stateful* security models for symmetric encryption.
  - Reflecting the desire to protect the *order* of messages in the secure channel.
  - And wide use of sequence numbers in secure channel protocols.

- IND-sfCCA security:
  - Attacker has access to an LoR encryption oracle and a decryption oracle.
  - Both oracles are stateful (e.g. via sequence numbers).
  - Model allows adversary to advance states to any chosen value via queries to LoR encryption and decryption oracles.
  - Adversary wins game if he can guess hidden bit *b* of encryption oracle.

- sfAE security can be defined similarly.

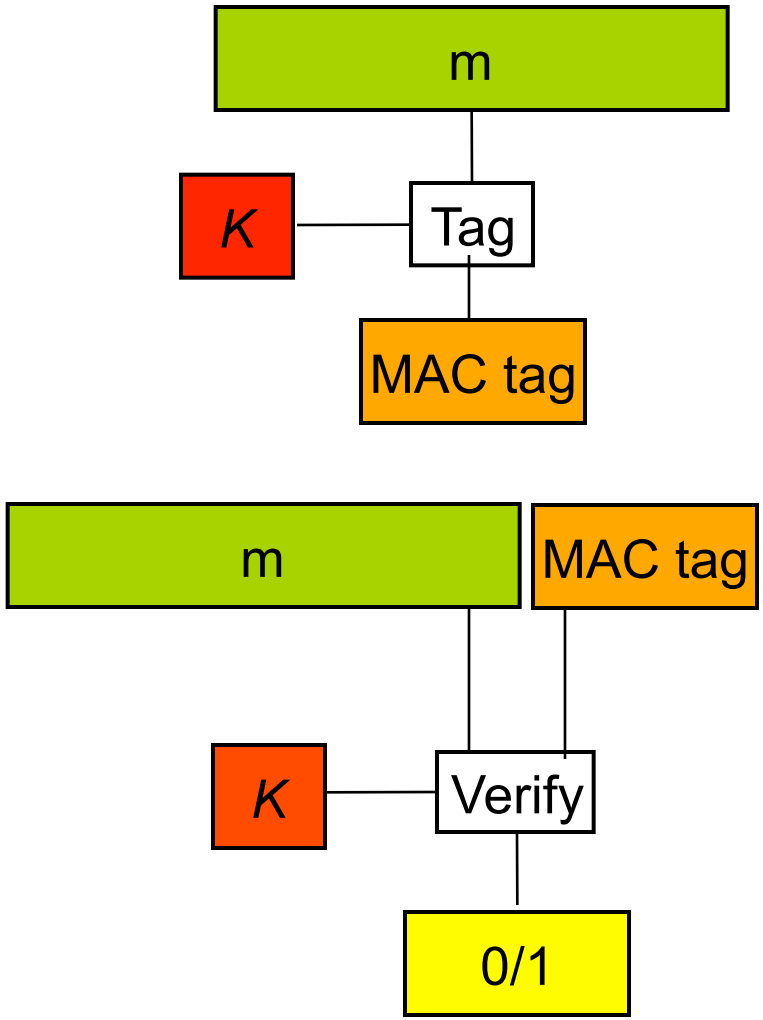
MACs

- Message Authentication Codes (MACs) provide authenticity/integrity protection for messages.
  - Symmetric analogue of a digital signature.

- Syntax: MAC = (KGen,Tag,Verify).
  - Tag has as input a key *K*, a message *m* of arbitrary length, and outputs a short MAC tag *τ*.
  - Verify has as input a key *K*, a message *m,* a MAC tag *τ* and outputs 0 or 1, indicating correctness of tag *τ* for *m* under *K.*

- HMAC is a general method for building a MAC scheme from a hash function.
  - Very widely used in secure protocols.

MACs



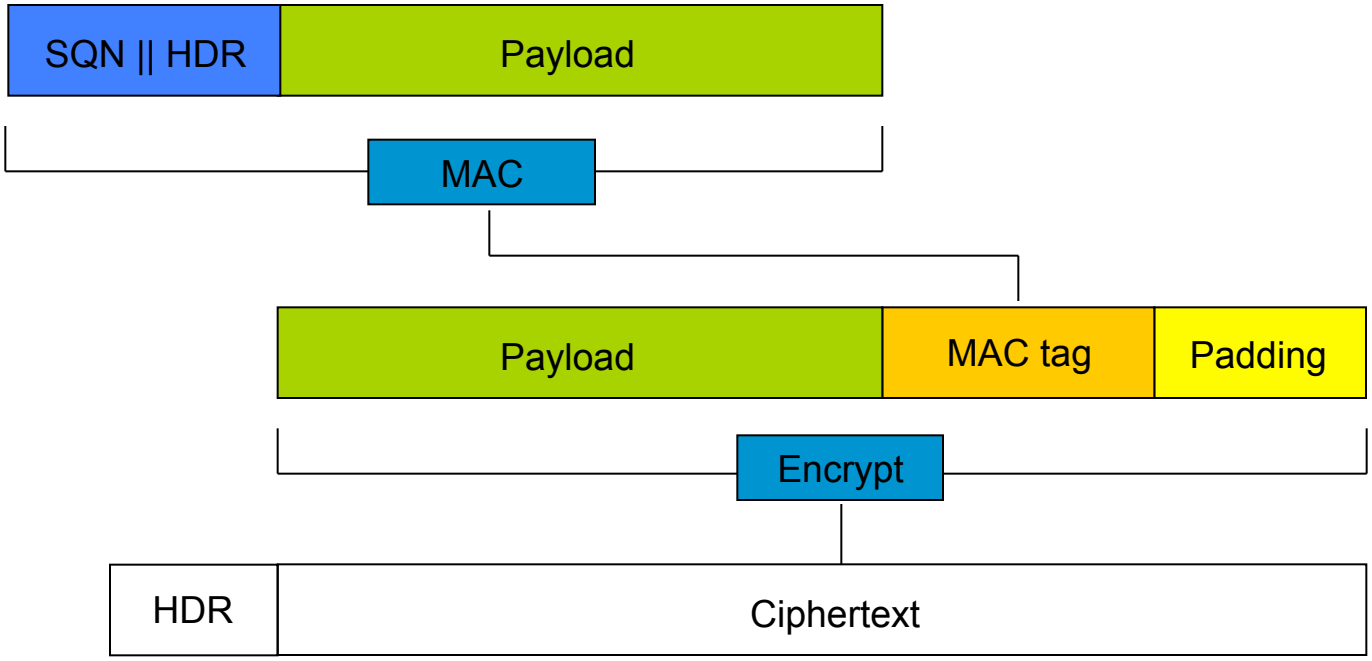
- Key security requirement is *unforgeability*.
- Having seen MAC tags for many chosen messages, an adversary cannot create the correct MAC tag for another chosen message.
- Strong and weak forms of unforgeability:
  - New MAC tag on (possibly) queried message versus MAC tag on unqueried message.
  - SUF-CMA and (W)UF-CMA security

Achieving AE Security via Generic Composition

- [BN00] considered how to achieve IND-CCA/AE security by generic composition of IND-CPA secure encryption schemes and (S)UF-CMA secure MACs.

- Encrypt-then-MAC: achieves AE security
  - As used by IPsec ESP enc+auth.
  - Needs SUF-CMA MAC.
- Encrypt-and-MAC: **Not** even CPA secure in general!
  - MAC can leak plaintext information but still be SUF-CMA secure.
  - But specific instantiations may be AE/IND-CCA secure, e.g. as used in SSH [BKN02,PW10].
- MAC-then-Encrypt: **Not** CCA secure in general!
  - But easy to show IND-CPA and INT-PTXT security.

Application to TLS Record Protocol



MAC	HMAC-MD5, HMAC-SHA1, HMAC-SHA256
Encrypt	CBC-AES128, CBC-AES256, CBC-3DES, RC4-128
Padding	“00” or “01 01” or “02 02 02” or …. or “FF FF….FF”

Application to TLS Record Protocol

- The TLS Record Protocol employs a (stateful) MAC-then-encrypt composition.
  - With associated data (the Record header).

- This is known to be **not** generically secure, according to the results of [BN00].
  - We can construct an IND-CPA secure encryption scheme and a SUF-CMA secure MAC scheme for which the MAC-then-encrypt composition fails to be IND-CCA secure.
  - But it is INT-PTXT and IND-CPA secure.
  - Is that enough?

Application to TLS Record Protocol

- Building on results of Krawczyk [K01], the basic MAC-then-encrypt construction can be shown to be AE (and so IND-CCA) secure *for the special case of CBC mode encryption*.

- This extends to the stateful setting, as formalised in [BKN02].

- AE security also holds for RC4 under the assumption that its output is pseudorandom.

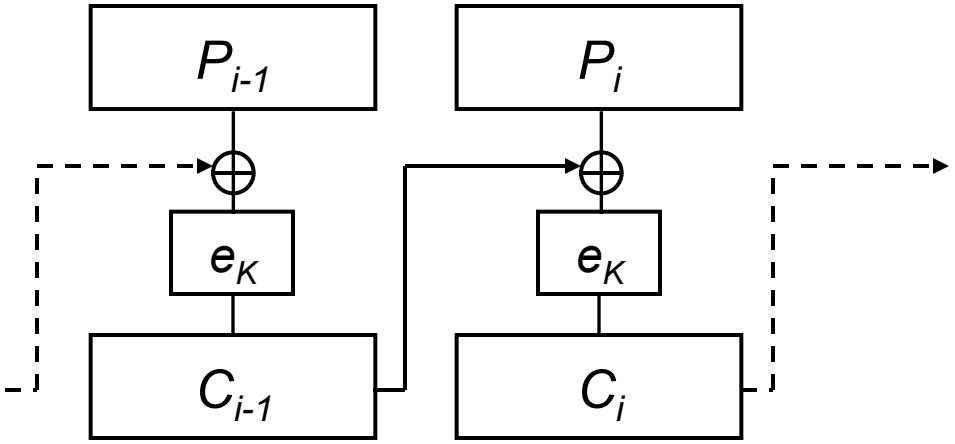
Application to TLS – Caveats

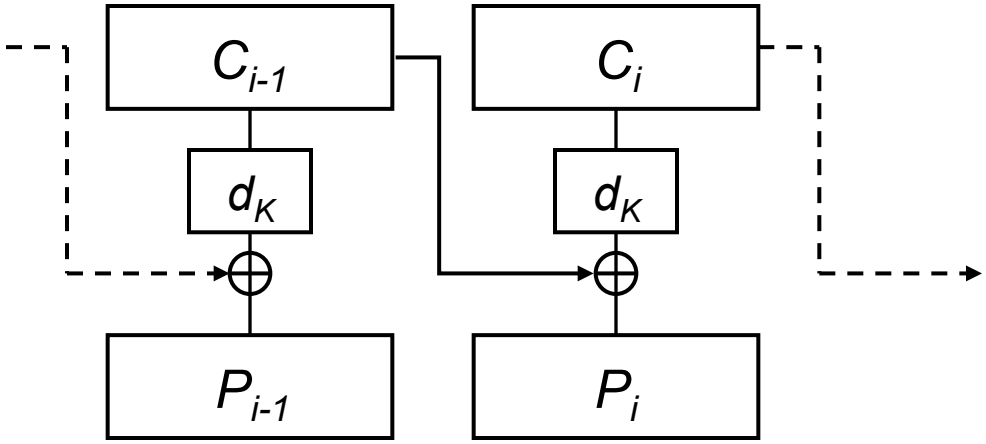
- Krawczyk's analysis assumes random IVs for CBC mode.
  - SSL v3.0 and TLS 1.0 use chained IVs.
- TLS is really using MAC-Encode-Encrypt.
  - With a specific padding scheme for the Encode step.
  - Decryption can fail in more than one way, so potentially multiple decryption failure symbols $\perp_1, \perp_2, \perp_3, \ldots$
- Padding does not arise anywhere in the analysis in [K01].
  - Data is assumed to be block-aligned, and MAC size = block size.
- RC4 has known statistical weaknesses.
- Do these gaps between theory and reality matter?

Outline

- TLS overview
- **TLS Record Protocol**
  - Theory
  - **Attacks**
  - Security analysis
- TLS Handshake Protocol
  - Security analysis
- Discussion

CBC Mode in TLS





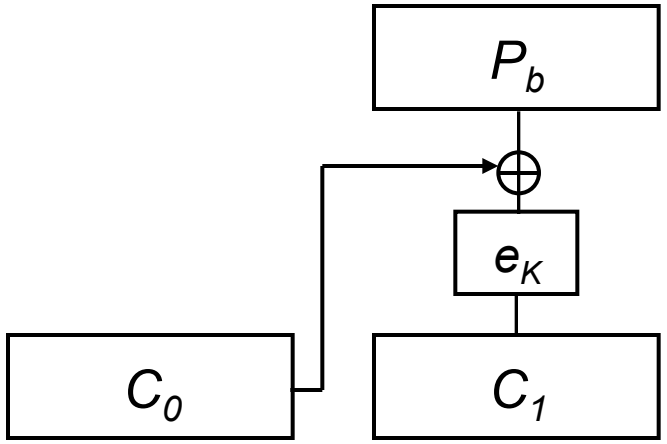
- SSLv3 and TLS 1.0 use a chained IV in CBC mode.
  - IV for current message is the last ciphertext block from the previous message.
- Modified in TLS 1.1, 1.2.
  - TLS 1.2 now has explicit IV and recommends IV SHOULD be chosen at random for each message.

Attacking Predictable IVs

- IV chaining in SSLv3 and TLS 1.0 leads to a *chosen-plaintext distinguishing attack* against TLS.
  - First observed for CBC mode in general by Rogaway in 1995.
  - Then applied to TLS by Dai and Moeller in 2004.
  - Extended to theoretical plaintext recovery attack by Bard in 2004/2006.
  - Turned into a **practical** plaintext recovery attack on HTTP cookies by Duong and Rizzo in 2011.
  - The BEAST!

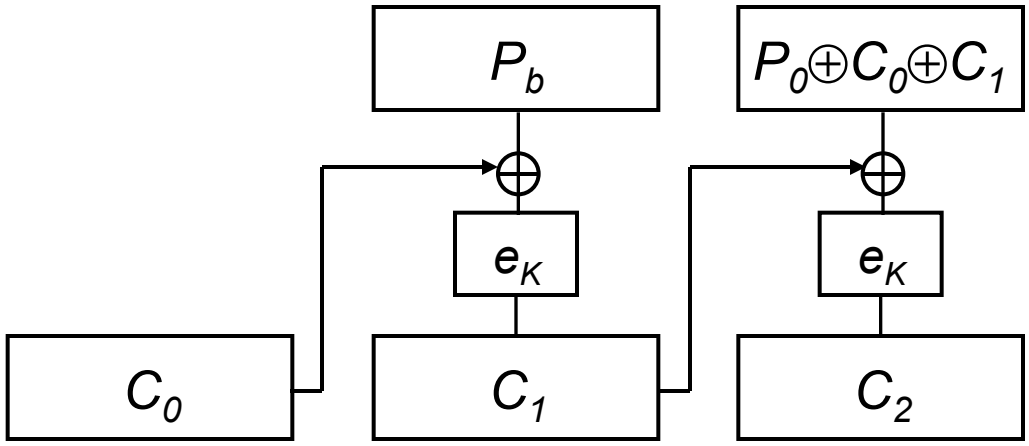
Attacking Predictable IVs

- Suppose attacker wishes to distinguish encryptions of single blocks P_0, P_1.
- Attacker asks for encryption of message P_0 or P_1.
- Attacker receives ciphertext $C = C_1$ for message P_b where some known, previous block C_0 was used as the IV.

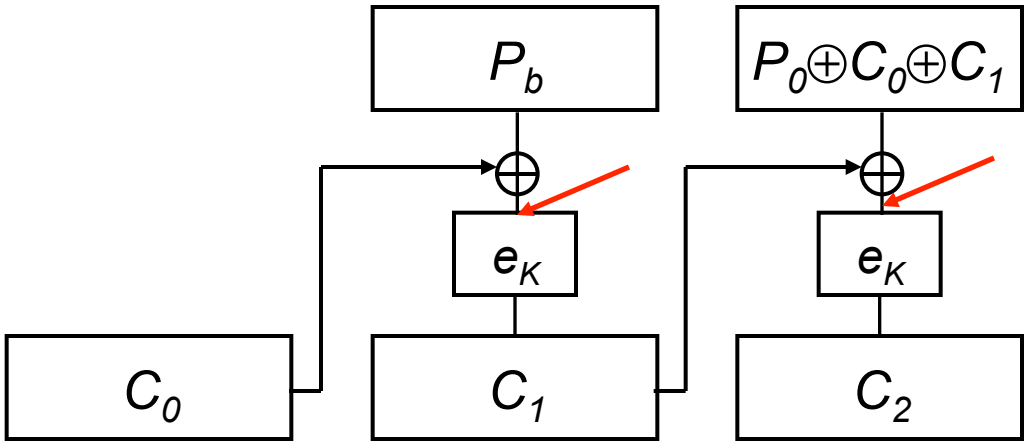


Attacking Predictable IVs

- C_1 will be used as the IV for the *next* encrpytion.
- Attacker now asks for the encryption of the single block $P_0 \oplus C_0 \oplus C_1$.
- Attacker receives single block ciphertext C_2.

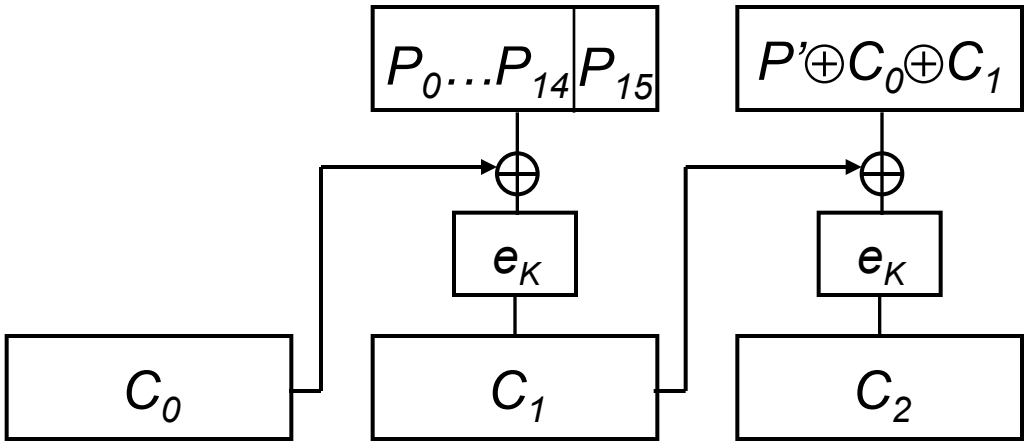


Attacking Predictable IVs



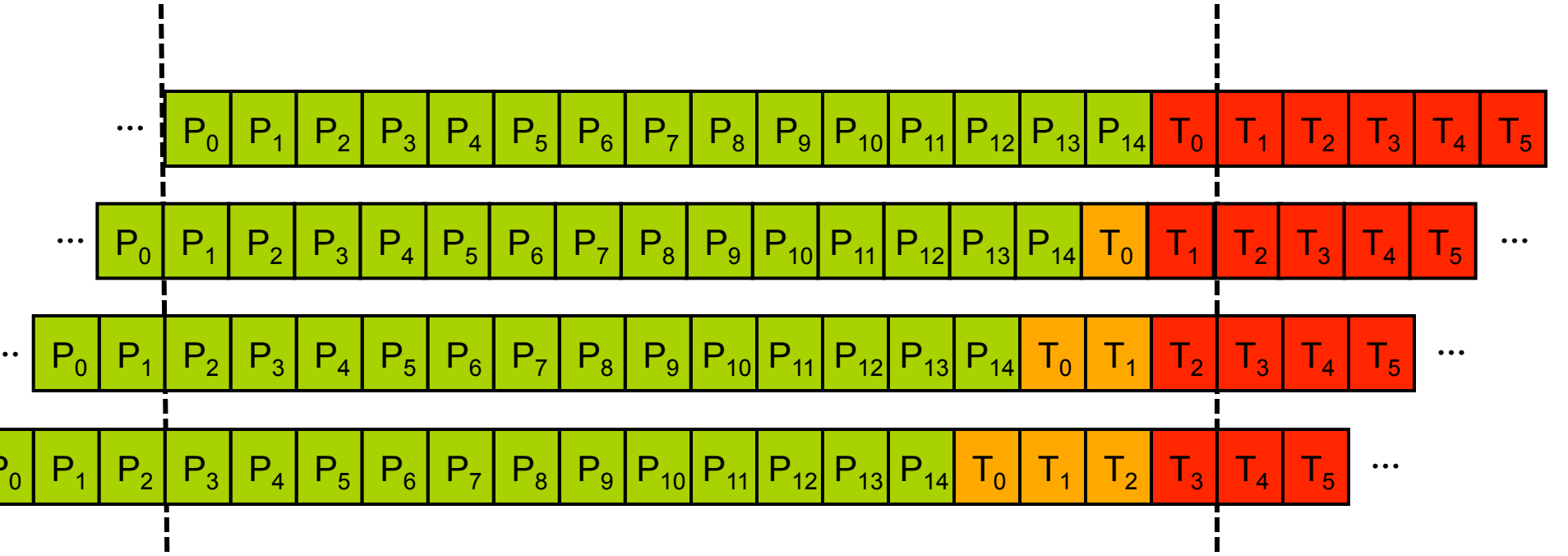
- If $P_b = P_0$, then inputs to block cipher are the same in both encryptions.
- Hence, if $P_b = P_0$, then $C_1 = C_2$.
- Otherwise, if $P_b = P_1$, then C_1 is not equal to C_2.
- So looking at C_1 and C_2 gives us a test to distinguish encryptions of P_0 and P_1.

The BEAST – Part 1



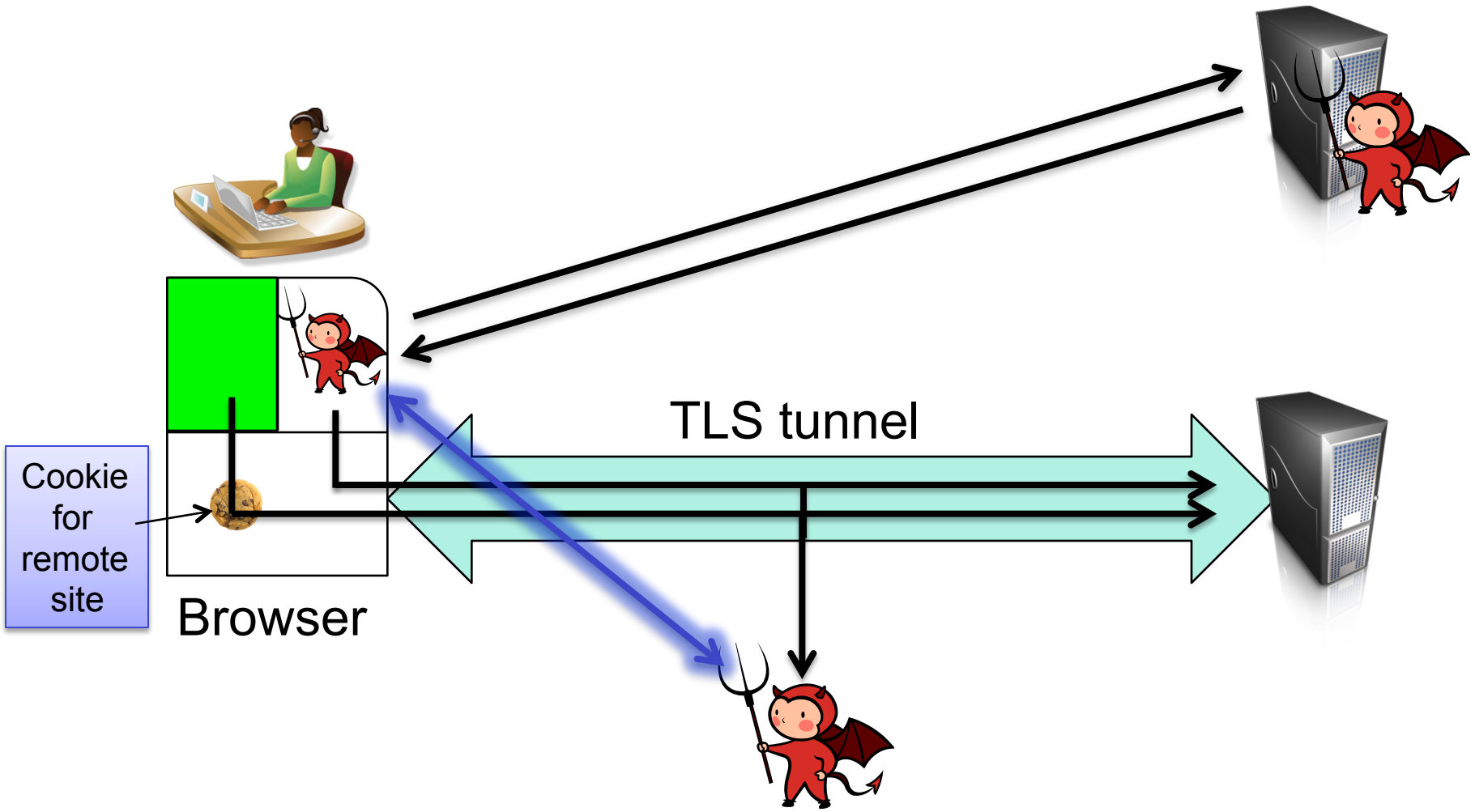
- Assume bytes $P_0, P_1, \ldots P_{14}$ are known, try to recover P_{15}.
- Use $P_0 P_1 \ldots P_{14}$ as first 15 bytes of P'.
- Iterate over 256 possible values in last position (15) in P'.
- $P'_{15} = P_{15}$ if and only if $C_2 = C_1$.
- So average of 128 trials to extract P_{15} when remaining bytes in block are known.

The BEAST – Part 2



- Now assume attacker can control position of bytes in stream with respect to CBC block boundaries (chosen boundary privilege).
- Repeat previous single-byte recovery attack with sliding bytes.
  - Green: initially known bytes.
  - Red: unknown (target) bytes.
  - Orange: recovered bytes.

The BEAST and JavaScript



The BEAST

- Key features:
  - BEAST JavaScript loaded ahead of time into client browser.
  - Target is HTTP secure cookie.
  - Use HTTP padding to control positions of unknown bytes.
  - Browser Same Origin Policy (SOP) bypass is needed to get fine control over byte/block positions.
  - JavaScript needs to communicate with MITM attacker.

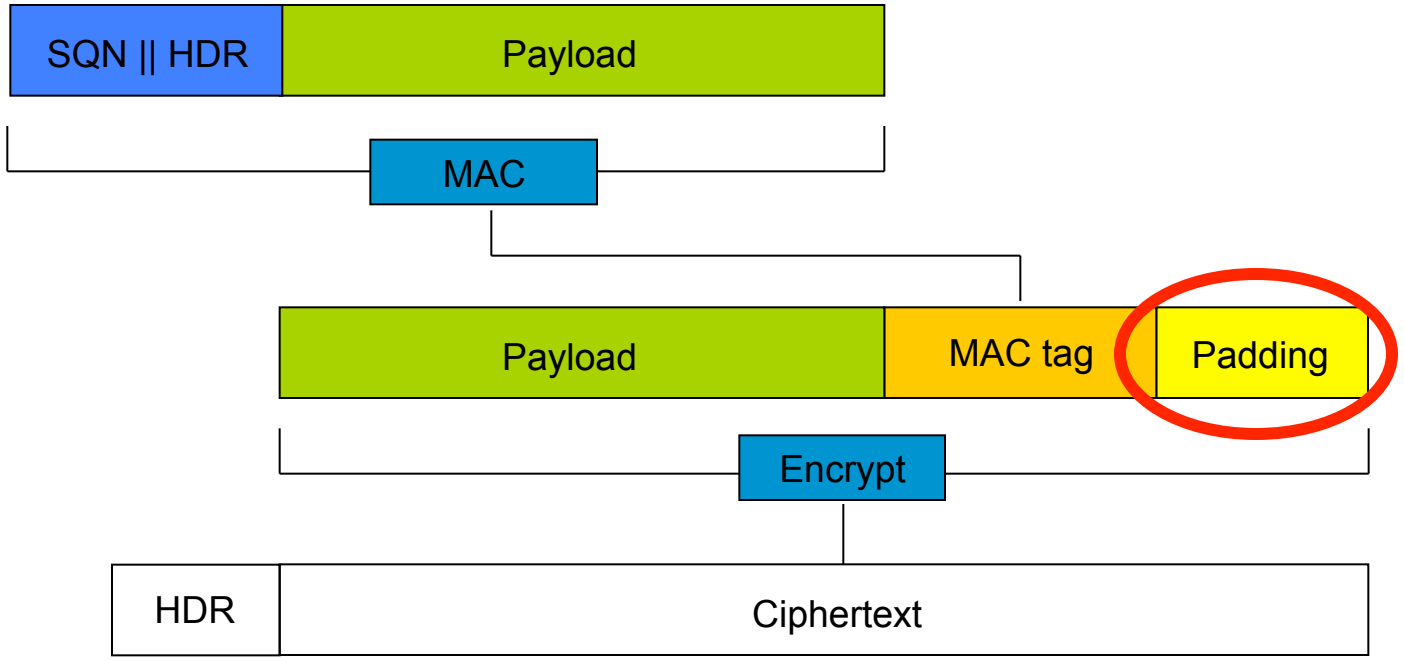
- JavaScript can also initiate its own TLS sessions to remote host.
  - Browser will then automatically inject HTTP cookies into TLS session on behalf of malware.
  - No SOP bypass needed.
  - Enables multi-session attacks targeting HTTP cookies.
  - More later!

The BEAST – Countermeasures

- Switch to using TLS 1.1 or 1.2.
  - Uses random IVs, so attack prevented.

- For TLS 1.0 users, some hacks are available:
  - Use 1/n-1 record splitting.
    - Now implemented in most but not all browsers.
    - Safari (Apple): status unknown.
  - Send 0-length dummy record ahead of each real record.
    - Breaks some implementations.
  - Switch to using RC4.
    - As recommended by many expert commentators.

TLS Record Protocol: MAC-Encode-Encrypt



MAC	HMAC-MD5, HMAC-SHA1, HMAC-SHA256
Encrypt	CBC-AES128, CBC-AES256, CBC-3DES, RC4-128
Padding	“00” or “01 01” or “02 02 02” or …. or “FF FF….FF”

TLS Record Protocol Padding

- Padding in TLS 1.0 and up has a particular format:
  - Always add at least 1 byte of padding.
  - If *t* bytes are needed, then add *t* copies of the byte representation of *t-1*.
  - So possible padding patterns in TLS are:

00;

01 01;

02 02 02;

⋮

TLS Record Protocol Padding

- Variable length padding is permitted in all versions of TLS.

- Up to 256 bytes of padding in total:

  FF FF…. FF

- From TLS 1.0:

  Lengths longer than necessary might be desirable to frustrate attacks on a protocol based on analysis of the lengths of exchanged messages.

Handling Padding During Decryption

- TLS 1.0 error alert:

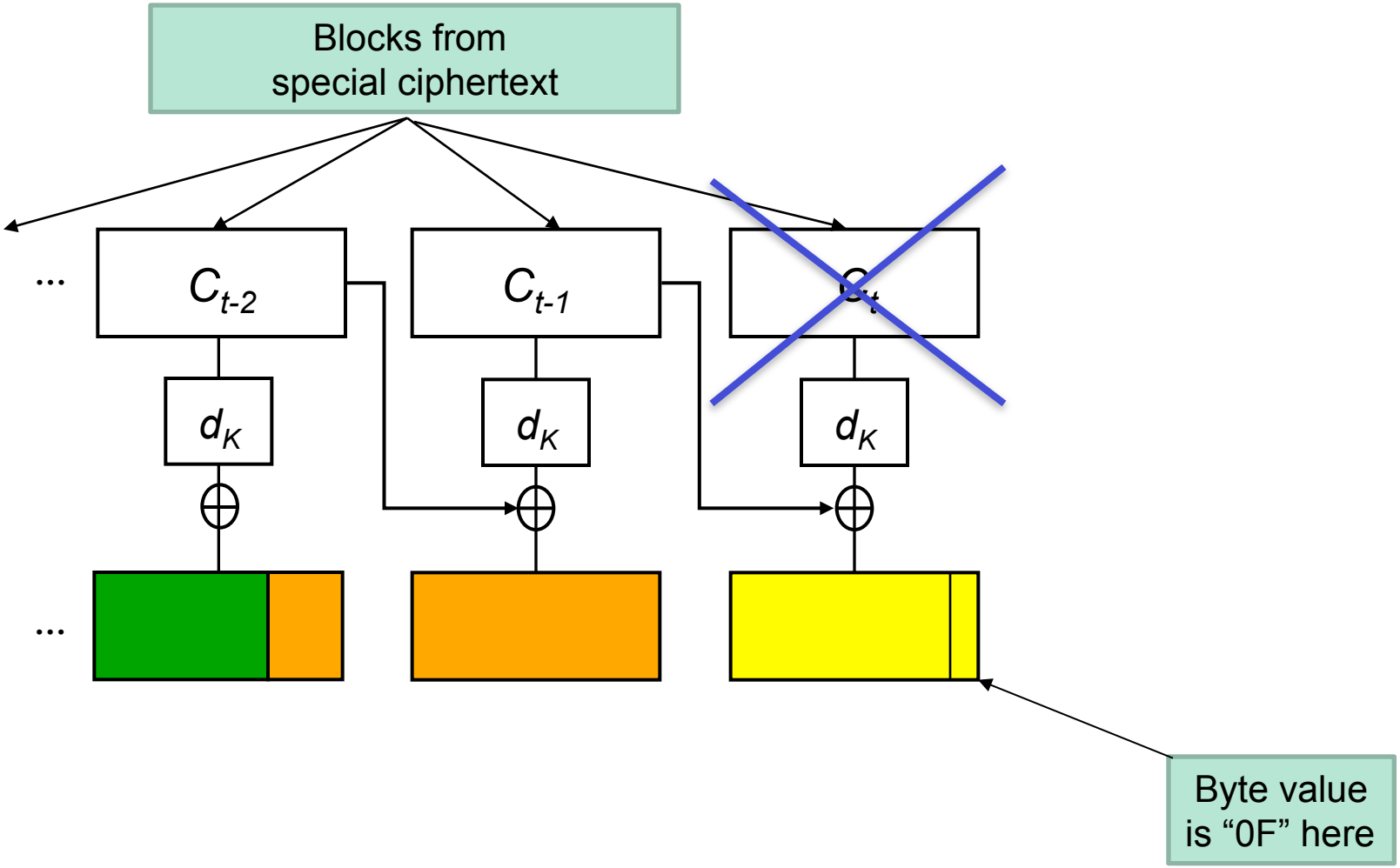
  decryption_failed: A TLSCiphertext decrypted in an invalid way: either it wasn`t an even multiple of the block length or its padding values, when checked, weren't correct. This message is always fatal.

- Suggests padding format should be checked, but without specifying exactly what checks should be done.

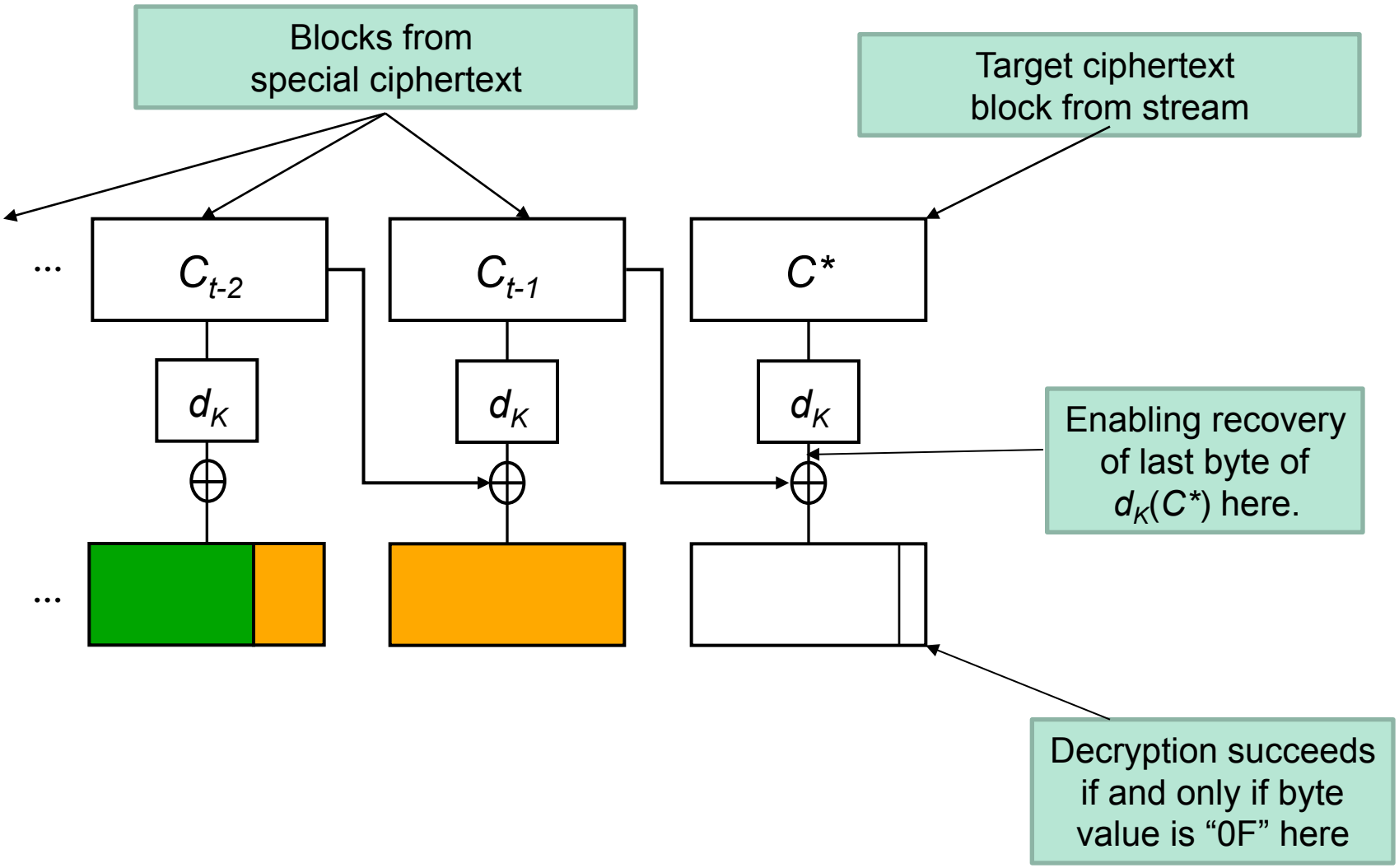
Preventing Weak Padding Checks

- Decryption sequence:
  - CBC mode decrypt, remove padding, check MAC.
- [M02]: failure to check padding format leads to a simple attack recovering the last byte of plaintext from any block.
- Assumptions:
  - Attacker has a TLS ciphertext containing a complete block of padding.
  - So MAC ends on block boundary for this ciphertext.
  - Padding removed by inspecting last byte only.

Moeller Attack for TLS



Moeller Attack for TLS



Preventing Weak Padding Checks

- Decryption succeeds if and only if:

$$(C_{t-1})_{15} \oplus (d_K(C^*))_{15} = \text{“0F”}$$

- Hence attacker can recover last byte of $d_K(C^*)$ with probability 1/256.
- This enables recovery of last byte of original plaintext P^* corresponding to C^* in the CBC stream.

- Hence, in TLS 1.1 and up:

Each uint8 in the padding data vector MUST be filled with the padding length value. The receiver MUST check this padding….

Full Padding Check

- Suppose that TLS does a *full* padding check.
- So decryption checks that bytes at the end of the plaintext have one of the following formats:

00;

01, 01;

02, 02, 02;

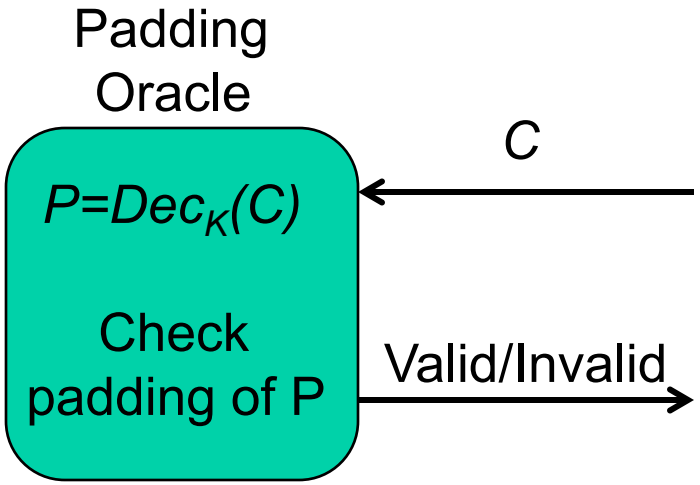
⋮

FF, FF,……..FF

and outputs an error if *none* of these formats is found.

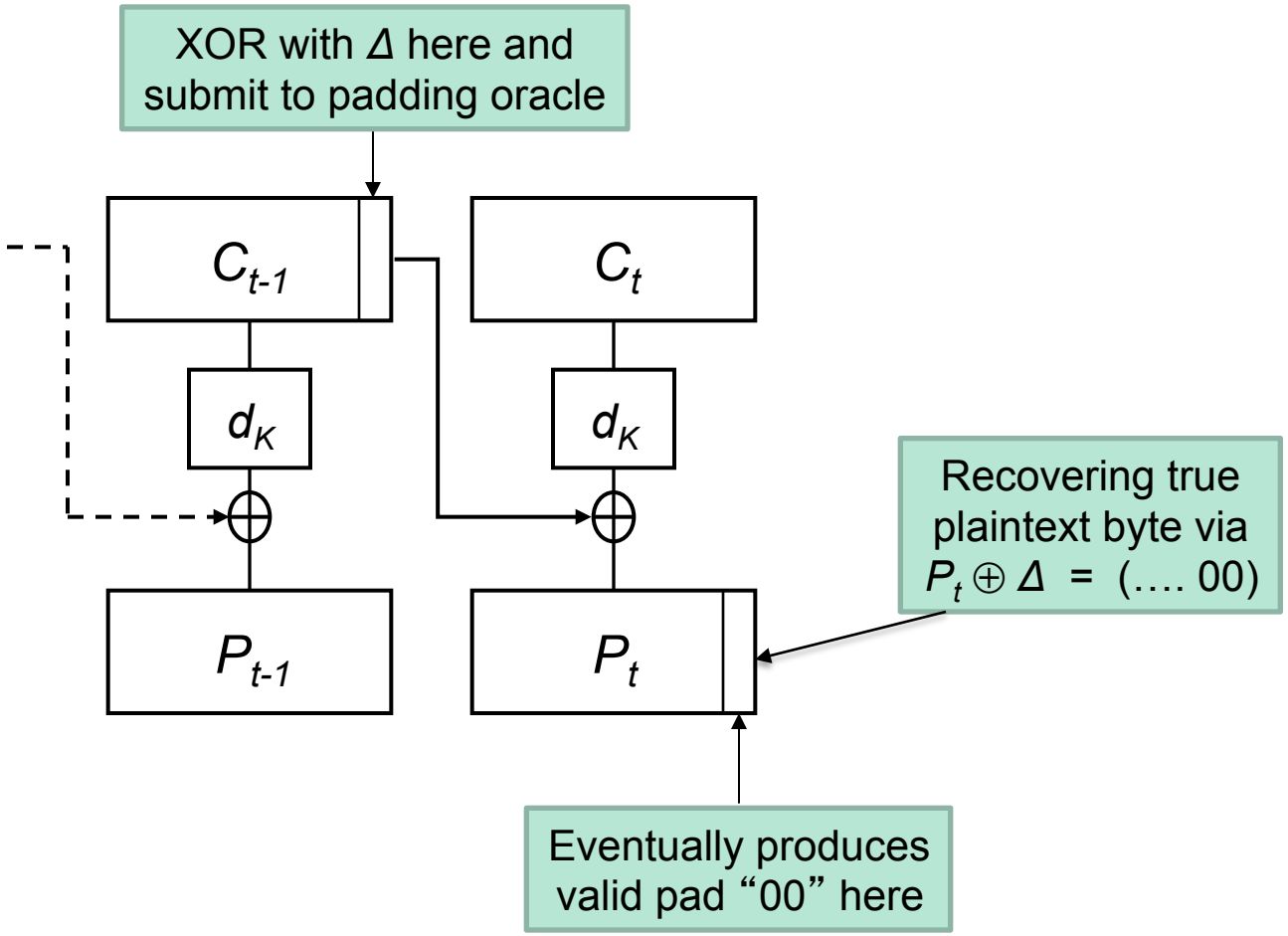
Padding Oracles

- Vaudenay [V02] proposed the concept of a *padding oracle.*

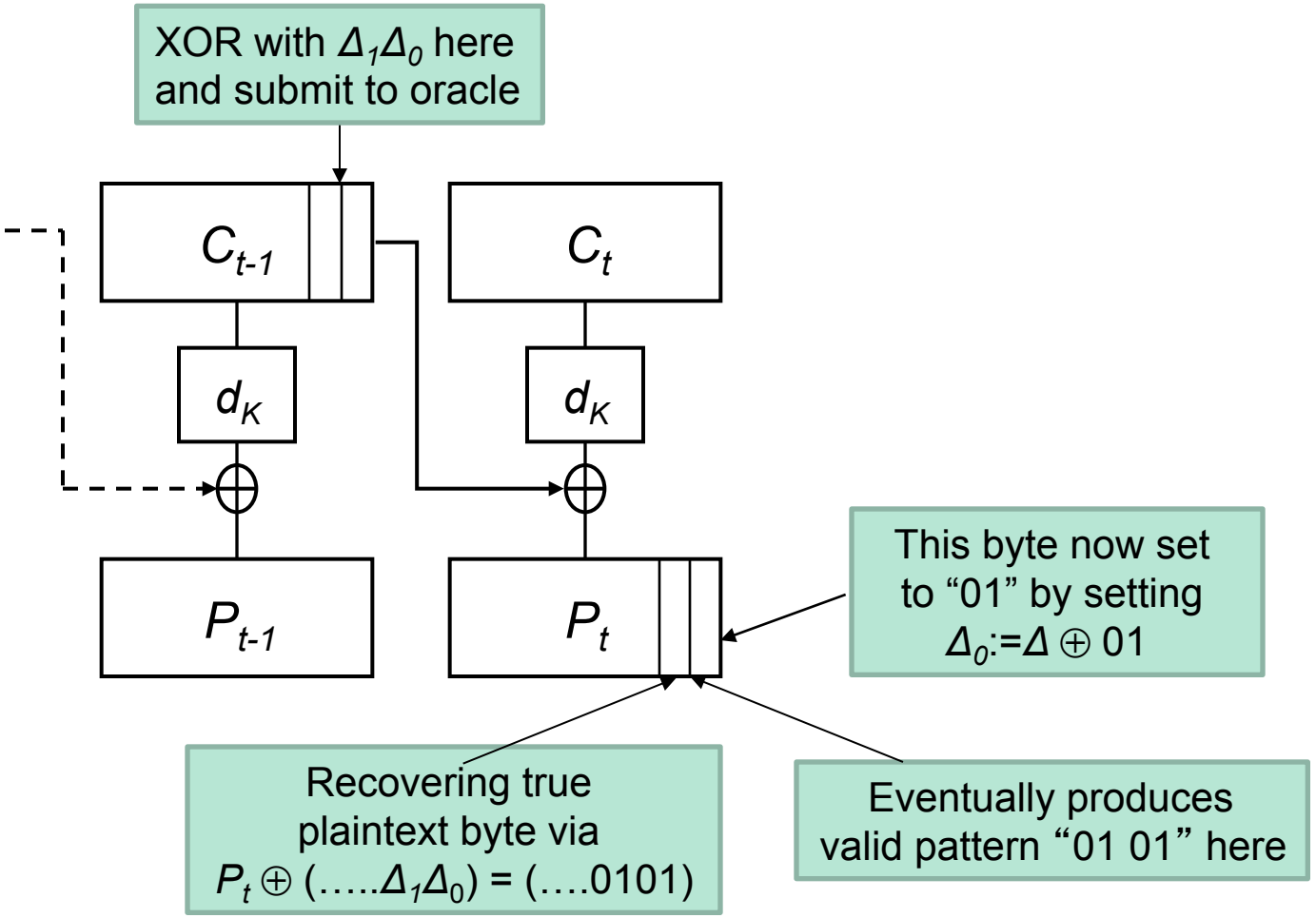


- Vaudenay showed that, for CBC mode and for certain padding schemes, a padding oracle can be used to build a decryption oracle!

Padding Oracle Attack for TLS Padding



Padding Oracle Attack for TLS Padding



Padding Oracle Attack for TLS Padding

- An average of 128 trials are needed to extract the last byte of each plaintext block.
- Can extend to the entire block, with an average of 128 trials per byte.
- Can extend to entire ciphertext.
  - Because attacker can place *any* target block as last block of ciphertext.

TLS Padding Oracles In Practice?

- In TLS, an error message during decryption can arise from either a failure of the padding check or a MAC failure.

- Vaudenay's padding oracle attack will produce an error of one type or the other.
  - Padding failure indicates *incorrect* padding.
  - MAC failure indicates *correct* padding.

- If these errors are *distinguishable*, then a padding oracle attack should be possible.

TLS Padding Oracles In Practice?

Good news (for the attacker):

- The error messages arising in TLS 1.0 *are* different:
  - `bad_record_mac`
  - `decryption_failed`

Bad news:

- But the error messages are encrypted, so cannot be seen by the attacker.
- And an error of either type is *fatal*, leading to immediate termination of the TLS session.

TLS Padding Oracles In Practice?

Canvel *et al.* [CHVV03] :

- A MAC failure error message is likely to appear on the network **later** than a padding failure error message.
  - Because the MAC is only checked if the padding is good.

- So *timing* the appearance of error messages can give us the required padding oracle.
  - Even if the error messages are encrypted.

- **But** the errors are fatal, so the attacker can still only learn one byte of plaintext, and then with probability only 1/256.

OpenSSL and Padding Oracles

Canvel *et al.* [CHVV03]:

- The attacker can still decrypt reliably if a *fixed* plaintext is repeated in a *fixed* location across many TLS sessions.
  - e.g. password in login protocol or session cookie.
  - A multi-session attack.
  - Modern viewpoint: use BEAST-style malware.

- OpenSSL had a detectable timing difference.
  - Roughly 2ms difference for long messages.
  - Enabling recovery of TLS-protected Outlook passwords in about 3 hours.

Summary: TLS and Padding Oracles

[V02,CHVV03]:

- Specifics of TLS padding format can be exploited to mount a plaintext recovery attack.
- No chosen-plaintext requirement.
- The attack depends on being able to distinguish *good* from *bad* padding.
  - In practice, this is done via a timing side-channel.
  - The MAC is only checked if padding good, and the MAC is always bad in the attack.
  - Distinguish cases by timing TLS error messages.

Countermeasures?

- Redesign TLS:
  - Pad-MAC-Encrypt or Pad-Encrypt-MAC.
  - Too invasive, did not happen.
- Switch to RC4.
- Or add a fix to ensure *uniform errors*?
  - If attacker can't tell difference between MAC and pad errors, then maybe TLS's MEE construction is secure?
  - So how should TLS implementations ensure uniform errors?

Ensuring Uniform Errors

From the TLS 1.2 specification:

…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.

Compute the MAC on what though?

Ensuring Uniform Errors

For instance, if the pad appears to be incorrect, the implementation might assume a zero-length pad and then compute the MAC.

- This approach is adopted in many implementations, including OpenSSL, NSS (Chrome, Firefox), BouncyCastle, OpenJDK, …

- One alternative (GnuTLS and others) is to remove as many bytes as are indicated by the last byte of plaintext and compute the MAC on what's left.

Ensuring Uniform Errors

… 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.

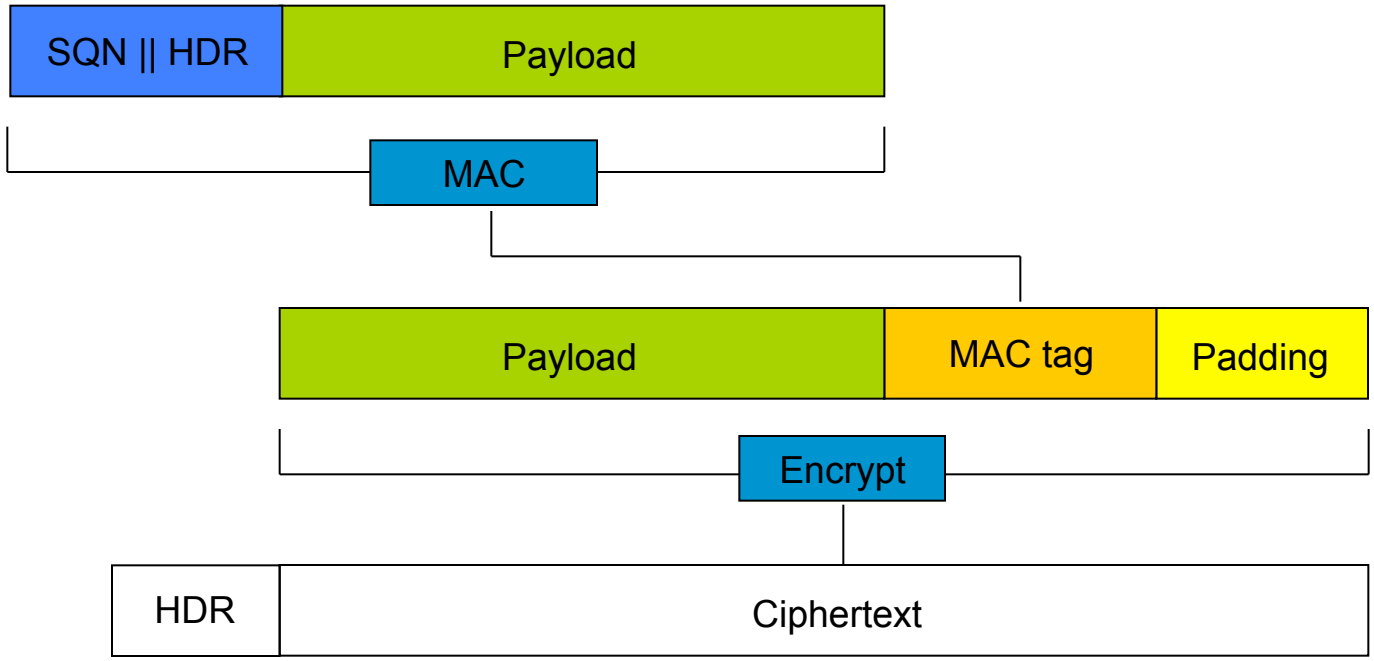
Ensuring Uniform Errors

… 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.

Lucky 13 [AP13]

- Distinguishing attacks and full plaintext recovery attacks against TLS-CBC implementations following the advice in the TLS 1.2 spec.
  - And variant attacks against those that do not.
- Applies to all versions of SSL/TLS.
  - SSLv3.0, TLS 1.0, 1.1, 1.2.
  - And DTLS.
- Demonstrated in the lab against OpenSSL and GnuTLS.

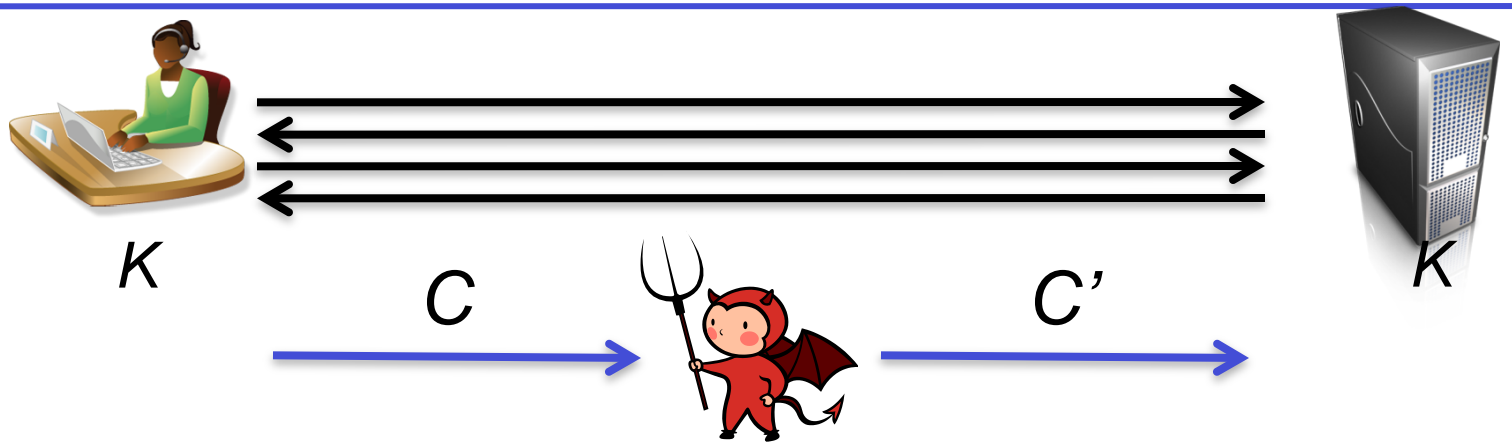
Reminder: MAC-Encode-Encrypt in TLS



Lucky 13 – Basic Idea

- TLS decryption removes padding and MAC tag to get PAYLOAD.
- HMAC computed on SQN || HDR || PAYLOAD.
- HMAC computation involves iteration of hash compression function, e.g. MD5, SHA-1, SHA-256.
- Running time of HMAC depends on *L*, the byte length of SQN || HDR || PAYLOAD:
  - $L \leq 55$ bytes: 4 compression functions;
  - $56 \leq L \leq 119$: 5 compression functions;
  - $120 \leq L \leq 183$: 6 compression functions;
  - ….

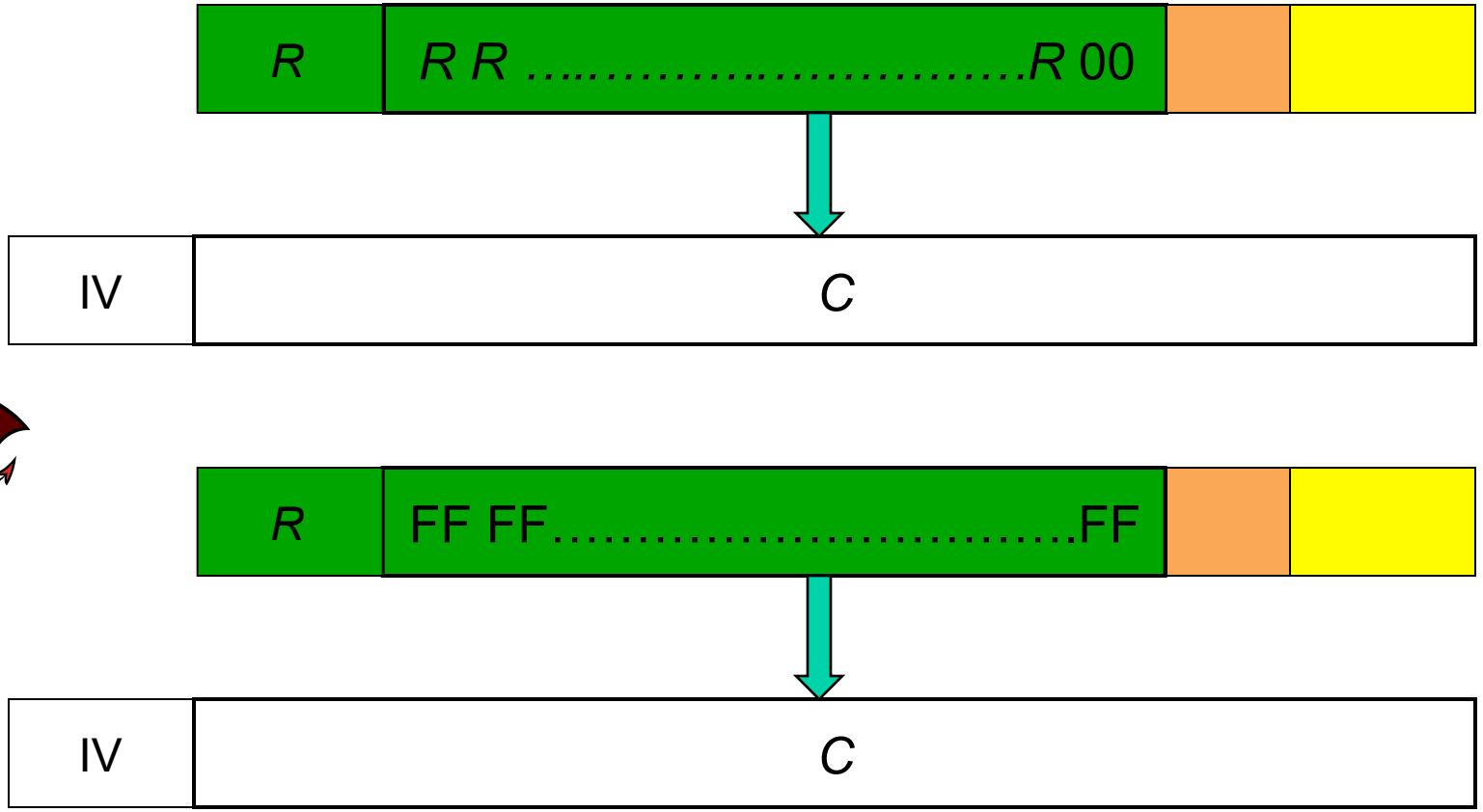
Lucky 13 Distinguishing Attack



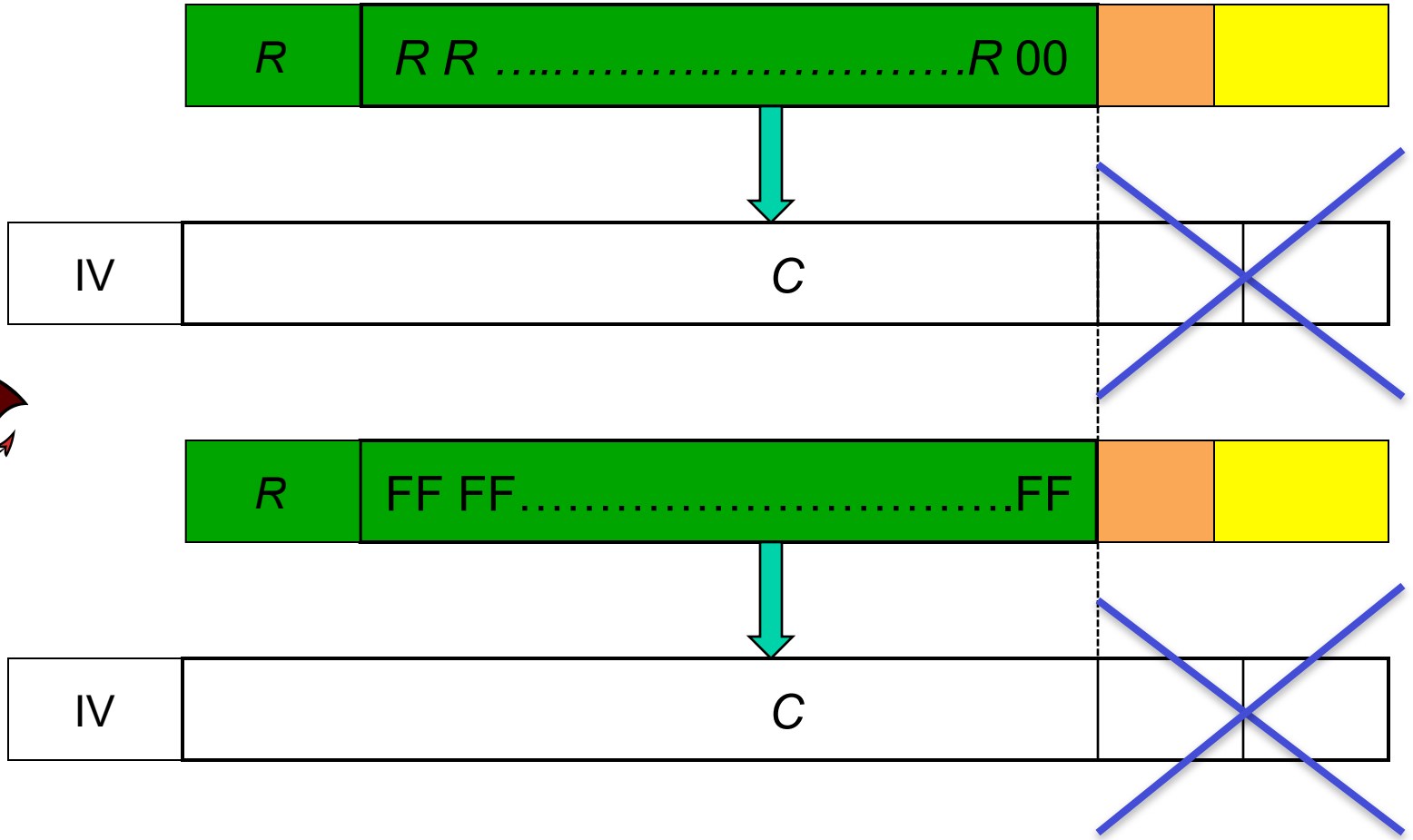
$C = \mathrm{Enc}_K(M)$ M is either $R^{287} \| 00$ or $R^{32} \| \mathrm{FF}^{256}$

- Adversary intercepts C, mauls, and forwards on to recipient.
- Time taken to respond with error message will indicate whether $M = R^{287} \| 00$ or $R^{32} \| \mathrm{FF}^{256}$.
- Ciphertext-only distinguishing attack.

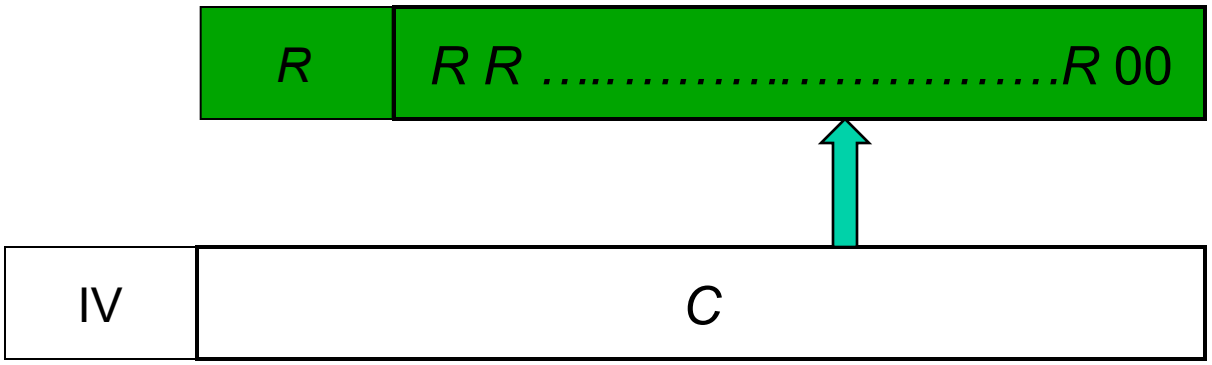
Lucky 13 Distinguishing Attack – Choose



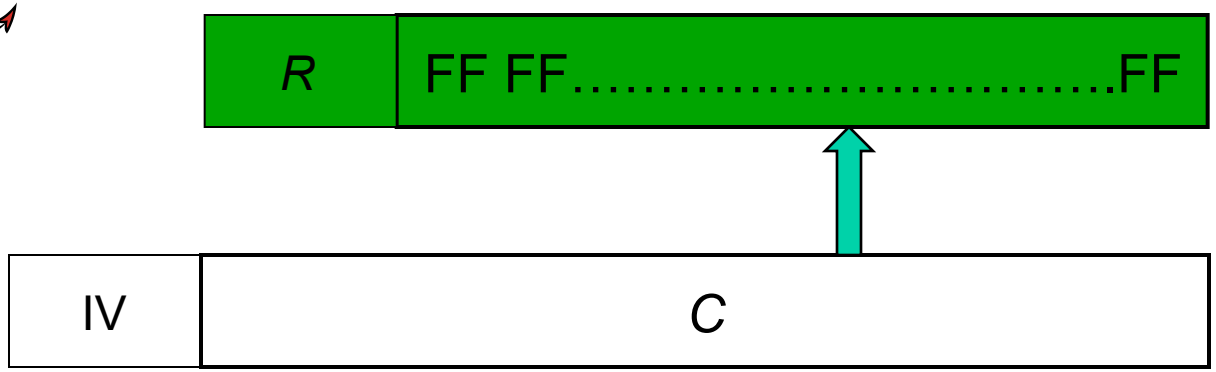
Lucky 13 Distinguishing Attack – Maul



Lucky 13 Distinguishing Attack – Inject

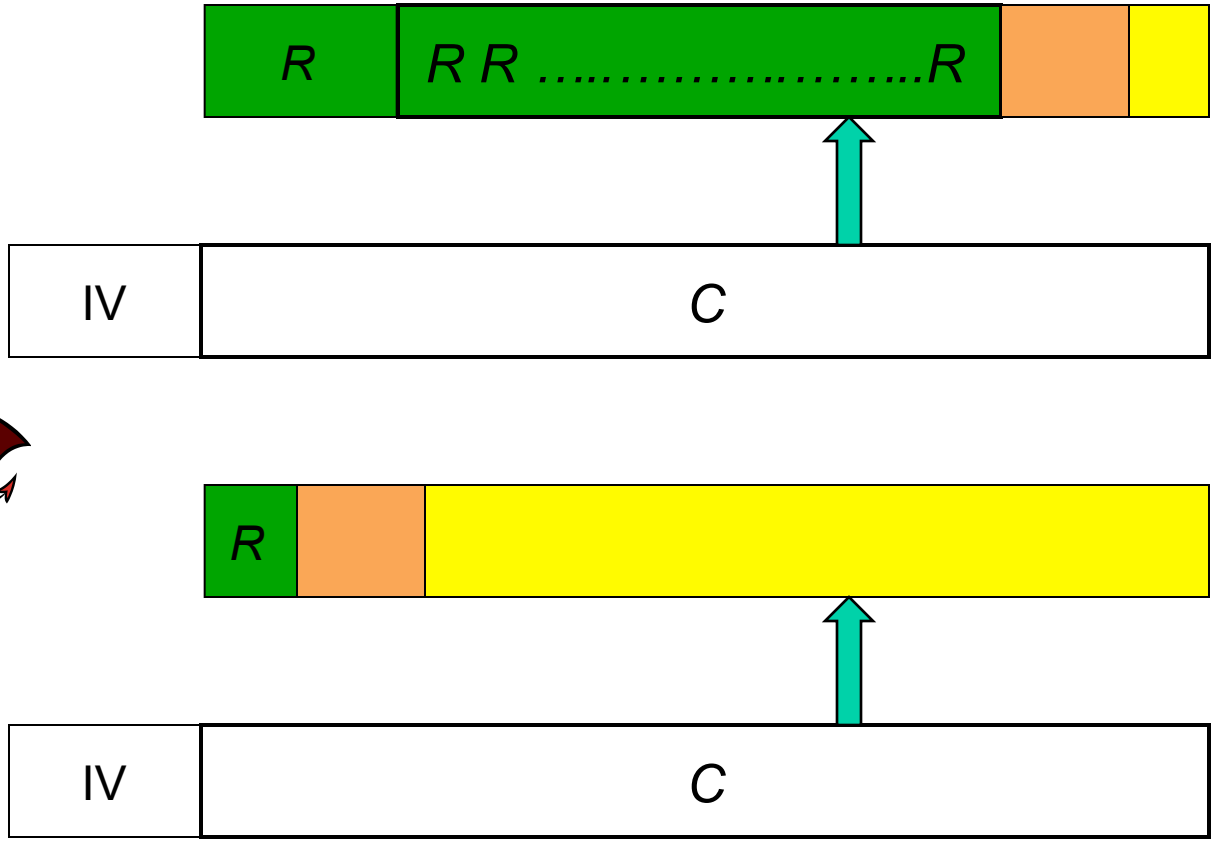


1-byte valid padding
20-byte MAC
267-byte message

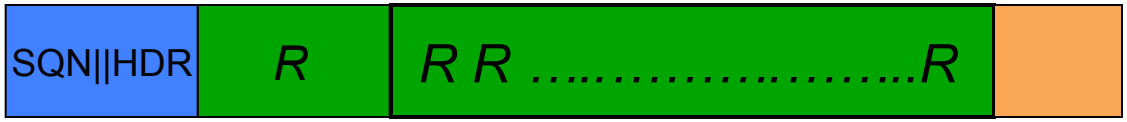


256-byte valid padding
20-byte MAC
12-byte message

Lucky 13 Distinguishing Attack – Decrypt

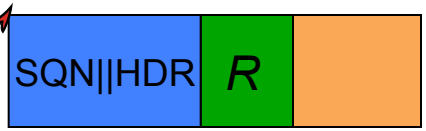


Lucky 13 Distinguishing Attack – Decrypt



280 bytes

Slow MAC verification

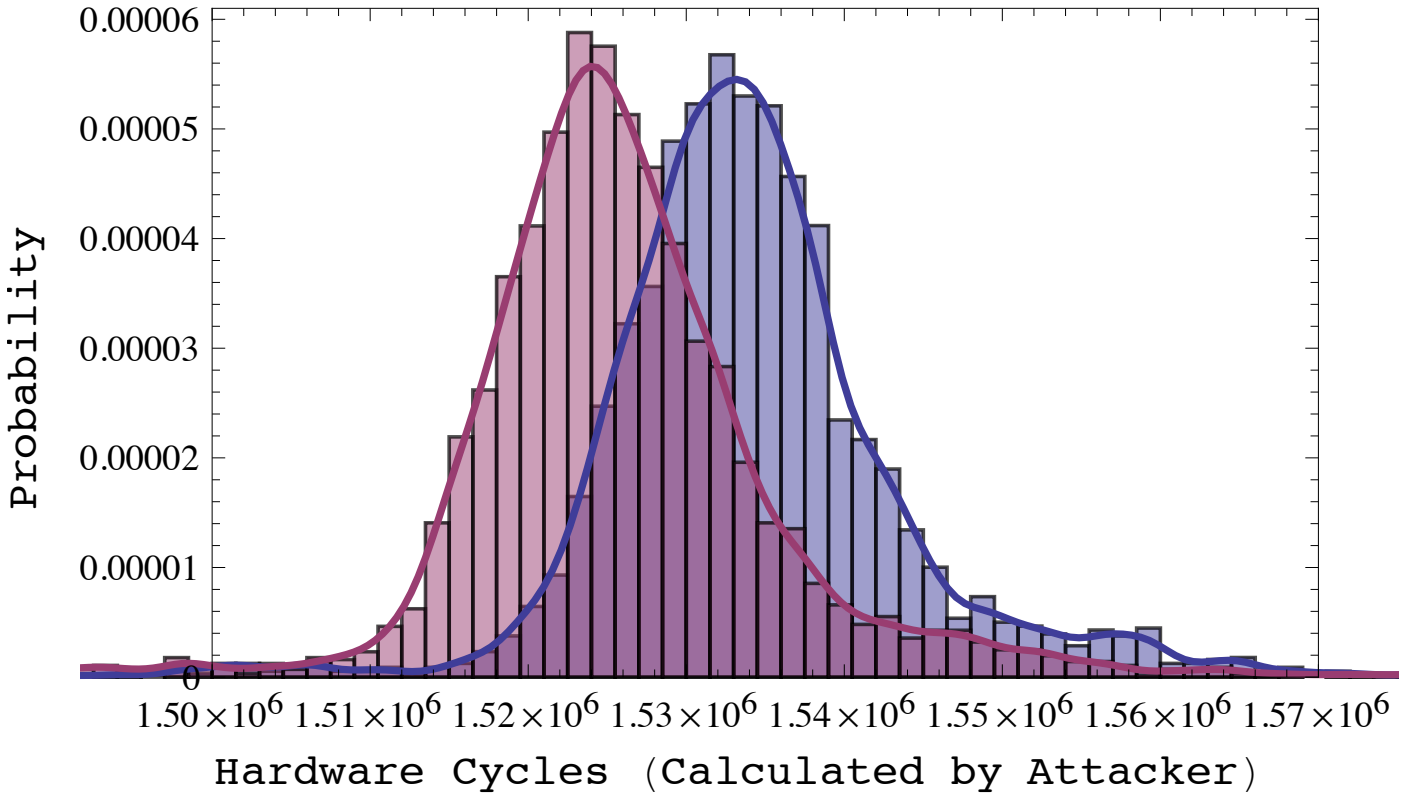


25 bytes

Fast MAC verification

Timing difference: 4 SHA-1 compression function evaluations

Experimental Results for Distinguishing Attack

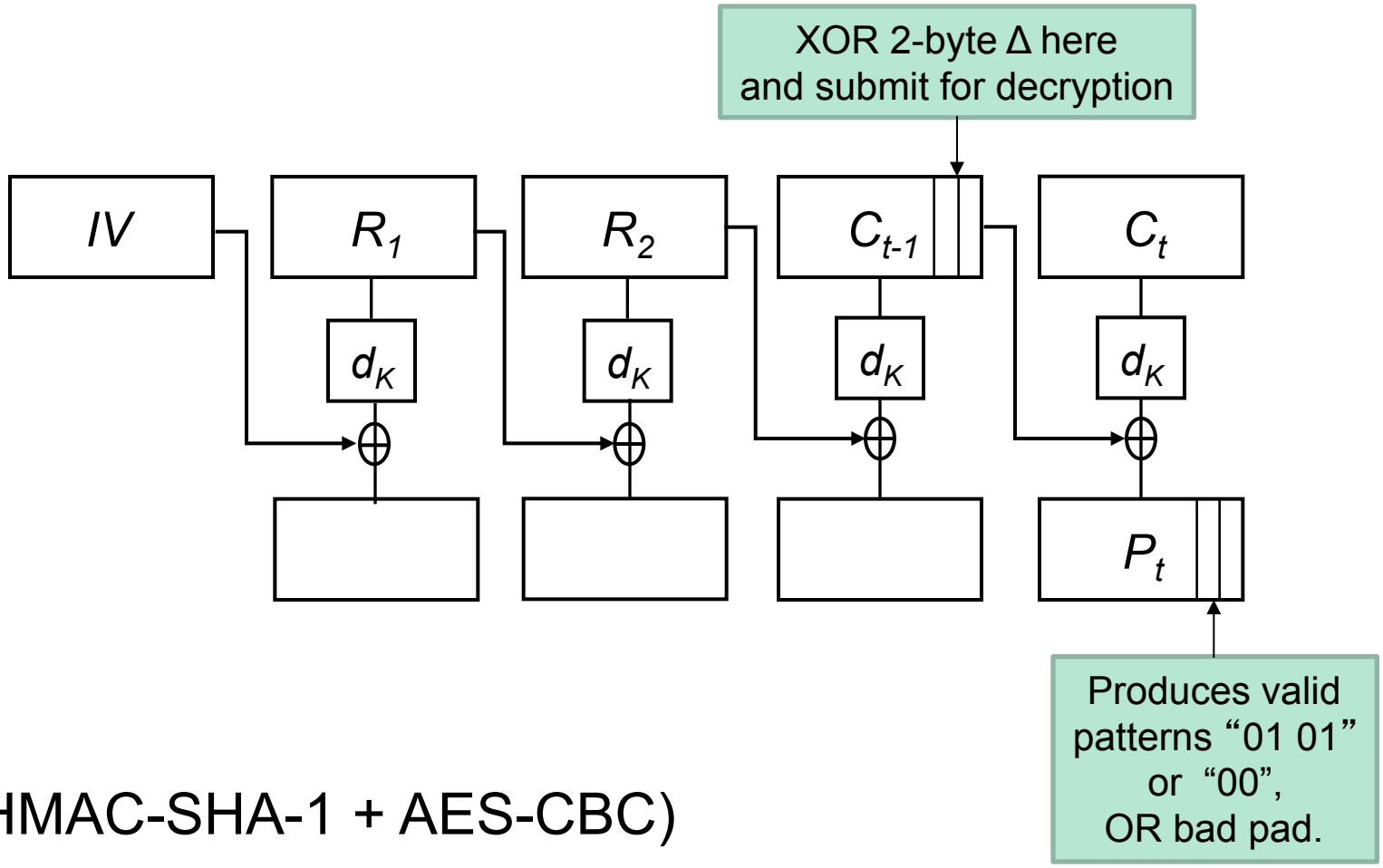


- OpenSSLv1.0.1 on server running at 1.87Ghz.
- 100 Mbit LAN.
- Difference in means is circa 3.2 μs.

Success Probability

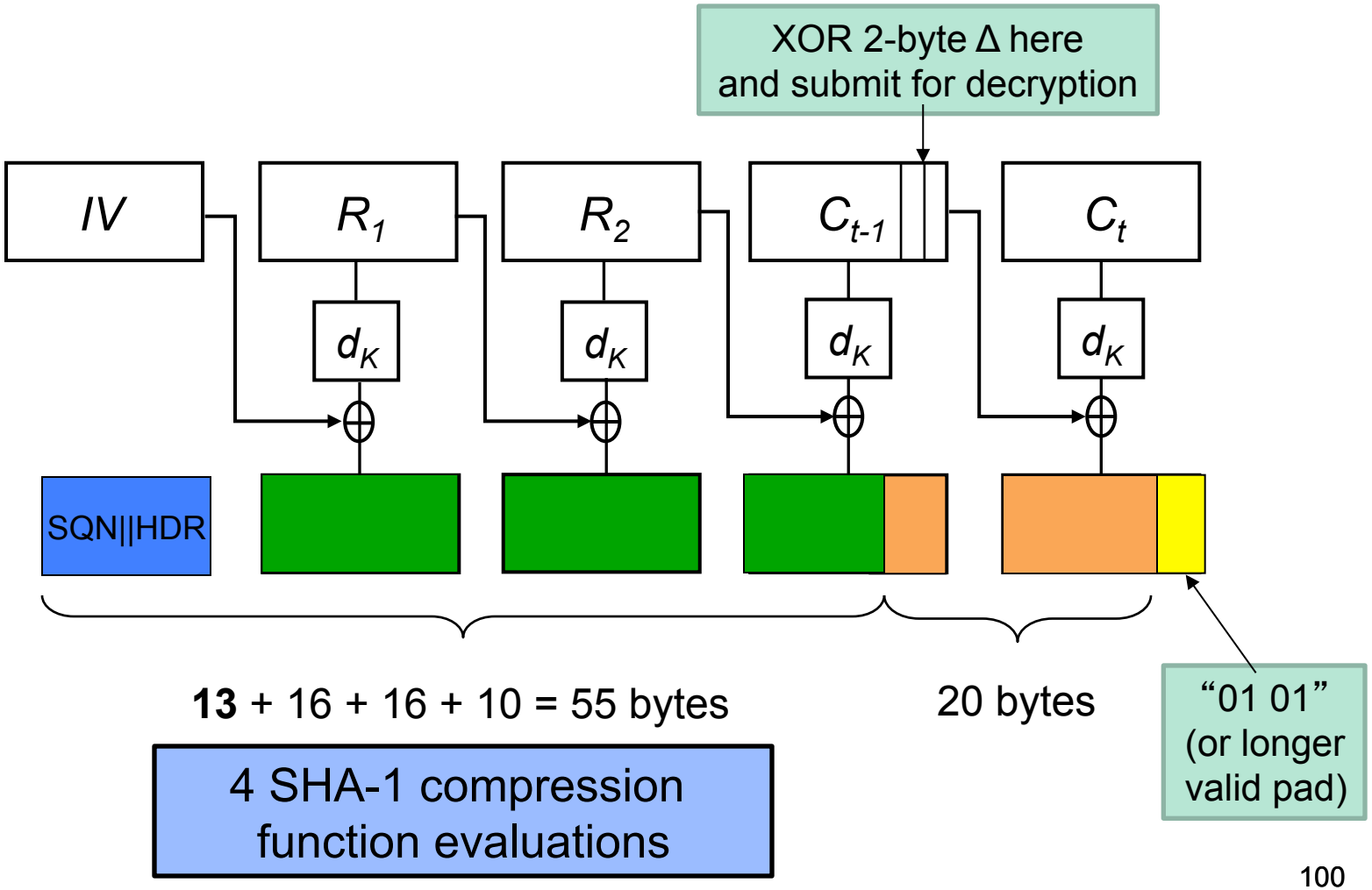
Number of Sessions	Success Probability
1	0.756
4	0.858
16	0.951
64	0.992
128	1

Lucky 13 – Plaintext Recovery

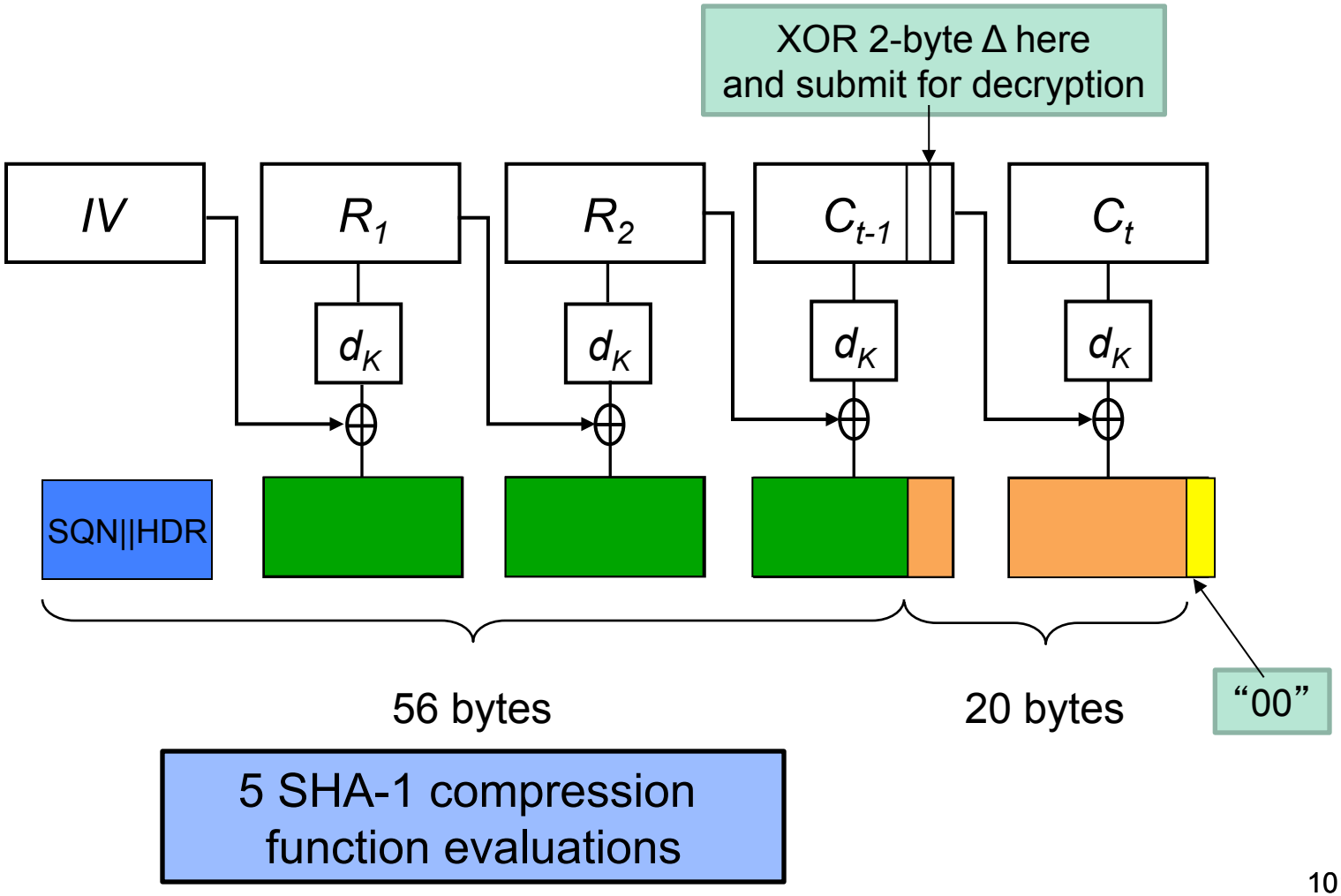


(HMAC-SHA-1 + AES-CBC)

Case: “01 01” (or longer valid pad)

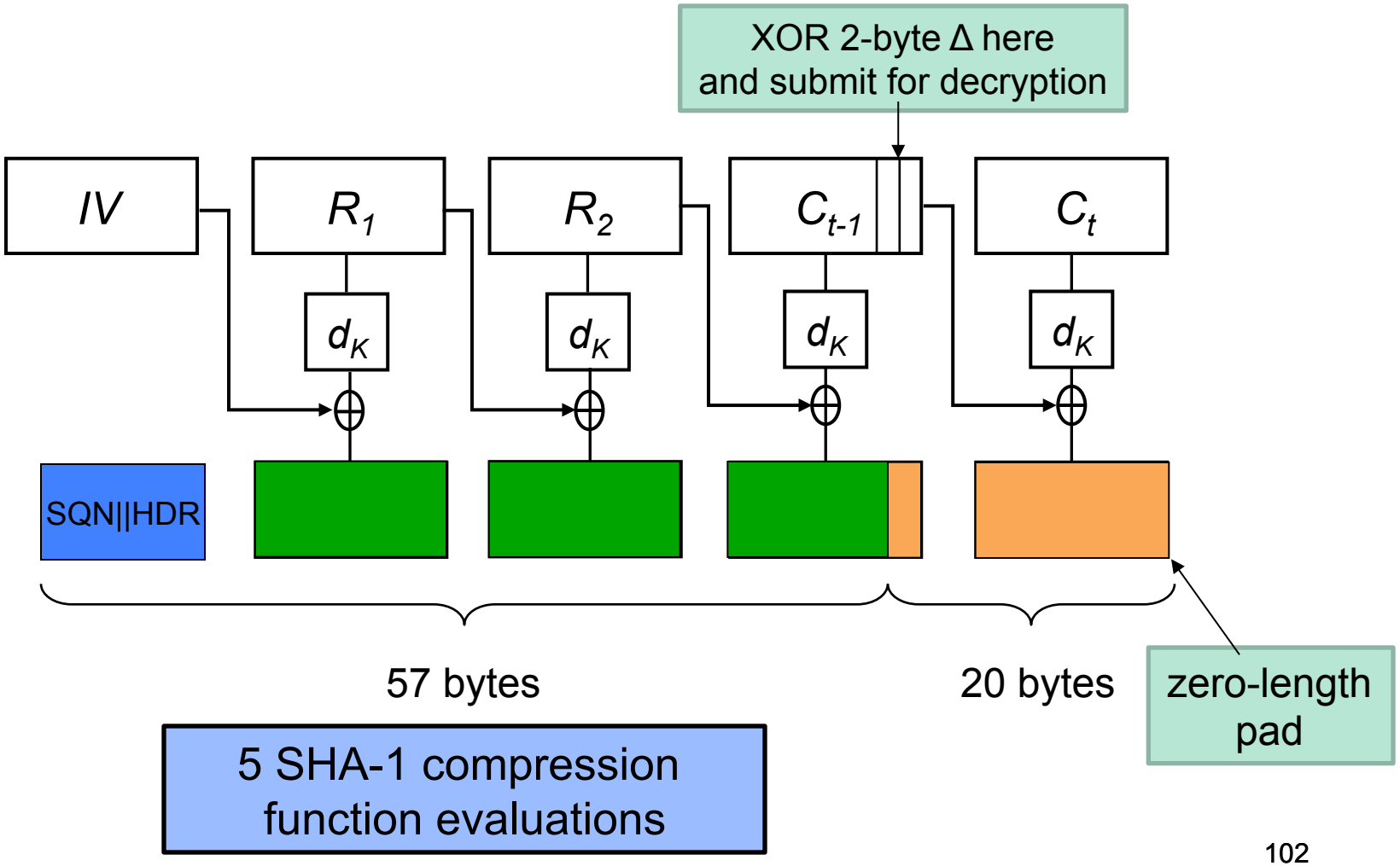


Case: “00”



5 SHA-1 compression function evaluations

Case: Bad padding



Lucky 13 – Plaintext Recovery

- The injected ciphertext causes bad padding and/or a bad MAC.
  - This leads to a TLS error message, which the attacker times.
- There is a timing *difference* between “01 01” case and the other 2 cases.
  - A single SHA-1 compression function evaluation.
  - Roughly 1000 clock cycles, 1μs range on typical processor.
  - Measurable difference on same host, LAN, or a few hops away.
- Detecting the “01 01” case allows last 2 plaintext bytes in the *target* block C_t to be recovered.
  - Using the standard CBC algebra.
  - Attack then extends easily to all bytes.

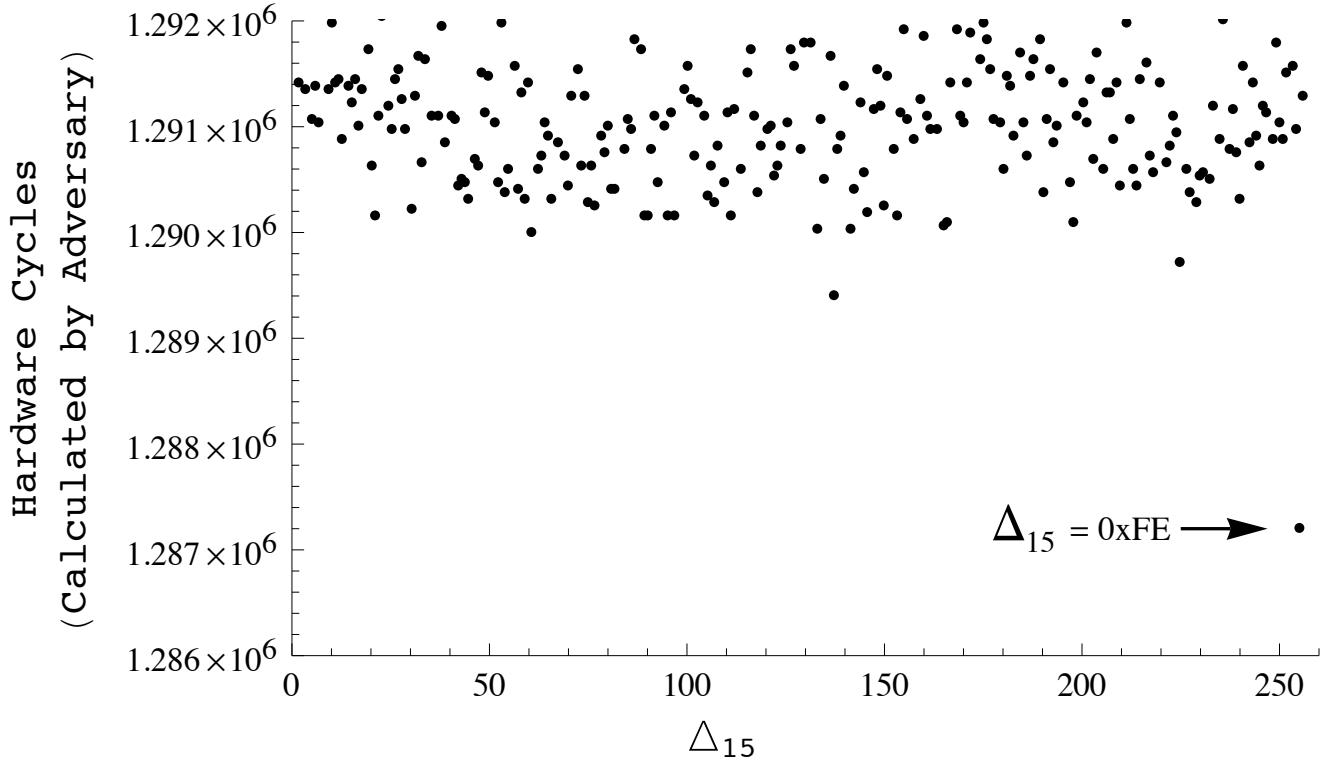
Lucky 13 – Attack Cost

- We need 2^{16} attempts to try all 2-byte Δ values.
- And we need around 2^{7} trials for each Δ value to reliably distinguish the different events.
  - Noise level depends on experimental set-up.
- Each trial kills the TLS session.
- Hence the headline attack cost is 2^{23} sessions, all encrypting the same plaintext.
- Looks rather theoretical?

Lucky 13 – Improvements

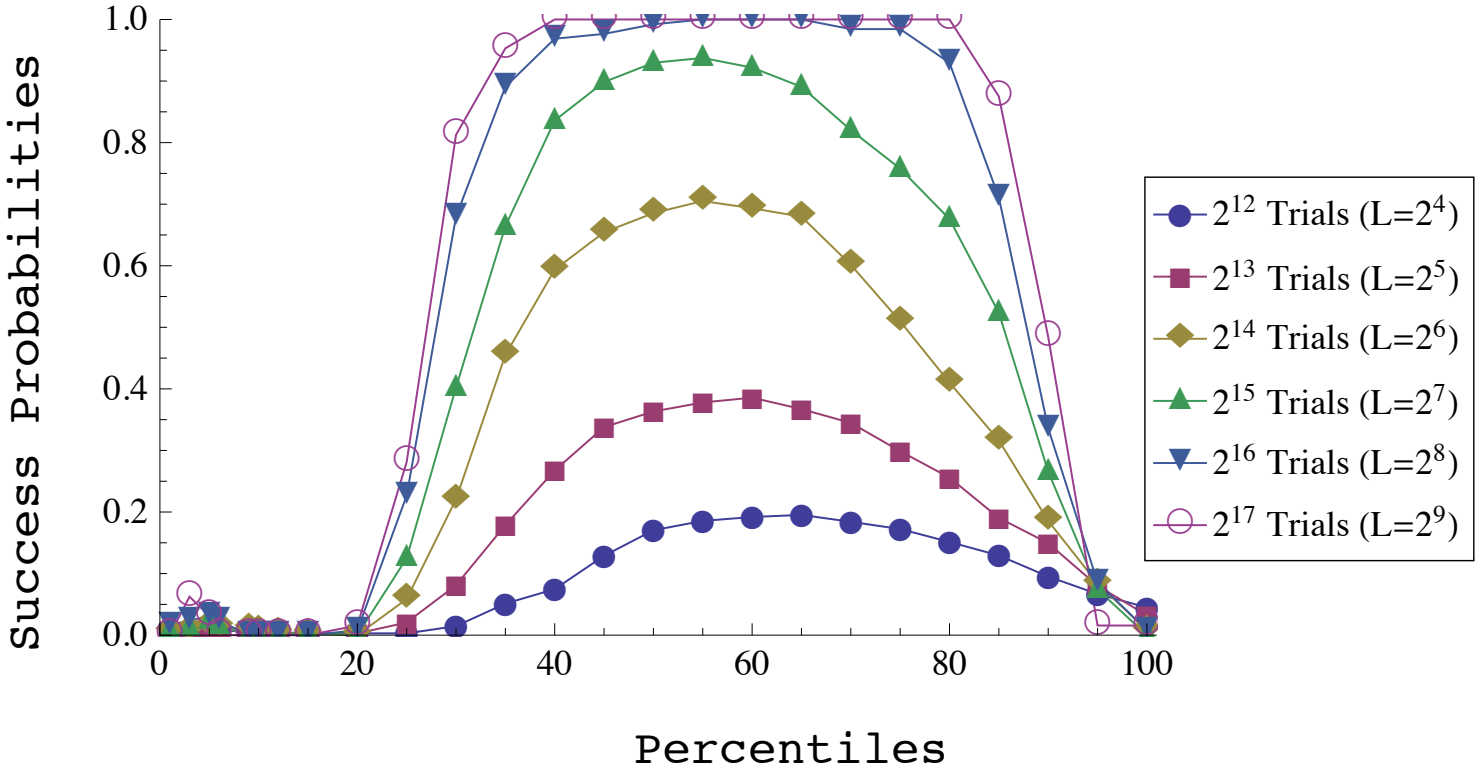
- If 1-out-of-2 last bytes known, then we only need 2^8 attempts per byte.
- If the plaintext is `base64` encoded, then we only need 2^6 attempts per byte.
  - And 2^7 trials per attempt to de-noise, for a total of 2^{13}.
- BEAST-style attack targeting HTTP cookies.
  - Malicious client-side Javascript makes HTTP GET requests.
  - TLS sessions are automatically generated and HTTP cookies attached.
  - Pad GET requests so that 1-out-of-2 condition always holds.
  - Cost of attack is 2^{13} GET requests per byte of cookie.
  - Now a practical attack!

Experimental Results



- Byte 14 of plaintext set to 01; byte 15 set to FF.
- Modify Δ in position 15.
- OpenSSLv1.0.1 on server running at 1.87Ghz, 100 Mbit LAN.
- Median times (noise not shown).

Experimental Results



OpenSSL: recovering last byte in a block, using percentile test to extract correct byte value, no assumptions on plaintext.

Lucky 13 – Further Extensions

- The attack extends to other MAC algorithms.
  - Nice interplay between block-size, MAC tag size and 13-byte field SQN || HDR.
- The attack extends to other methods for dealing with bad padding.
  - e.g. as in GnuTLS, faster but partial plaintext recovery.
- [The attack can be applied to DTLS.
  - No error messages, but simulate these via DTLS Heartbeats.
  - Errors non-fatal, so can execute attack in a single session.
  - Cam amplify timing differences using AlFardan-Paterson techniques (NDSS 2012).]

Lucky 13 – Impact

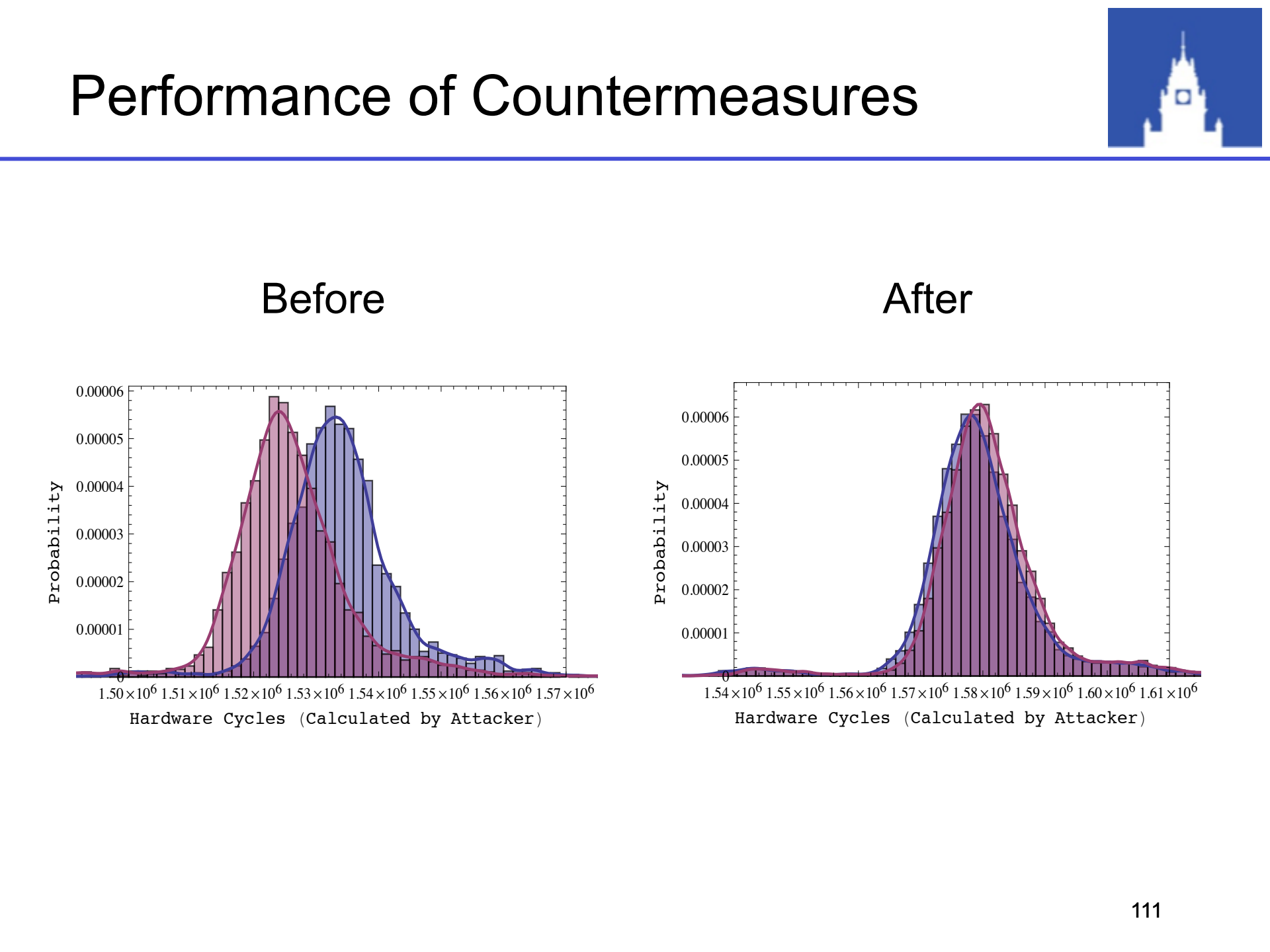
- **OpenSSL** patched in versions 1.0.1d, 1.0.0k and 0.9.8y, released 05/02/2013.
- **NSS** (Firefox, Chrome) patched in version 3.14.3, released 15/02/2013.
- **Opera** patched in version 12.13, released 30/01/2013
- **Oracle** released a special critical patch update of JavaSE, 19/02/2013.
- **BouncyCastle** patched in version 1.48, 10/02/2013
- Also **GnuTLS**, **PolarSSL**, **CyaSSL**, **MatrixSSL**.
- **Microsoft** "determined that the issue had been adequately addressed in previous modifications to their TLS and DTLS implementation".
- **Apple:** status unknown.

(Full details at: www.isg.rhul.ac.uk/tls/lucky13.html)

Lucky 13 – Countermeasures

- We really need constant-time decryption for TLS-CBC.
- Add dummy hash compression function computations when padding is *good* to ensure total is the same as when padding is *bad*.
- Add dummy padding checks to ensure number of iterations done is independent of padding length and/or correctness of padding.
- Watch out for length sanity checks too.
  - Need to ensure there's enough space for *some* plaintext after removing padding and MAC, but without leaking any information about amount of padding removed.
- TL;DR: it's a bit of a nightmare.

Performance of Countermeasures



Before

After

Other Lucky 13 Countermeasures?

- Introduce random delays during decryption.
  - Surprisingly *in*effective, analysis in [AP13].
- Redesign TLS:
  - Pad-MAC-Encrypt or Pad-Encrypt-MAC.
  - Currently, some discussion on TLS mailing lists.
  - No easy deployment route, seems unlikely to happen.
- Switch to TLS 1.2
  - Has support for AES-GCM and AES-CCM.
- Use RC4.

RC4

- The RC4 stream cipher has long been known to have statistical weaknesses.
  - e.g. Mantin-Shamir bias, recent work of Maitra *et al.*, Isobe *et al.*
  - Most attention has been given to the initial few bytes of keystream.
  - The focus has been on finding and giving theoretical explanations for *individual* biases, and on key-recovery attacks.
- Usual countermeasure is to discard the initial bytes of keystream and use only “good” bytes.
- So what does RC4 in TLS do?

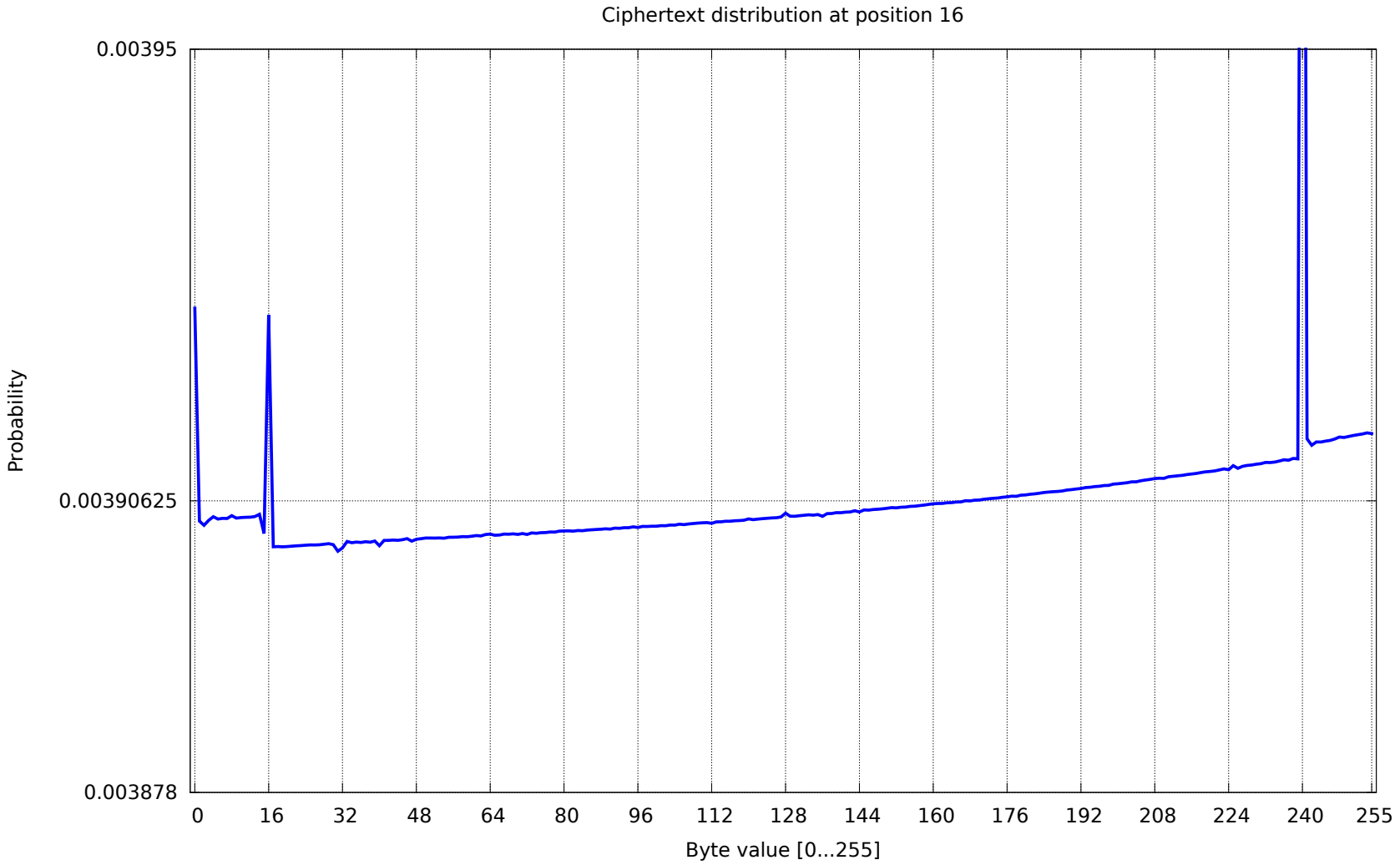
RC4 in TLS

- TLS does not discard initial keystream bytes!
  - Because it hurts performance too much;
  - The biases are small anyway; and only exist in the first few bytes.

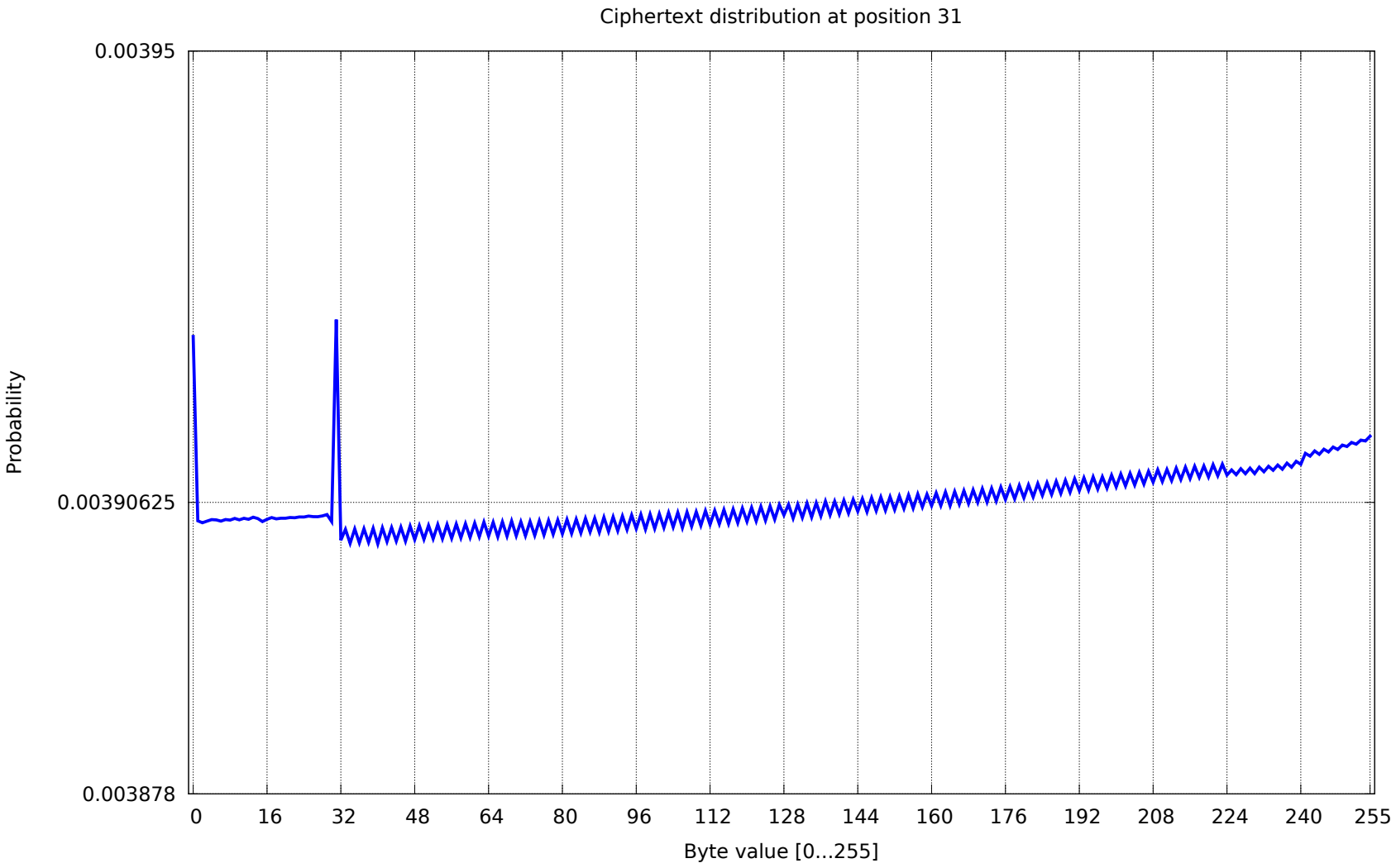
- [ABPPS13]: we estimated the biases in the first 256 output bytes by sampling RC4 keystreams for 2^{45} random 128-bit keys.

- We found many previously unreported biases of significant size…
  - www.isg.rhul.ac.uk/tls/biases.pdf

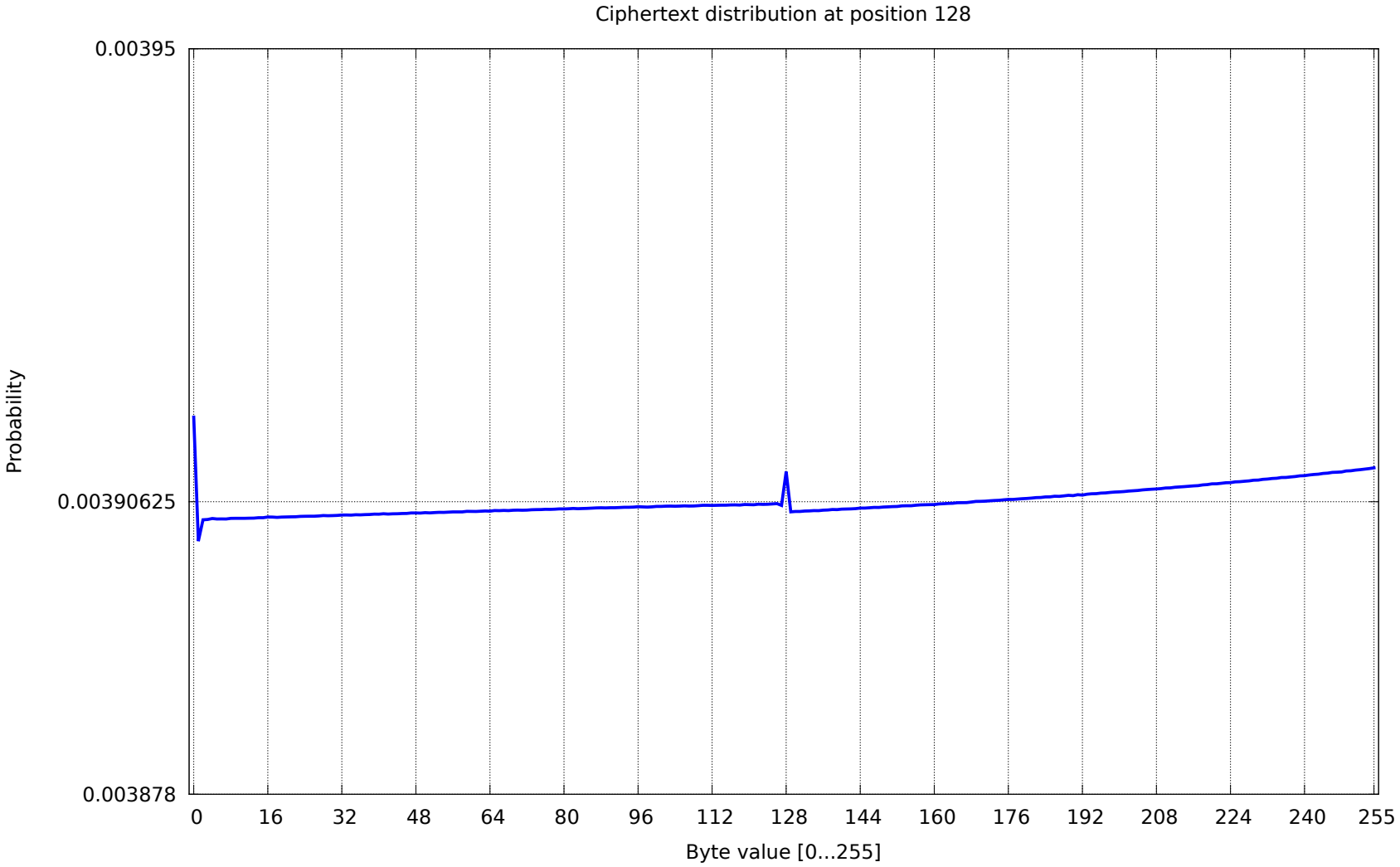
Biases in byte 16 of RC4 Output



Biases in byte 31 of RC4 Output



Biases in byte 128 of RC4 Output



TLS-RC4 Attack

- These biases are large enough to enable plaintext recovery attacks on first 256 bytes of TLS sessions.

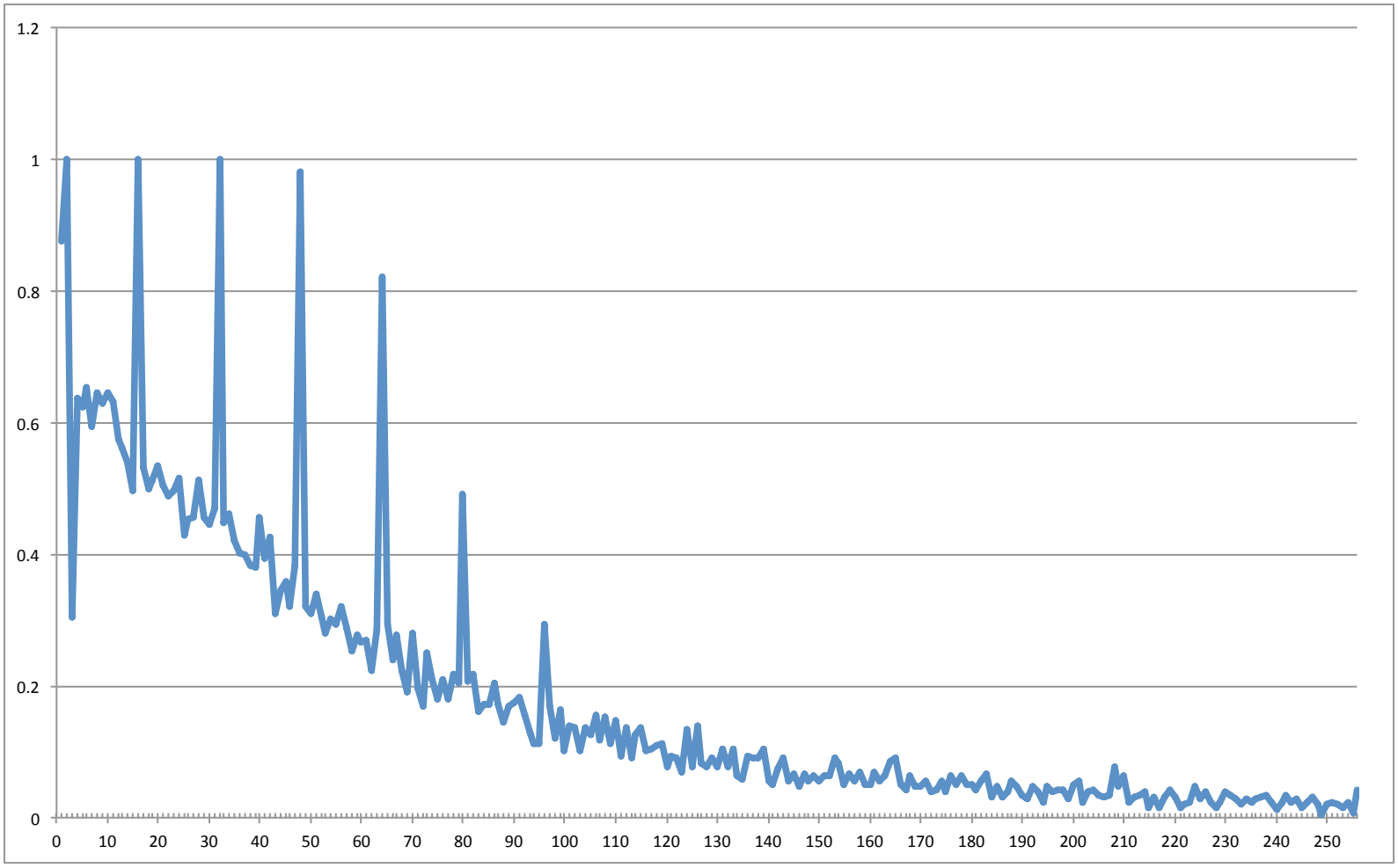
- Needs a multi-session attack.
  - BEAST-style malware as possible generation mechanism.

- Attack using Bayesian technique:
  - Gather many ciphertext samples C from different sessions.
  - For each byte position i we have:

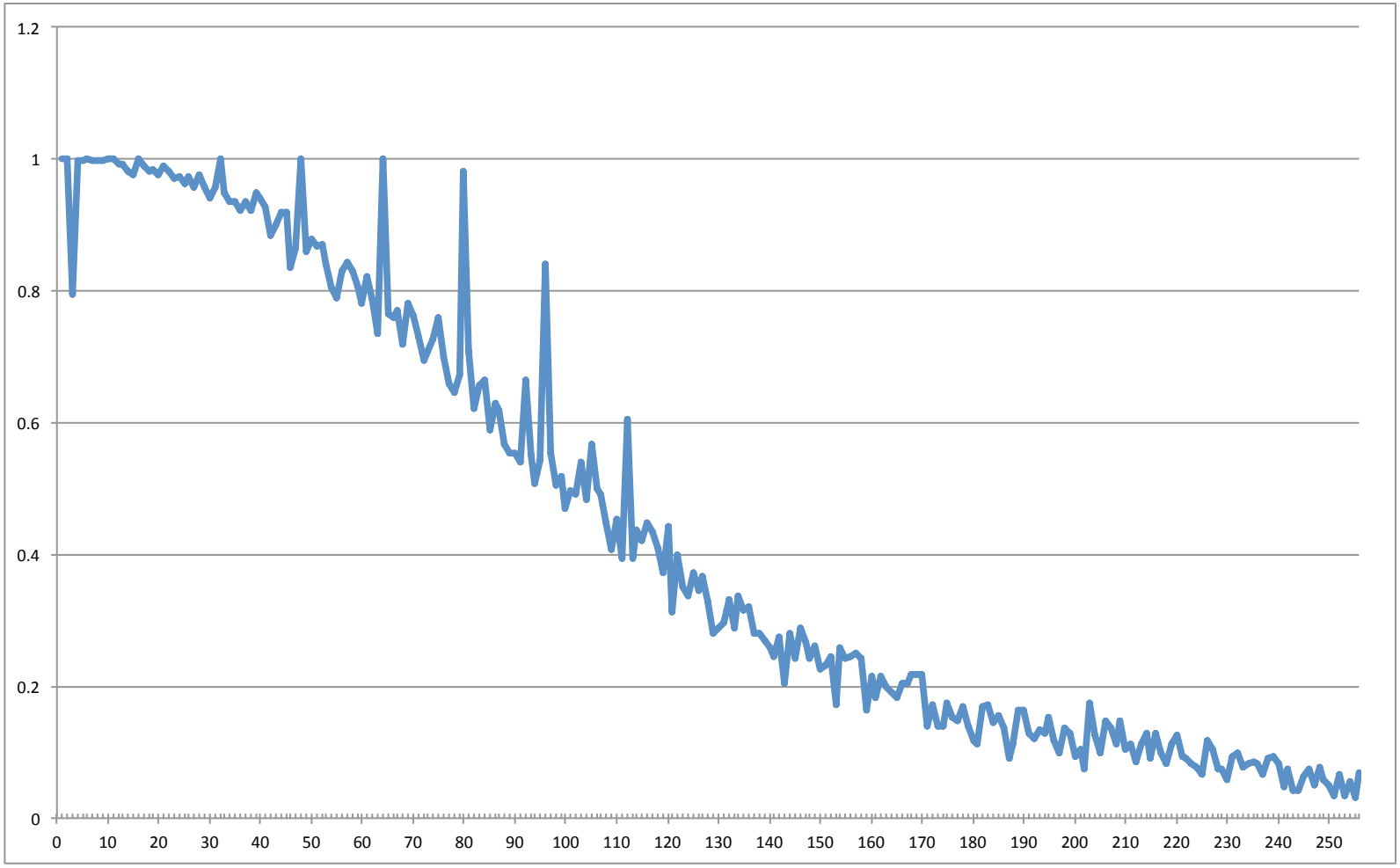
$$C_i = P_i \oplus K_i$$

  - A guess for P_i then induces a distribution on K_i.
  - Estimate likelihood of that induced distribution on K_i by comparing it to the previously measured distribution on K_i.
  - Select as correct plaintext byte the candidate P_i giving highest likelihood for the distribution on K_i.

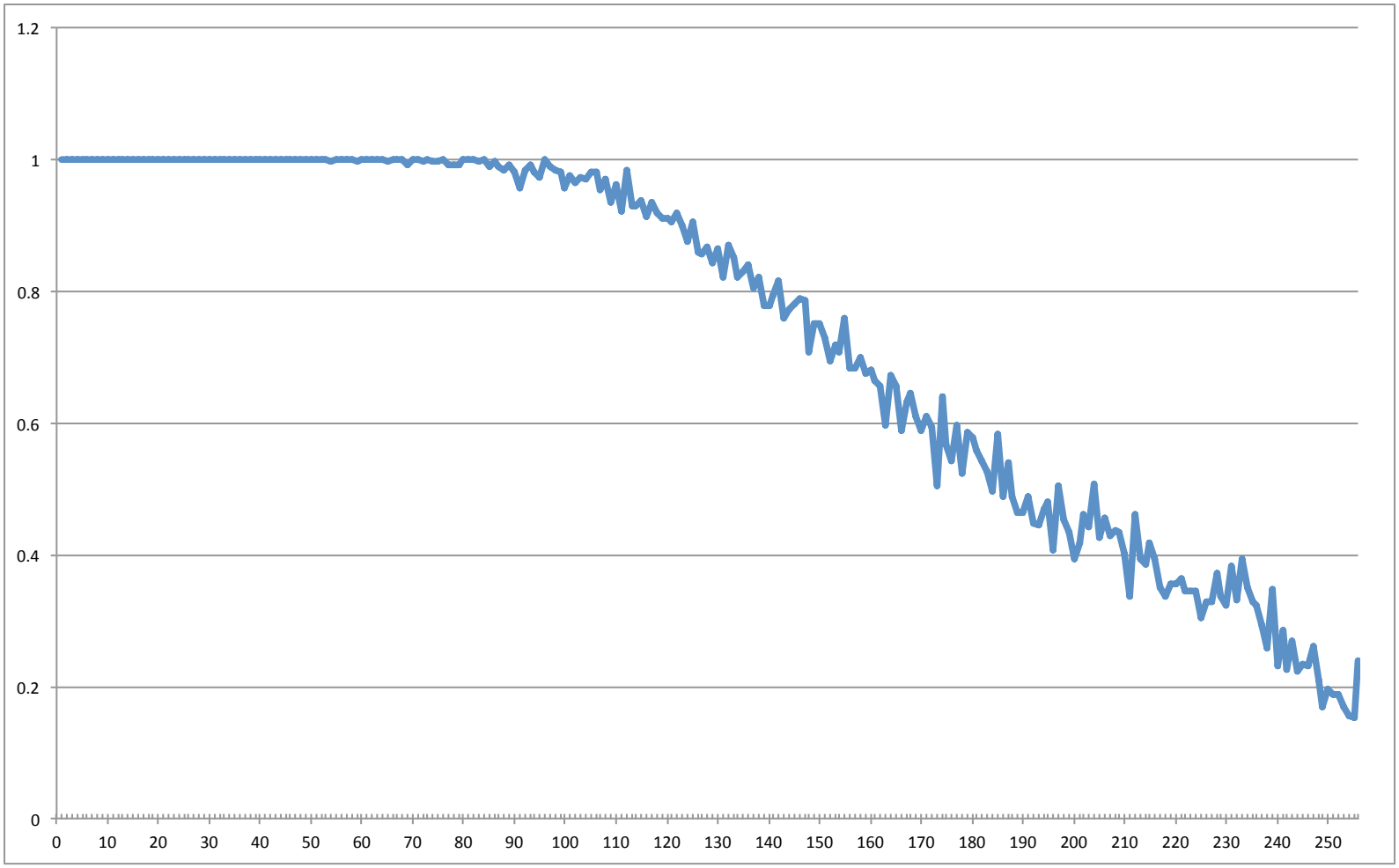
Success probability: 2^{24} sessions



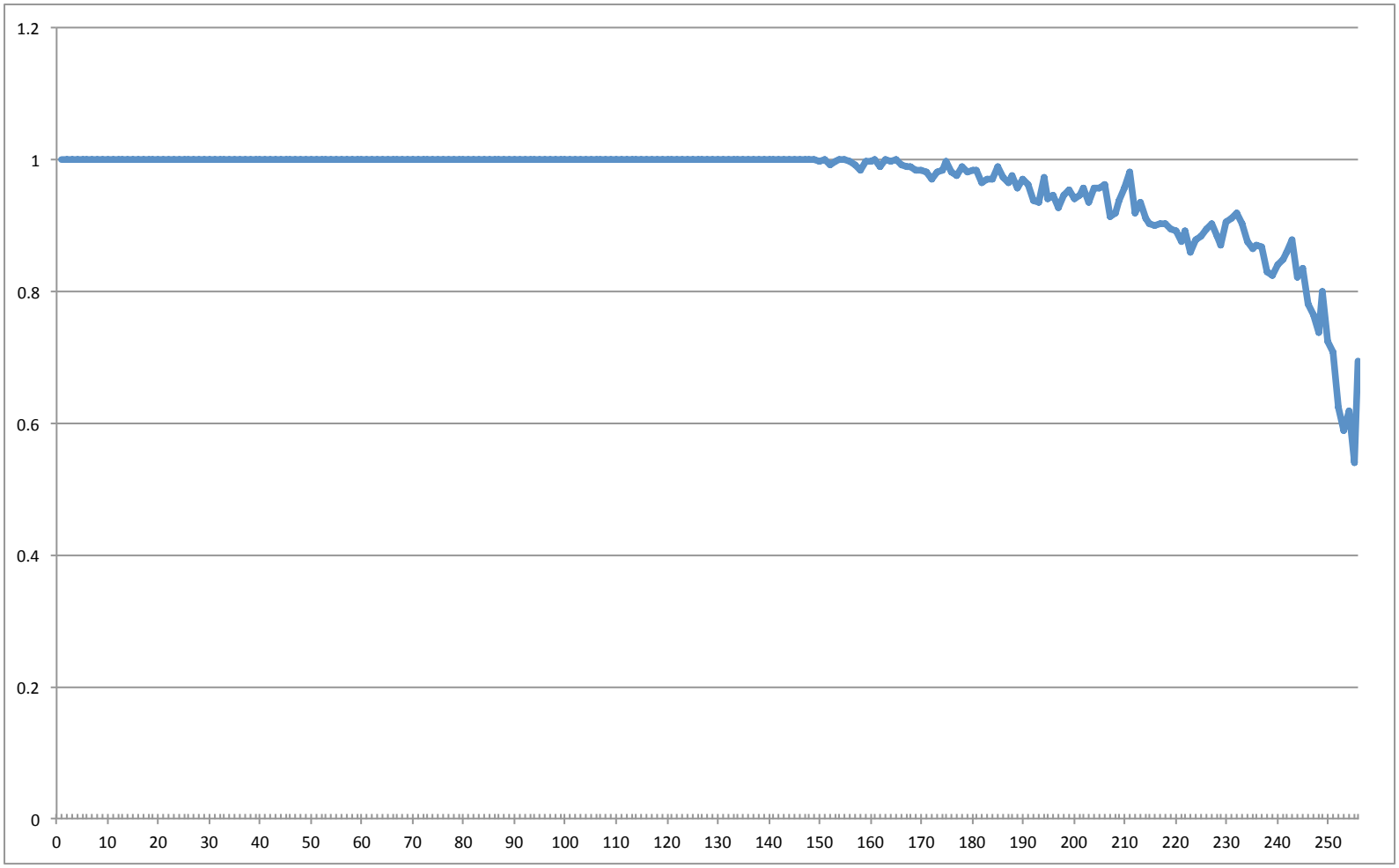
Success probability: 2^{26} sessions



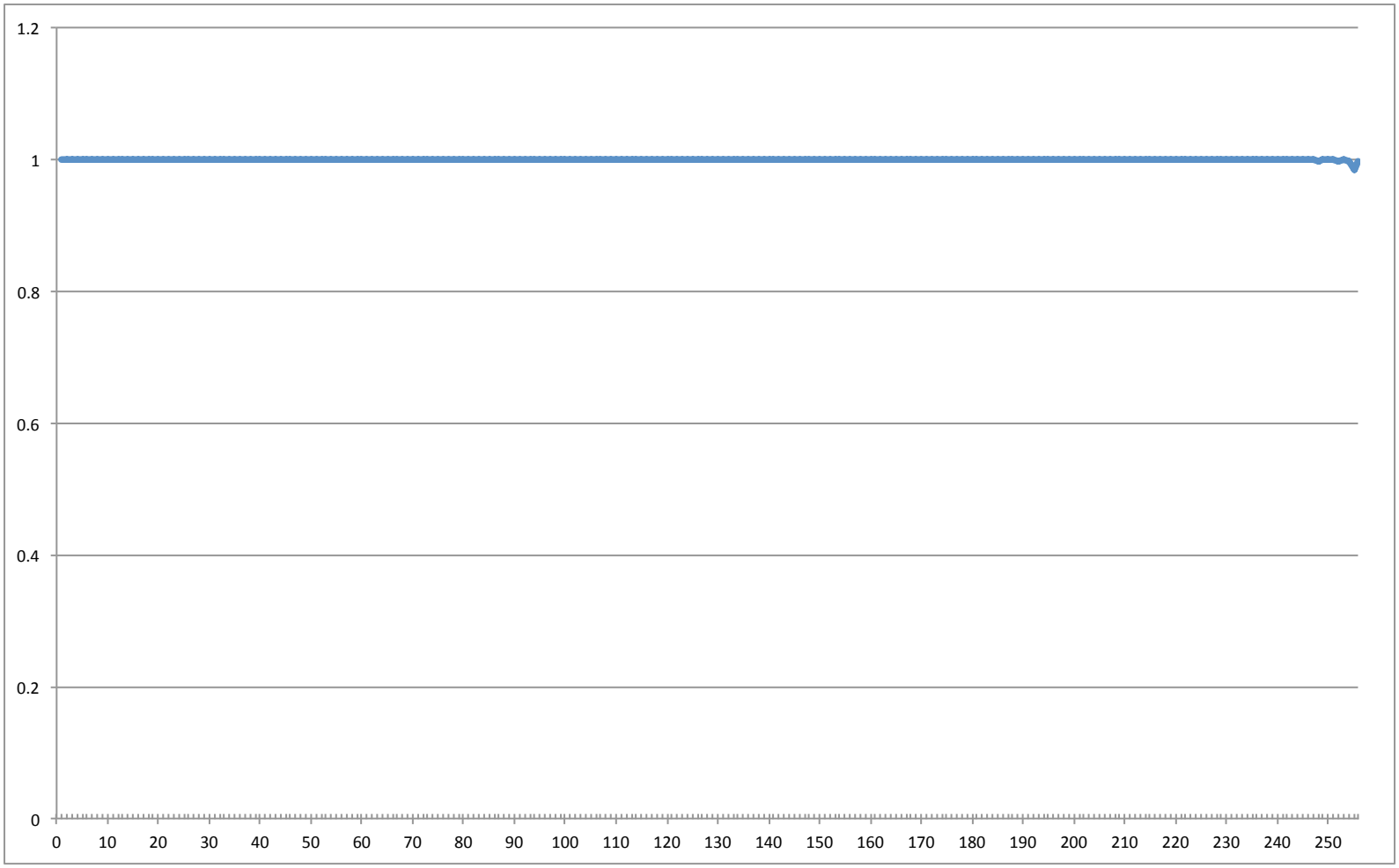
Success probability: 2^{28} sessions



Success probability: 2^{30} sessions



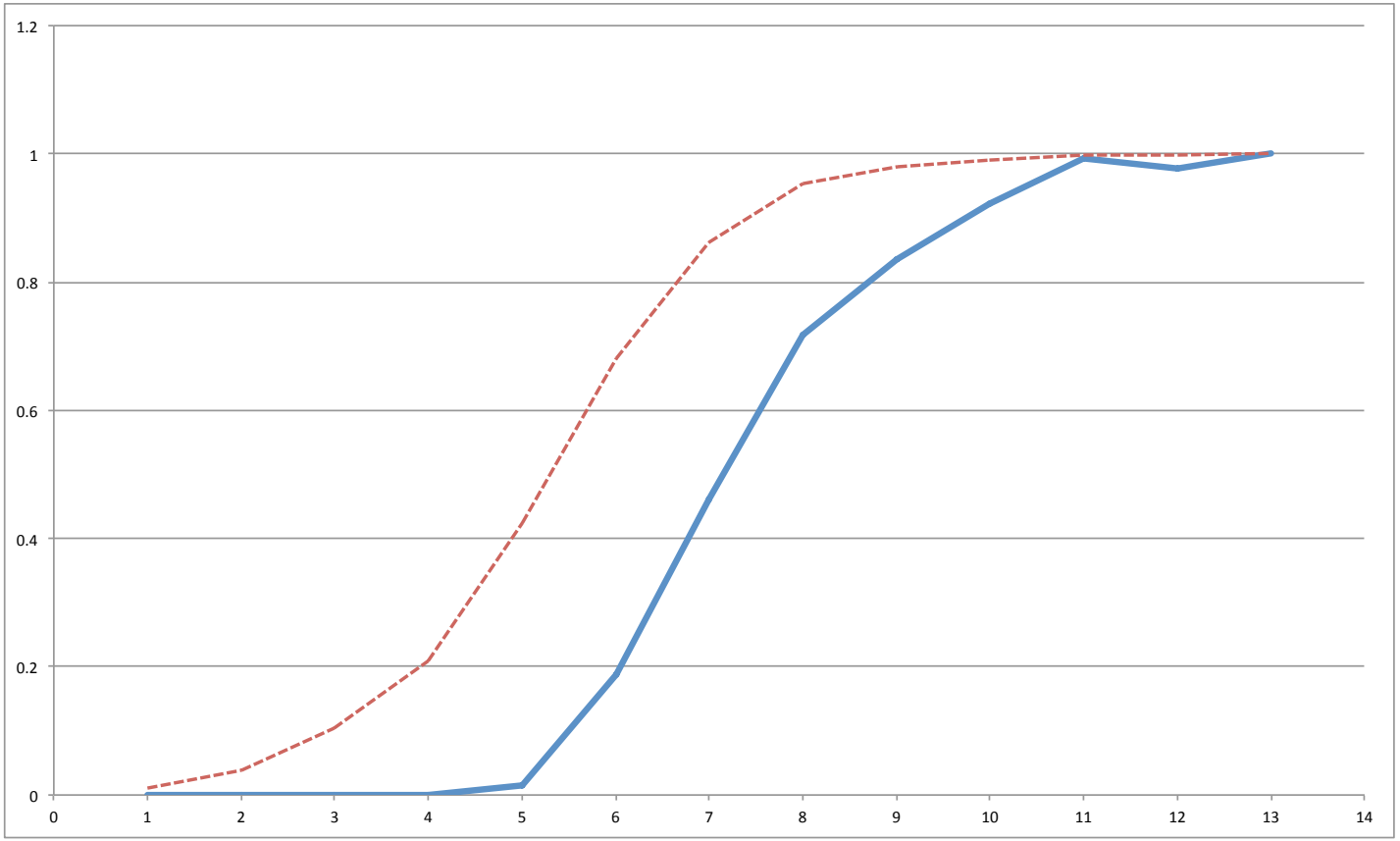
Success probability: 2^{32} sessions



Possible Attack Enhancements

- Many sessions are needed.
- The attack can only target the first 256 plaintext bytes.
  - Containing less interesting HTTP headers.
- In [ABPPS13], both problems were solved using 2-byte Fluhrer-McGrew biases.
  - Smaller biases, but persistent throughout keystream.
  - Arrange for HTTP session cookie to be repeatedly sent at predictable locations in keystream.
  - Use Viterbi-style algorithm to do ML estimation of plaintext bytes.
  - Roughly 2^{33} encryptions needed for reliable recovery.

Results for 2-byte Biases



x-axis: units of 2^{30} encryptions.

Blue line: success rate for 16-byte plaintext recovery.

Red line: success rate for individual byte recovery.

TLS-RC4 Attack – Countermeasures

- We can't just discard initial output bytes without updating all clients and servers simultaneously.
  - And this doesn't help against 2-byte attacks anyway.

- We had lots of discussion with vendors on *ad hoc* measures for HTTP.
  - Randomisation.
  - Burn-off initial bytes via short messages.
  - Put limits on number of times cookies can be sent.

TLS-RC4 Attack – Impact

- Fewer vendors have reacted publicly.
  - Google focussed on implementing TLS 1.2.
  - Microsoft disabled RC4 in Windows 8.1 Preview.
  - Opera has implemented cookie limit countermeasure.

- Further details at: www.isg.rhul.ac.uk/tls

CRIME/BREACH

- Duong and Rizzo [DR12] found a way to exploit TLS's optional compression feature.
  - Similar to idea in 2002 paper by Kelsey.
- Compression algorithms are stateful.
  - Replace repeated strings by shorter references to previous occurrences.
- Degree of compression obtained for chosen plaintext reveals something about prior plaintexts!
- This small amount of leakage can be boosted to get plaintext recovery attack for HTTP cookies.
  - Using same chosen plaintext vector as for BEAST.
- Countermeasures: disable compression; use variable length padding.
- BREACH: similar ideas, applied to HTTP compression.

Outline

- TLS overview
- **TLS Record Protocol**
  - Theory
  - Attacks
  - **Security analysis**
- TLS Handshake Protocol
  - Security analysis
- Discussion

Security Proofs for TLS Record Protocol

- Proof for RC4 not possible because of statistical flaws in stream cipher.
  - c.f. theoretical result of Krawczyk
- AE modes (AES-GCM, AES-CCM) already come with security proofs.
  - They achieve sfAEAD security under reasonable assumptions.
- This leaves CBC mode for further analysis.
  - So what have we learned so far?

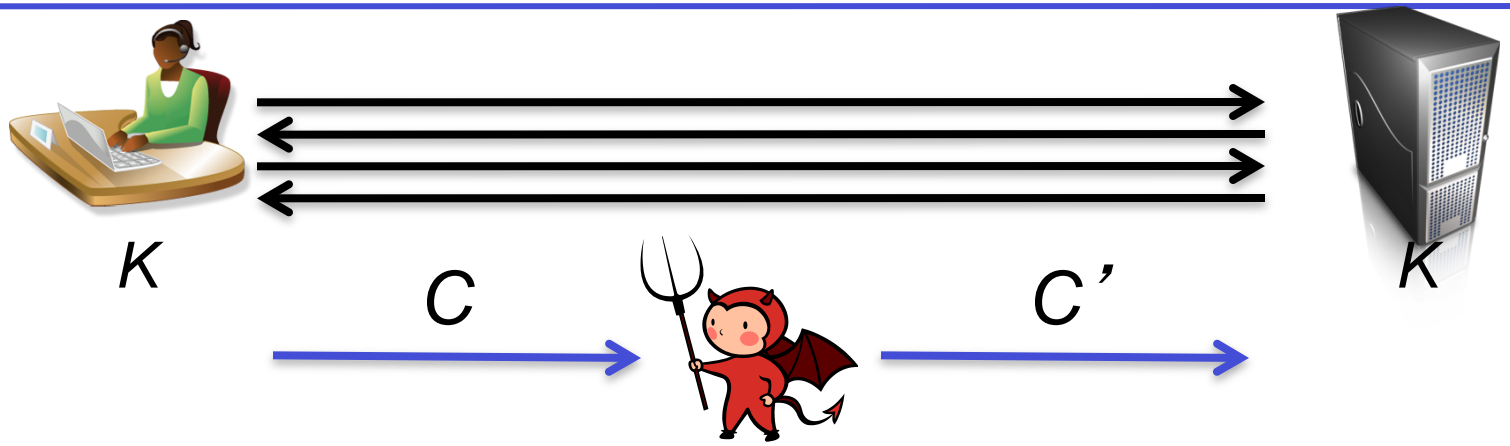
Security Proofs for TLS Record Protocol (CBC mode)

By now, a standards-compliant implementation of TLS in CBC mode should have:

- Explicit, random IVs
  - To prevent Dai-Rogaway-Moeller/BEAST
- Padding checks
  - To prevent Moeller attack.
- Uniform behaviour under padding and MAC failures
  - To prevent padding oracle and Lucky 13 attacks.
- Variable length padding.
  - To disguise true plaintext lengths.

[PRS11] Attack Against TLS

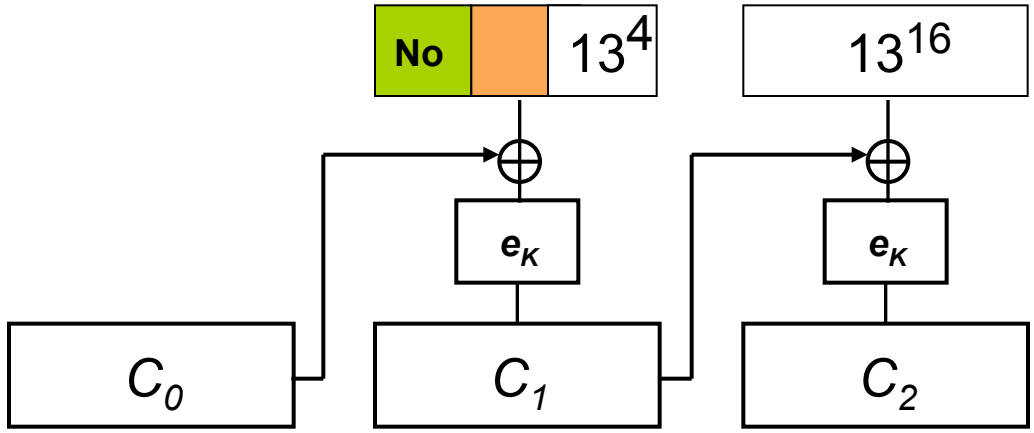
https://amazon.com



$C = \text{Enc}_K(M)$ M is either “Yes” or “No”

- Adversary intercepts C, flips a few bits, and forwards it on to recipient.
- How recipient responds will indicate whether M = “Yes” or “No”.
- Ciphertext-only distinguishing attack.
- The attack works when MAC size < block size.

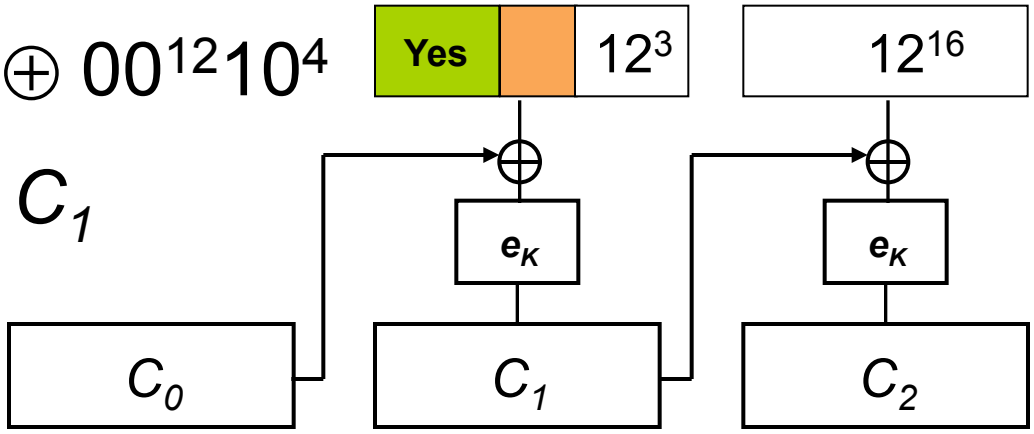
MAC length t = 80, block length n = 128



Byte 13 is hex for 19

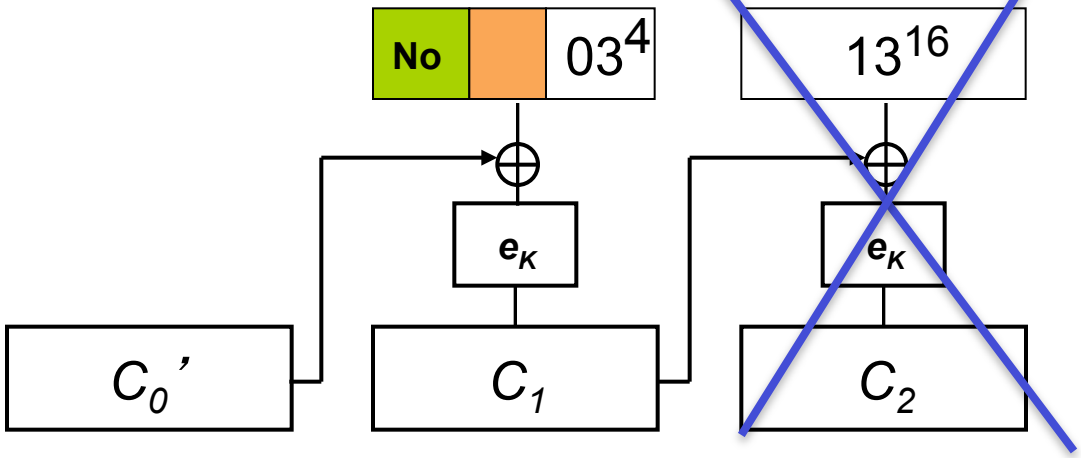
$C_0' = C_0 \oplus 00^{12}10^4$

$C' = C_0' C_1$



Byte 12 is hex for 18

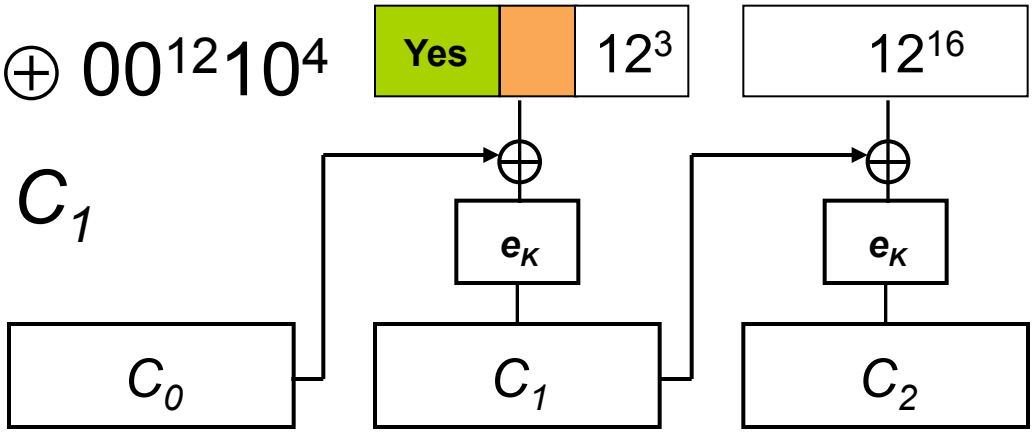
MAC length t = 80, block length n = 128



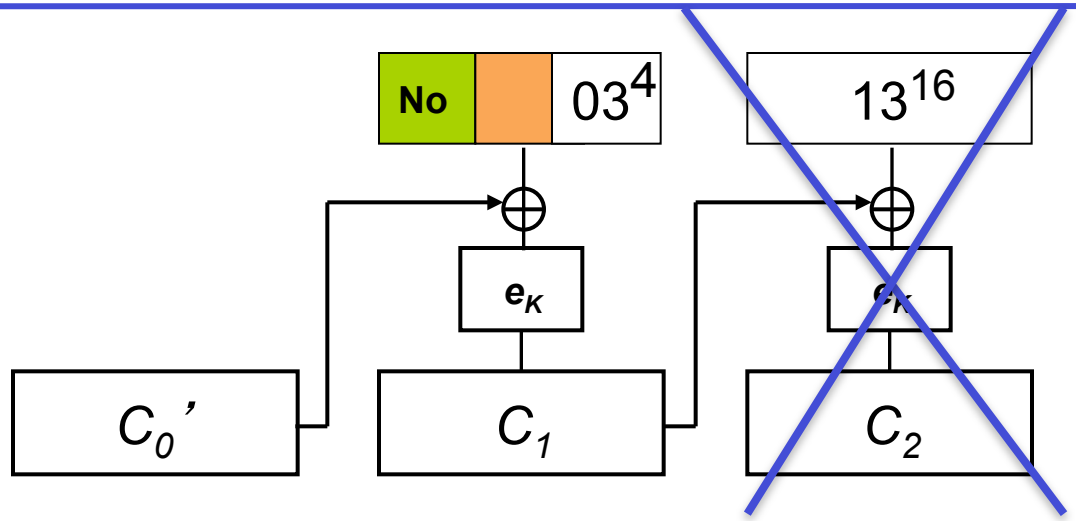
Decrypts fine to "No"

$C_0' = C_0 \oplus 00^{12}10^4$

$C' = C_0' \, C_1$



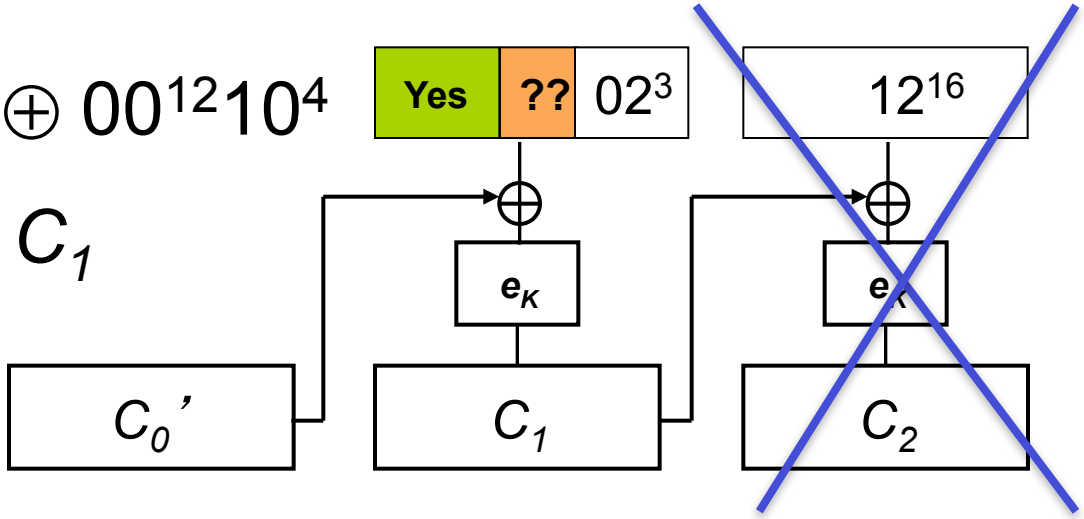
MAC length t = 80, block length n = 128



Decrypts fine to "No"

$C_0' = C_0 \oplus 00^{12}10^4$

$C' = C_0' \, C_1$



MAC will not verify, decryption fails

Where Does the [PRS11] Attack Apply?

For TLS 1.2:

Block length

n = 64 for 3DES

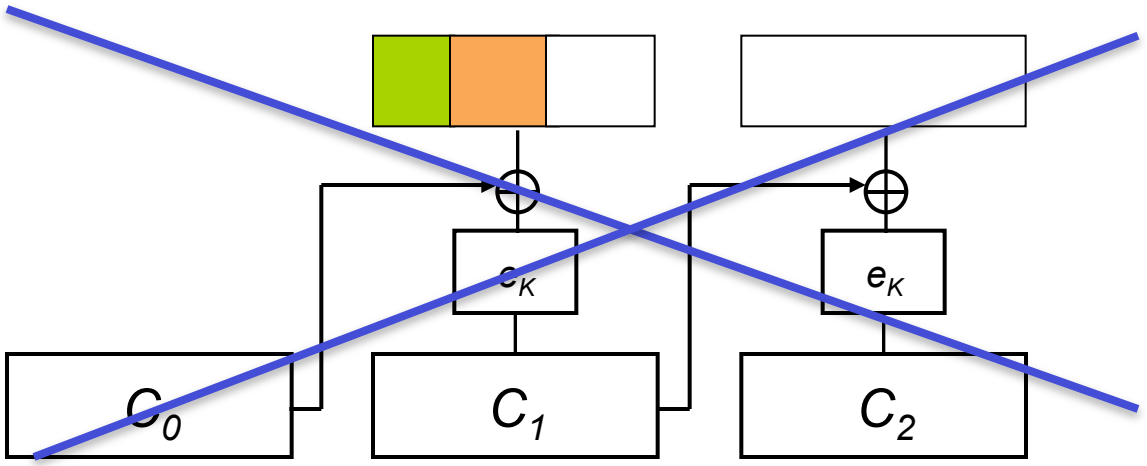
n = 128 for AES

MAC length

t = 128 for HMAC-MD5

t = 160 for HMAC-SHA1

t = 256 for HMAC-SHA256

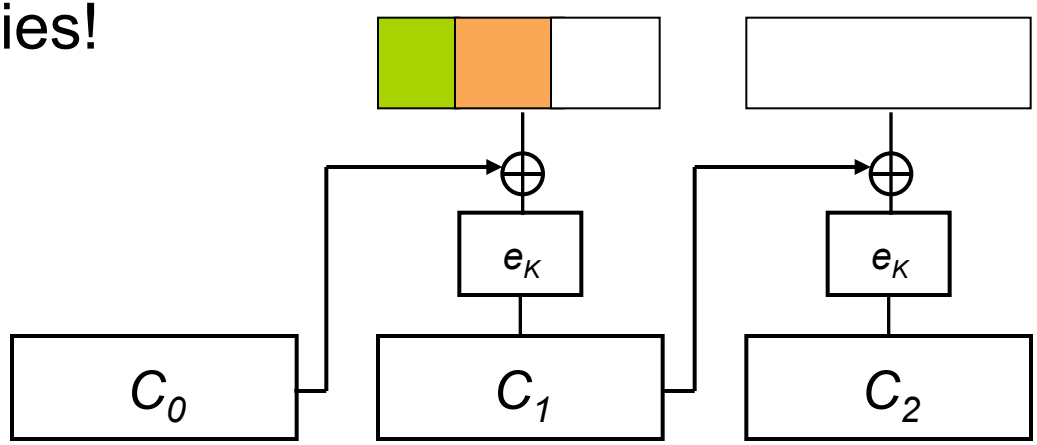


Where Does the [PRS11] Attack Apply?

For TLS 1.2 with truncated MAC extension (RFC 6066):

Block length	**MAC length**
n = 64 for 3DES	t = 80 for Truncated HMAC-MD5
n = 128 for AES	t = 80 for Truncated HMAC-SHA1
	t = 80 for Truncated HMAC-SHA256

Attack applies!



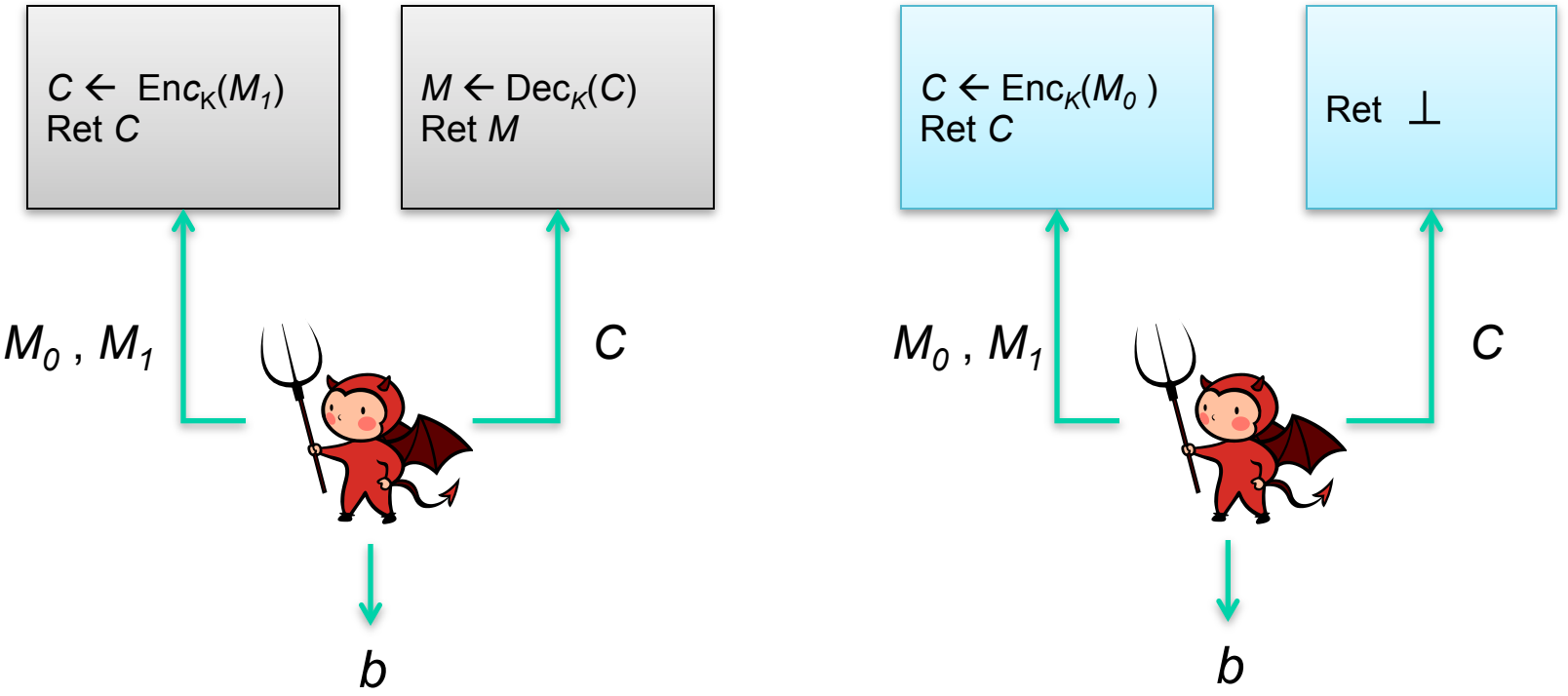
Consequences of the [PRS11] Attack

- This does **not** yield an attack against TLS, but only because no short MAC algorithms are *currently* supported in implementations.

- The attack is "only" a distinguishing attack.
  - Does not seem to extend to plaintext recovery.
  - But ciphertext-only rather than chosen-plaintext.

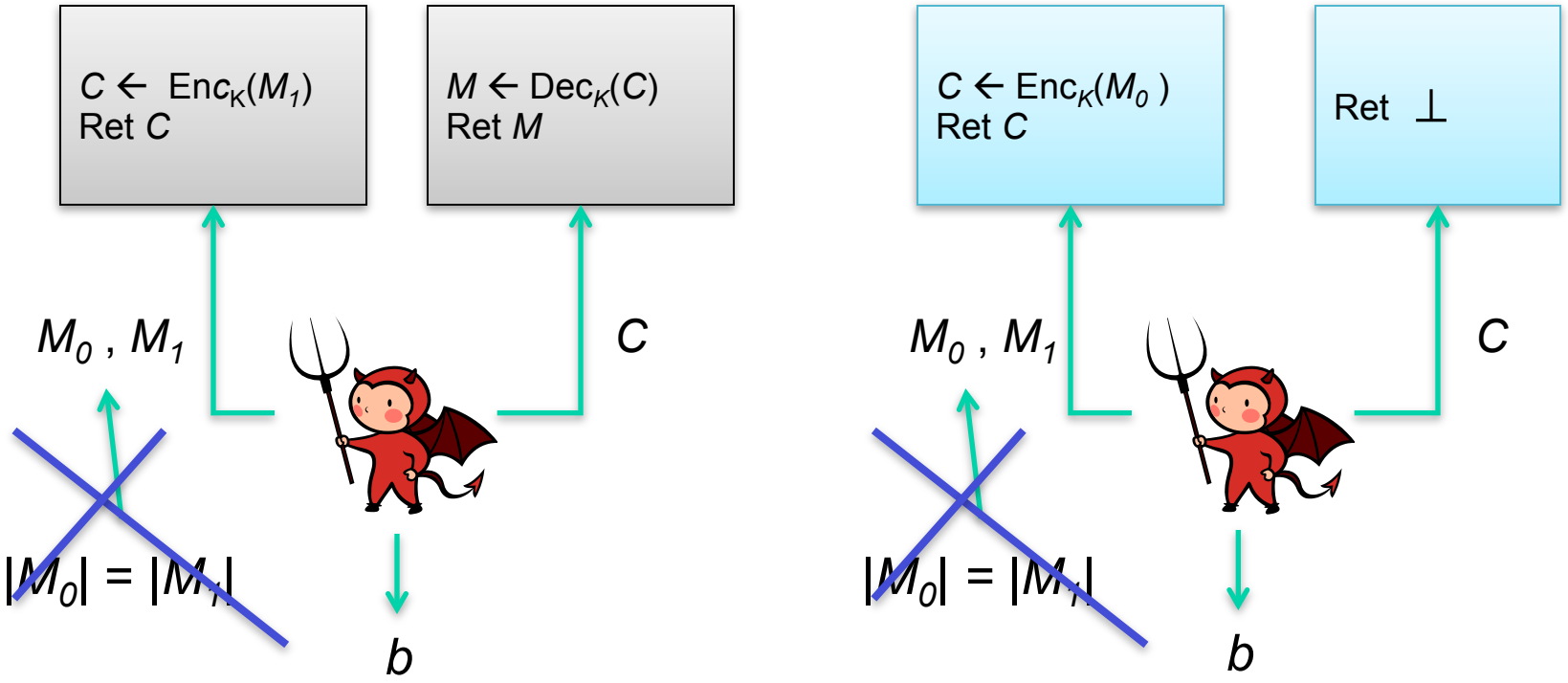
- The attack presents a barrier to obtaining proofs of security for TLS MEE construction.
  - Attack exploits variable length padding to break INT-CTXT security, leading to IND-CCA attack.

Combined AE Security Notion



AE ⇄ IND-CPA + INT-CTXT

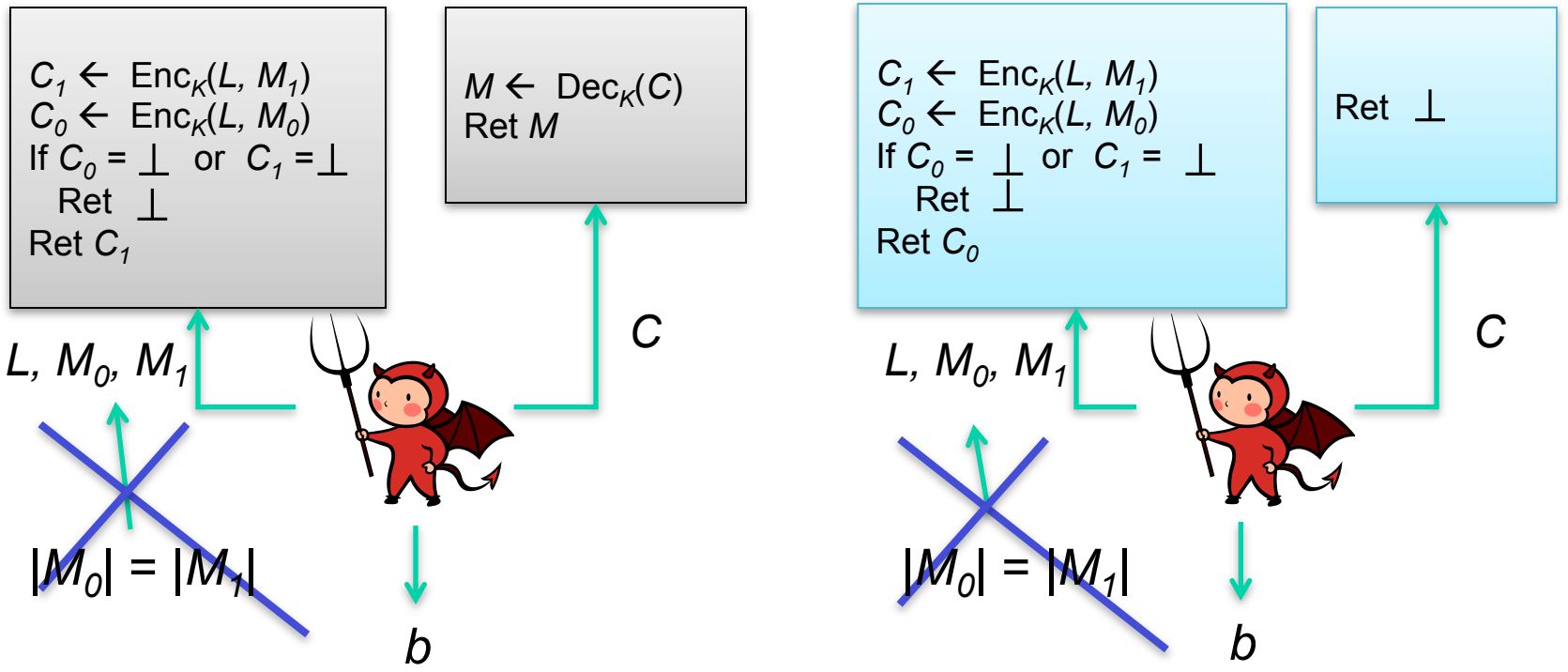
Combined AE Security Notion



Authenticated-Encryption security does not protect against adversary who can select messages of different lengths.

So [PRS11] attack is outside this model.

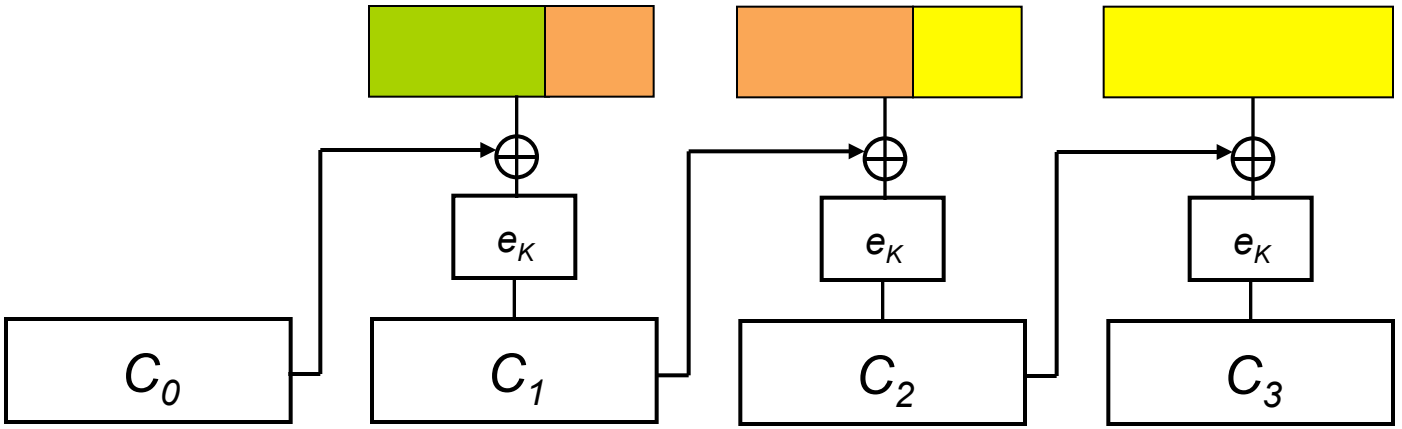
Length-hiding Authenticated Encryption (LHAE) Security



LHAE security protects against learning partial information about messages of (some) different lengths and forging ciphertexts

LHAE ⇄ **LH-IND-CPA + INT-CTXT** ⇄̸ **AE**

Towards LHAE Security



LHAE ⇄ **LH-IND-CPA + INT-CTXT**

Showing LH-IND-CPA is easy from IND-CPA of CBC.

INT-PTXT is straightforward from results of [BN00].

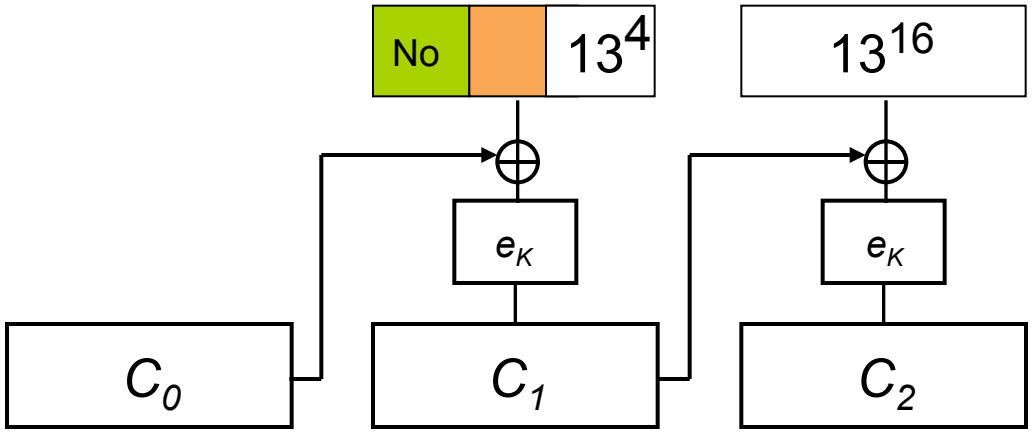
But we need INT-CTXT, and INT-PTXT does not imply it.

Collision-Resistant Decryption (CRD) Security

This is exactly the ‘gap’ between INT-PTXT and INT-CTXT:

INT-CTXT ⇄ **INT-PTXT + CRD**

Recall in our attack, adversary creates a new ciphertext that decrypts to a previously encrypted message.



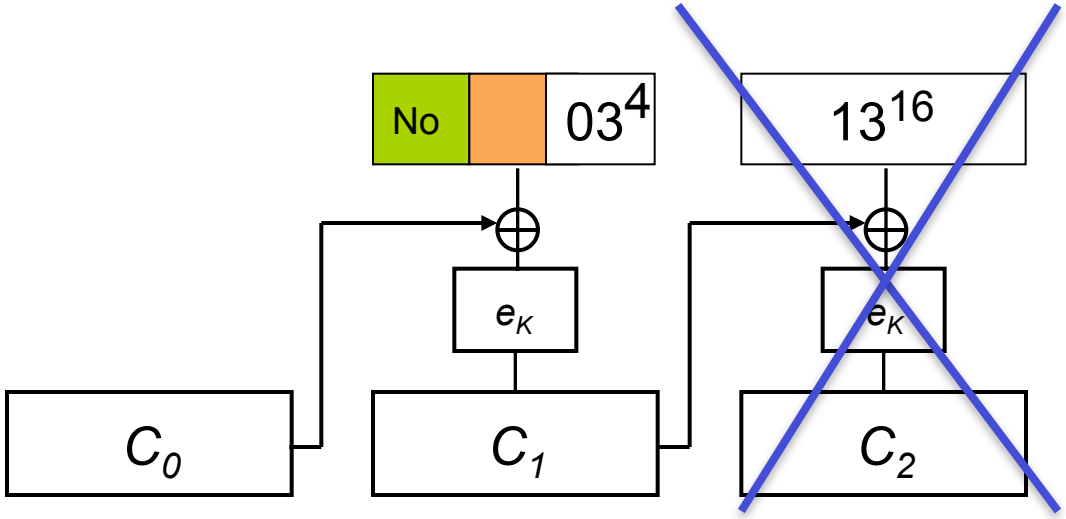
Byte 13 is hex for 19

Collision-Resistant Decryption (CRD) Security

This is exactly the ‘gap’ between INT-PTXT and INT-CTXT:

INT-CTXT ⇄ **INT-PTXT + CRD**

Recall in our attack, adversary creates a new ciphertext that Decrypts to a previously encrypted message.



Byte 13 is hex for 19

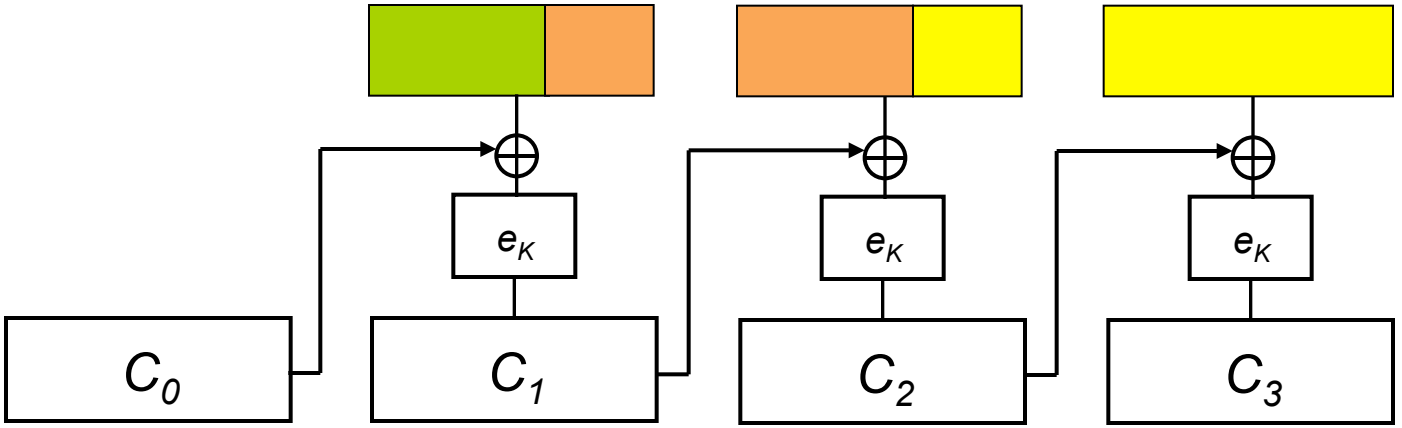
Achieving CRD security shows that no such attacks exist

LHAE Security for TLS

Theorem ([PRS11], informal statement)
Suppose E is a block cipher with block size *n* that is sprp-secure.
Suppose MAC has tag size *t* and is prf-secure.
Suppose that for all messages *M* queried by the adversary:

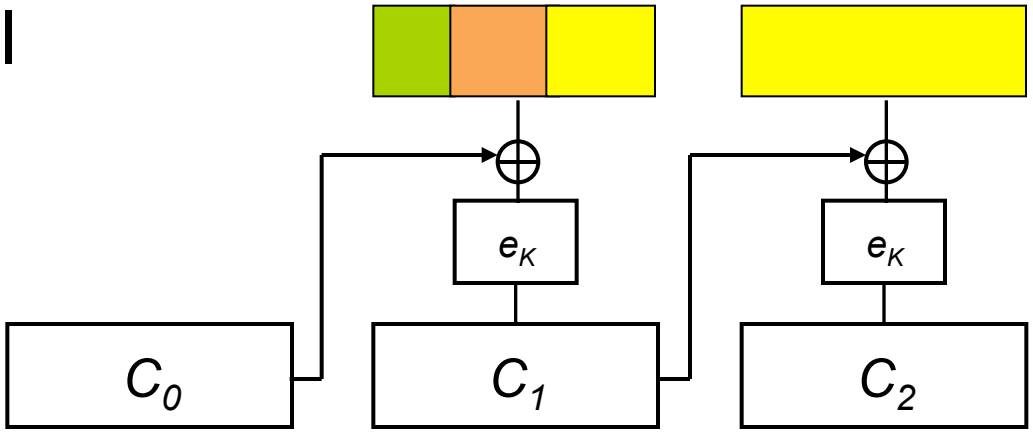
$$|M| + t \geq n.$$

Then MEE with CBC mode encryption, random IVs, TLS padding, and *uniform errors* is (LH)AE secure.

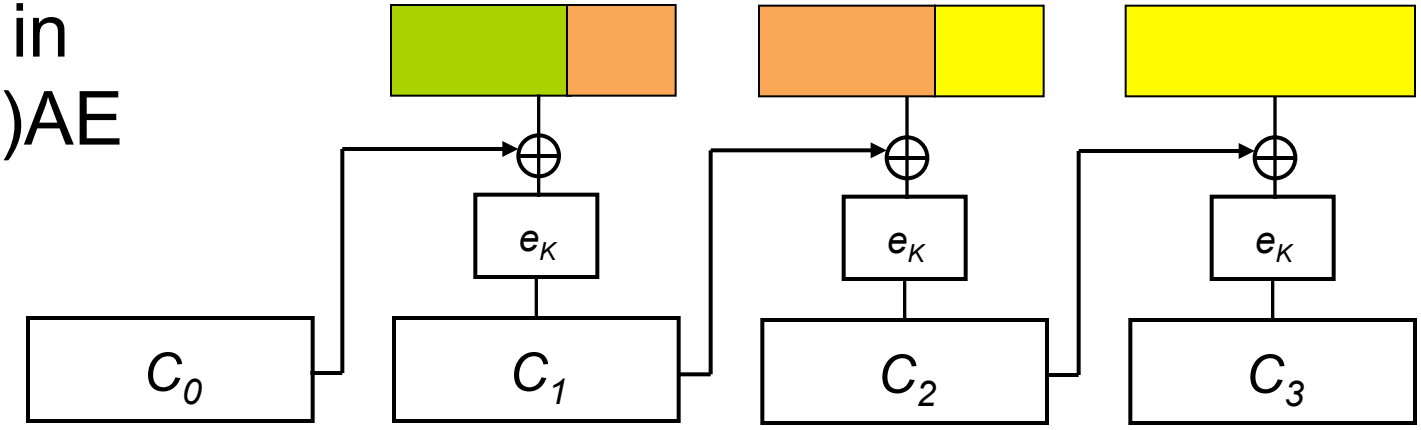


[PRS11]: Tag size matters!

Practical attacks exist



Secure in the (LH)AE model



Outline

- TLS overview
- TLS Record Protocol
  - Theory
  - Attacks
  - Security proofs
- **TLS Handshake Protocol**
  - Security analysis
- Discussion

TLS Handshake Protocol Security Goals

- Entity authentication of participating parties.
  - Server nearly always authenticated, client more rarely.
  - Appropriate for most e-commerce applications.

- Establishment of a fresh, shared secret.
  - Shared secret used to derive further keys.
  - For confidentiality and authentication/integrity in SSL Record Protocol.

- Secure negotiation of all cryptographic parameters.
  - SSL/TLS version number.
  - Encryption and hash algorithms.
  - Authentication and key establishment methods.
  - To prevent version rollback and ciphersuite downgrade attacks.

TLS Handshake Protocol Run

- An illustrative protocol run follows.

- We choose a common configuration of TLS:
  - No client authentication.
  - Client sends `pre_master_secret` encrypted under Server's RSA public key
  - Server public key obtained from server certificate.
  - Server authenticated by ability to decrypt to obtain `pre_master_secret`, and construct correct `finished` message.

- Other protocol runs are similar.

TLS Handshake Protocol Run

M1: C → S: **`ClientHello`**

M2: S → C: `ServerHello,`
`[Certificate,ServerKeyExchange,`
`CertificateRequest,] ServerHelloDone`

M3: C → S: [`Certificate,`] `ClientKeyExchange,`
`[CertificateVerify,] ChangeCipherSpec,`
`ClientFinished`

M4: S → C: `ChangeCipherSpec,ServerFinished`

- `[]` denotes optional/situation-dependent field.
- `ChangeCipherSpec` messages not part of Handshake.

TLS Handshake Protocol Run

M1: C → S: `ClientHello`

- Message includes client version number.
  - 3.0 for SSLv3, 3.1 for TLS1.0, 3.2 for TLS1.1 and 3.3 for TLS1.2
- Message also includes `ClientNonce` and `SessionID`.
  - Nonce is 28 random bytes plus 4 bytes of time.
  - `SessionID` used to signal request to set up new connection for existing session or to signal completely new session.
  - Details later.
- Message offers list of ciphersuites.
  - Key exchange and authentication options, encryption algorithms, hash functions.
  - E.g. TLS_RSA_WITH_AES_256_CBC_SHA256; TLS_RSA_WITH_3DES_EDE_CBC_SHA.
  - Each encoded by 2-byte field.
  - More than 200 ciphersuites known (see http://www.thesprawl.org/research/tls-and-ssl-cipher-suites/).

TLS Handshake Protocol Run

M1: C → S: `ClientHello`

M2: S → C: **`ServerHello`**, `[`**`Certificate`**`, ServerKeyExchange, CertificateRequest,]` **`ServerHelloDone`**

M3: C → S: `[Certificate,] ClientKeyExchange, [CertificateVerify,] ChangeCipherSpec, ClientFinished`

M4: S → C: `ChangeCipherSpec,ServerFinished`

- `[]` denotes optional/situation-dependent field.
- `ChangeCipherSpec` messages not part of Handshake.

TLS Handshake Protocol Run

M2: S → C: **`ServerHello`**, `[Certificate, ServerKeyExchange, CertificateRequest,] ServerHelloDone`

- **`ServerHello`** message contains server version number.
- Contains `ServerNonce` and `SessionID`.
  - `SessionID` will match client's if new connection for existing session; otherwise selected by server.
- Selects single ciphersuite from list offered by client.
  - E.g. TLS_RSA_WITH_AES_256_CBC_SHA256.

TLS Handshake Protocol Run

M2: S → C: `ServerHello, [`**`Certificate`**`, ServerKeyExchange, CertificateRequest,]` **`ServerHelloDone`**

- **`Certificate`** message.
  - Allows client to validate server's public key back to acceptable root of trust.
- [Optional] `ServerKeyExchange` message.
  - Omitted in this protocol run – no server contribution to key establishment.
- [Optional] `CertificateRequest` message.
  - Omitted in this protocol run – no client authentication.
- Finally, **`ServerHelloDone`** indicating server is done.

TLS Handshake Protocol Run

M1: C → S: `ClientHello`

M2: S → C: `ServerHello, [Certificate, ServerKeyExchange, CertificateRequest,] ServerHelloDone`

M3: C → S: [`Certificate,`] **`ClientKeyExchange`**, [`CertificateVerify,`] **`ChangeCipherSpec`**, **`ClientFinished`**

M4: S → C: `ChangeCipherSpec,ServerFinished`

- `[]` denotes optional/situation-dependent field.
- `ChangeCipherSpec` messages not part of Handshake.

TLS Handshake Protocol Run

M3: C → S: [`Certificate,`] **`ClientKeyExchange`**, [`CertificateVerify,`] `ChangeCipherSpec`, `ClientFinished`

- **`ClientKeyExchange`** contains encryption of `pre_master_secret` under server's RSA public key.
- [Optional] `Certificate`, `CertificateVerify` messages.
  - Only sent when client is authenticated.
  - `CertificateVerify` message is typically a signature on nonces (and other values) exchanged in the protocol run.

TLS Handshake Protocol Run

M3: C → S: [`Certificate,`] `ClientKeyExchange,` [`CertificateVerify,`] **`ChangeCipherSpec`**, **`ClientFinished`**

- **`ChangeCipherSpec`** indicates that client is now switching to use of ciphersuite agreed for this session.
  - Sent using SSL/TLS Change Cipher Spec. Protocol.
  - Technically, an upper layer protocol.
  - Effectively, switches on Record Protocol crypto at client.
- **`ClientFinished`** message:
  - Computed as PRF applied to hash of all Handshake Protocol messages sent so far (by both sides).
  - Key for PRF is `master_secret`.
  - Provides protection of ciphersuite negotiation.

TLS Handshake Protocol Run

M1: C → S: `ClientHello`

M2: S → C: `ServerHello, [Certificate, ServerKeyExchange, CertificateRequest,] ServerHelloDone`

M3: C → S: [`Certificate,`] `ClientKeyExchange, [CertificateVerify,] ChangeCipherSpec, ClientFinished`

M4: S → C: **`ChangeCipherSpec`**, **`ServerFinished`**

- `[]` denotes optional/situation-dependent field.
- `ChangeCipherSpec` messages not part of Handshake.

TLS Handshake Protocol Run

M4: S → C: **`ChangeCipherSpec`**, **`ServerFinished`**

- **`ChangeCipherSpec`** indicates that server is now switching to ciphersuite agreed for this session.
  - Sent using SSL Change Cipher Spec. Protocol.
- Finally, **`ServerFinished`** message.
  - Computed as PRF applied to hash of all messages sent so far (by both sides).
  - Key for PRF is `master_secret`.
  - Server can only compute PRF if it can decrypt `ClientKeyExchange` in M3 to get `pre_master_secret` and then derive `master_secret`.
  - In this protocol run, provides server authentication and protection of ciphersuite negotiation.

TLS Handshake Protocol Run Summary

M1: C → S: **ClientHello**

M2: S → C: **ServerHello**, [**Certificate**, ServerKeyExchange, CertificateRequest,] **ServerHelloDone**

M3: C → S: [Certificate,] **ClientKeyExchange**, [CertificateVerify,] **ChangeCipherSpec**, **ClientFinished**

M4: S → C: **ChangeCipherSpec**,**ServerFinished**

- Important note: **Finished** messages are sent *after* **ChangeCipherSpec** messages, so are actually encrypted by Record Protocol!
- So keys set up by the Handshake are used in the Handshake itself.

Outline

- TLS overview
- TLS Record Protocol
  - Theory
  - Attacks
  - Security analysis
- **TLS Handshake Protocol**
  - **Security analysis**
- Discussion

Security Analysis of TLS Handshake Protocol

- Models for analysis of key exchange protocols are fairly mature.
  - Beginning with [BR93].
- But they are also quite complex.
  - Multiple, interacting parties.
  - Multiple sessions at each party.
  - Adversary given complete control of the network (“adversary is the network”).
  - Adversarial access to various session keys, long-term secrets, ephemeral values, randomness.
  - Forward security?
  - Corruption of parties?
  - Dishonest long-term key pair generation?
  - Modeling of CA/PKI?
  - Authentication and key security properties.

Security Analysis of TLS Handshake Protocol

- Additional barriers to analysis for TLS:
  - Protocol not designed with provable security in mind.
  - Protocol "mode" and ciphersuite are negotiated during the protocol itself and verified later via `Finished` messages.
  - Ditto with protocol version.
  - Unilateral and mutual authentication modes
  - Renegotiation and session resumption features.
  - The session key is used in the Handshake Protocol itself (so can't prove usual "indistinguishability of session keys" property).
  - RSA encryption used in TLS is not IND-CCA secure.
    - Bleichenbacher's attack addressed via protocol-specific fix.
  - [MVVP12] attack based on ECDH/DH ciphersuite switching.
  - Static DH harder to analyse then usual ephemeral DH because server acts as "fixed base DH oracle" by calculating and using g^{xy} for fixed g, g^x.
  - What to leave out and what to include when modelling the protocol?
- TL;DR: it's even harder than the Record Protocol!

Security Analysis of TLS Handshake Protocol

- Despite these barriers substantial progress has been made recently.

- [JKSS12]:
  - ACCE security notion, combining security of Handshake and Record Protocols.
  - Overcomes “indistinguishability barrier” from use of session key in Handshake Protocol.
  - Analysis of cryptographic core of individual *mutually authenticated ephemeral DH key exchange* ciphersuites.
  - Using a fairly faithful model of the TLS Protocol.

Security Analysis of TLS Handshake Protocol

- [KPW13]:
  - Adapt ACCE to unilateral case.
  - Analysis of cryptographic core of individual (*unilateral and mutually authenticated) RSA, static DH and ephemeral DH key exchange* ciphersuites.
  - Modular approach:
    - Extract Key Encapsulation Mechanism (KEM) from Handshake Protocol.
    - (S)ACCE security of TLS follows from constrained CCA security of this KEM.
    - Analyse the KEM for various ciphersuites.

- [GKS13]:
  - Formal security treatment of renegotiation and Ray-Dispensa attack.

Outline

- TLS overview
- TLS Record Protocol
  - Theory
  - Attacks
  - Security analysis
- TLS Handshake Protocol
  - Security analysis
- **Discussion**

Where Do We Stand Currently?

- Most TLS implementations now patched against BEAST.
- Many TLS implementations patched against Lucky 13.
- No simple TLS patch for RC4 attack.
  - Needs application-layer modifications.
- Disable TLS compression to prevent CRIME.
  - Still issues with compression at application layer (BREACH).

- We need TLS 1.2!
  - Use patched CBC mode until we get it.
  - Other people advise differently.
    - Their logic is that BEAST-vulnerable browsers still exist, so less-broken RC4 is preferable.

Discussion

- TLS's *ad hoc* MAC-Encode-Encrypt construction is hard to implement securely and hard to prove positive security results about.
  - Long history of attacks and fixes.
  - Each fix was the "easiest option at the time".
  - Now reached point where a 500 line patch to OpenSSL was needed to fully eliminate the Lucky 13 attack.
  - Attacks show that small details matter.
  - The actual TLS-CBC construction was only fully analysed in 2011.

- RC4 was known to be weak for many years.
  - Actual exploitation of weaknesses in a TLS context went unexplored.
  - Needed multi-session mechanism (BEAST technology) to make the attack plausible.

Discussion

- Once a bad cryptographic choice is out there in implementations, it's very hard to undo.
  - Old versions of TLS hang around for a long time.
  - There is no TLS product recall programme!
  - Slow uptake of TLS 1.1, 1.2.

- TLS is coming under sustained pressure from attacks.
  - BEAST, Lucky 13 and RC4 attacks are providing incentives to move to TLS 1.2.
  - Attacks are "semi-practical" but we ignore such attacks at our peril.
  - Good vendor response to Lucky 13, less so to RC4 attack.
    - One is fixable, the other not (really).

Lessons

- Attacks really do improve with age.
  - BEAST (1995 – 2011), Lucky 13 (Feb. ’13 – Mar. ‘13).
- Design any future AE schemes for a broad set of use-cases and attack vectors.
  - Fixed-key, many-key (multi-session) attacks.
  - Chosen-plaintext, partially known-plaintext, …
  - Compression-based attacks?
  - Look carefully at error conditions and handling.
- Be realistic about timescale for adoption in already deployed systems.

Research Directions?

- TLS Record Protocol cryptography has now been heavily analysed.
  - Still some mileage in looking at AE implementations?
- Major recent progress in analysing TLS Handshake protocol.
  - [JKSS12], [KPW13], [GKS12].
- Can still expect implementation issues to emerge.
  - Check the “OpenSSL Fact” twitter feed regularly!
- Complex system of interacting protocols can still throw up surprises.
  - Alert Protocol desynchronisation attack [BFKPS13].
  - TLS Renegotiation attack [RD09].

Literature

Theory for symmetric encryption:

- [BDJR97] Bellare *et al.*, FOCS 1997
- [BN00] Bellare and Namprempre, Asiacrypt 2000
- [K01] Krawczyk, Crypto 2001
- [BKN02] Bellare *et al.,* ACM-CCS 2001
- [RS06] Rogaway and Shrimpton, Eurocrypt 2006
- [PW10] Paterson and Watson, Eurocrypt 2010
- [PRS11] Paterson *et al.*, Asiacrypt 2011.
- [BDPS13] Boldyreva *et al.,* FSE 2013

Theory for key exchange in TLS:

- [BR93] Bellare and Rogaway, Crypto 1993.
- [JKSS12], Jager *et al.*, Crypto 2012.
- [GKS13] Giesen *et al.*, ACM-CCS 2013.
- [KPW13] Krawczyk *et al.*, Crypto 2013.

Literature

Attacks on TLS:

- [V02] Vaudenay, Eurocrypt 2002
- [M02] Moeller, http://www.openssl.org/~bodo/tls-cbc.txt, 2002
- [CHVV03] Canvel *et al.*, Crypto 2003
- [RD09] Ray and Dispensa, TLS renegotiation attack, 2009.
- [PRS11] Paterson *et al.*, Asiacrypt 2013
- [DR11] Duong and Rizzo, “Here come the XOR Ninjas”, 2011.
- [DR12] Duong and Rizzo, CRIME, 2012.
- [MVVP12] Mavrogiannopoulos *et al*.,CCS 2012.
- [AP13] N.J. AlFardan and K.G. Paterson, IEEE S&P, 2013.
- [ABPPS13] N.J. AlFardan *et al.*, USENIX Security, 2013.
- [BFKPS13] Bhargavan *et al.,* IEEE S&P, 2013.