Lecture 1
Kenny Paterson

Bar-Ilan Winter School on Symmetric Cryptography

Royal Holloway
University of London

Information Security Group

Lecture 1 Outline

Theory for symmetric encryption:

- Security models
- Modes of operation
- Message Authentication Codes (MACs)
- Generic composition

Theory for Symmetric Encryption

- Security models for symmetric encryption are well established.

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

    $m / \perp \leftarrow \text{Dec}_K(m)$.
  - Correctness requirement: for all $K \leftarrow \text{KGen}(1^k)$, for all *m* in *M:*

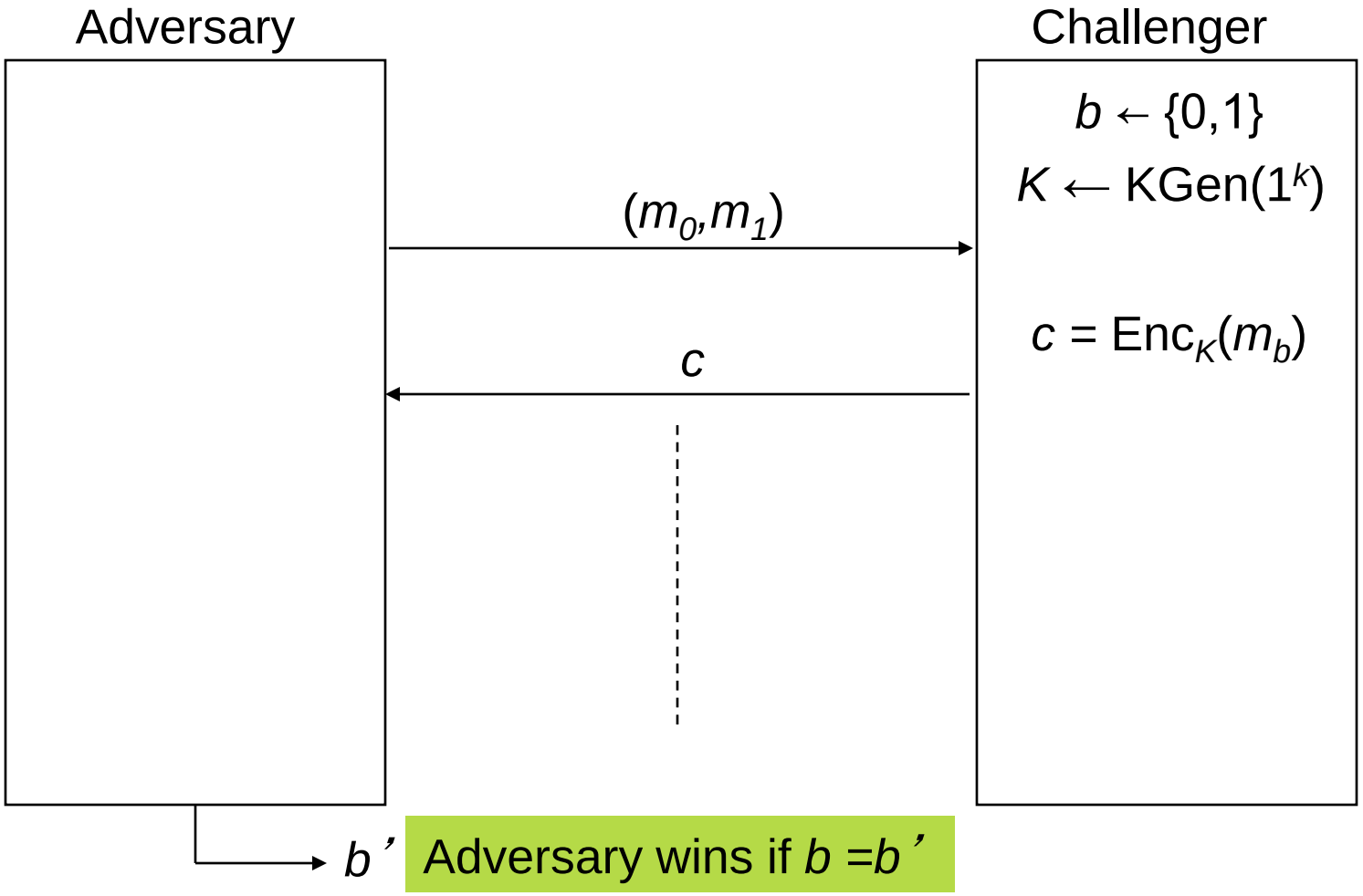
    $\text{Dec}_K(\text{Enc}_K(m)) = m.$
  - For now, we focus on *stateless* Enc and Dec algorithms.
  - Nonce-based algorithms in place of probablistic ones also possible [R04].

Theory for Symmetric Encryption

- IND-CPA security:
  - Adversary has repeated access to *Left-or-Right (LoR)* encryption oracle.
  - In each query, adversary submits pairs of equal length messages (m_0,m_1) to the oracle.
  - Receives *c*, an encryption of m_b, where *b* is a random bit.
  - Adversary outputs its estimate *b'* for bit *b*.
  - Adversary wins if it decides correctly.

- IND = Indistinguishable
- CPA = Chosen Plaintext Attack
- Formalised as a security game between the adversary and a challenger.

IND-CPA Security Game



IND-CPA Security

- The 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.
  - In symmetric crypto, the concrete approach is widely used.
  - Quantify adversary's success probability in terms of number of encryption queries and/or number of bits queried to encryption oracle.

IND-CPA Security

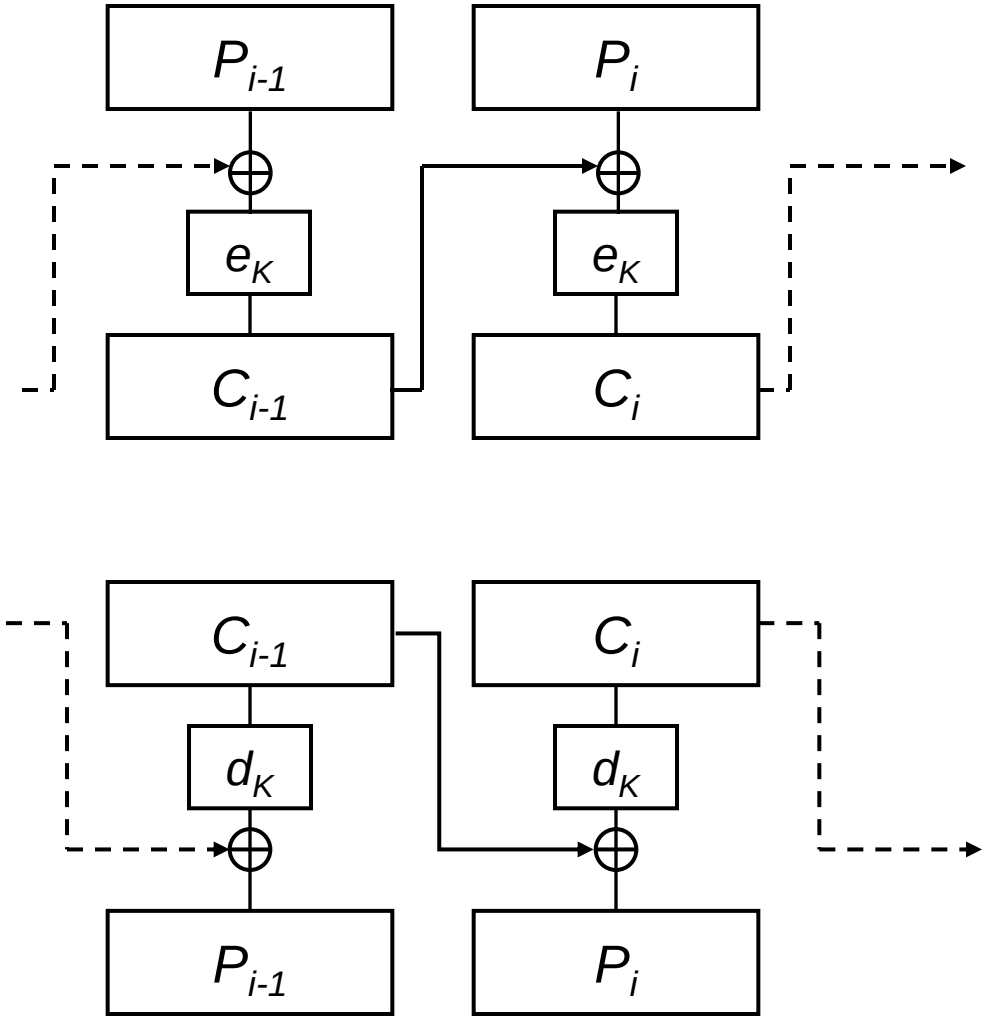
- Informally, IND-CPA is a computational version of perfect security.
  - Ciphertext leaks nothing about the plaintext.
  - Stronger notion than requiring the adversary to recover plaintext.

- [BDJR97] developed equivalent notions.
  - RoR-CPA, FtG-CPA and SEM-CPA.
  - The latter is a symmetric version of semantic security notion for PKE of Goldwasser-Micali.

IND-CPA Security

- Easy to achieve IND-CPA security using, for example, a suitable mode of operation of a block cipher f_K on $\{0,1\}^n$.
  - Block cipher in CBC mode with random IVs.
  - Block cipher in CTR mode.
  - See [BDJR97] for analysis.
  - Requires modelling of block cipher as PRP/PRF.
  - Recall definition:
    - Adversary has oracle access to $f_K(.)$ or Rand(.)
    - Adversary outputs guess for which world it is in.

CBC Mode

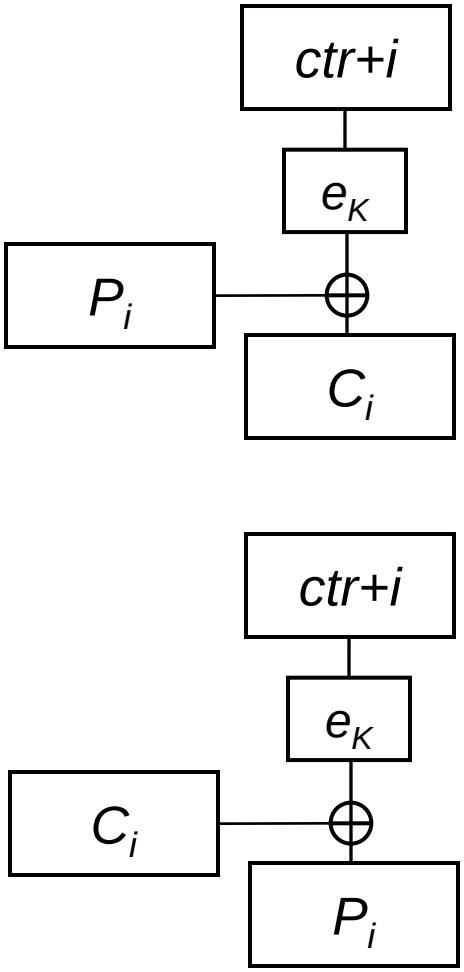


Initialisation Vector (IV):
- Defines C_0 for processing first block.
- IV often taken as random;
- Chained IVs also common in applications.

CBC mode needs some form of padding if plaintext lengths are not multiple of block length.
- (Much) more on padding later.

[BDJR97] security bound involves a $q^2/2^n$ term (quadratic loss).

CTR Mode



- CTR mode uses a block cipher to build a stream cipher.
- Block cipher does not even need to be a permutation!
- In simplest mode, random initial value for *ctr* is chosen for each message and transmitted with ciphertext.
- Encrypt blocks

  ctr, ctr+1, ctr+2, …

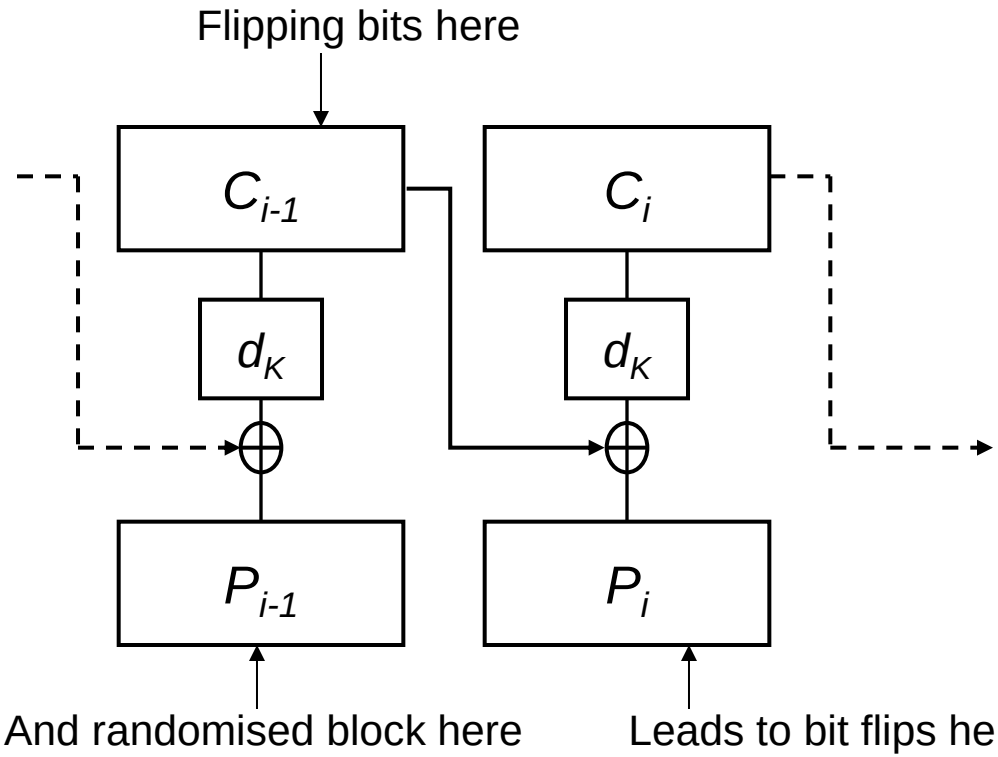
  to create a sequence of ciphertext blocks.
- Use this sequence as keystream (truncating last block as necessary).
- IND-CPA secure assuming block cipher is a PRF.
- Quadratic loss in security analysis; can be converted to linear loss by using stateful version of the scheme.

Motivating Stronger Security

- In CBC and CTR modes, an active adversary can *manipulate* ciphertexts.
  - For CTR mode, bit flipping in plaintext is trivial by performing bit flipping in the ciphertext.
  - Modify *c* to *c* XOR Δ to change the underlying plaintext from *p* to *p* XOR Δ .
  - CBC mode: see next slide.
  - Or create completely new ciphertexts from scratch?
    - A random string of bits of the right length is a valid ciphertext for *some* plaintext for both CBC and CTR modes!

Bit Flipping in CBC Mode

- Flipping bits in ciphertext block C_{i-1} leads to controlled changes in plaintext block P_i.
- But block P_{i-1} is randomised.



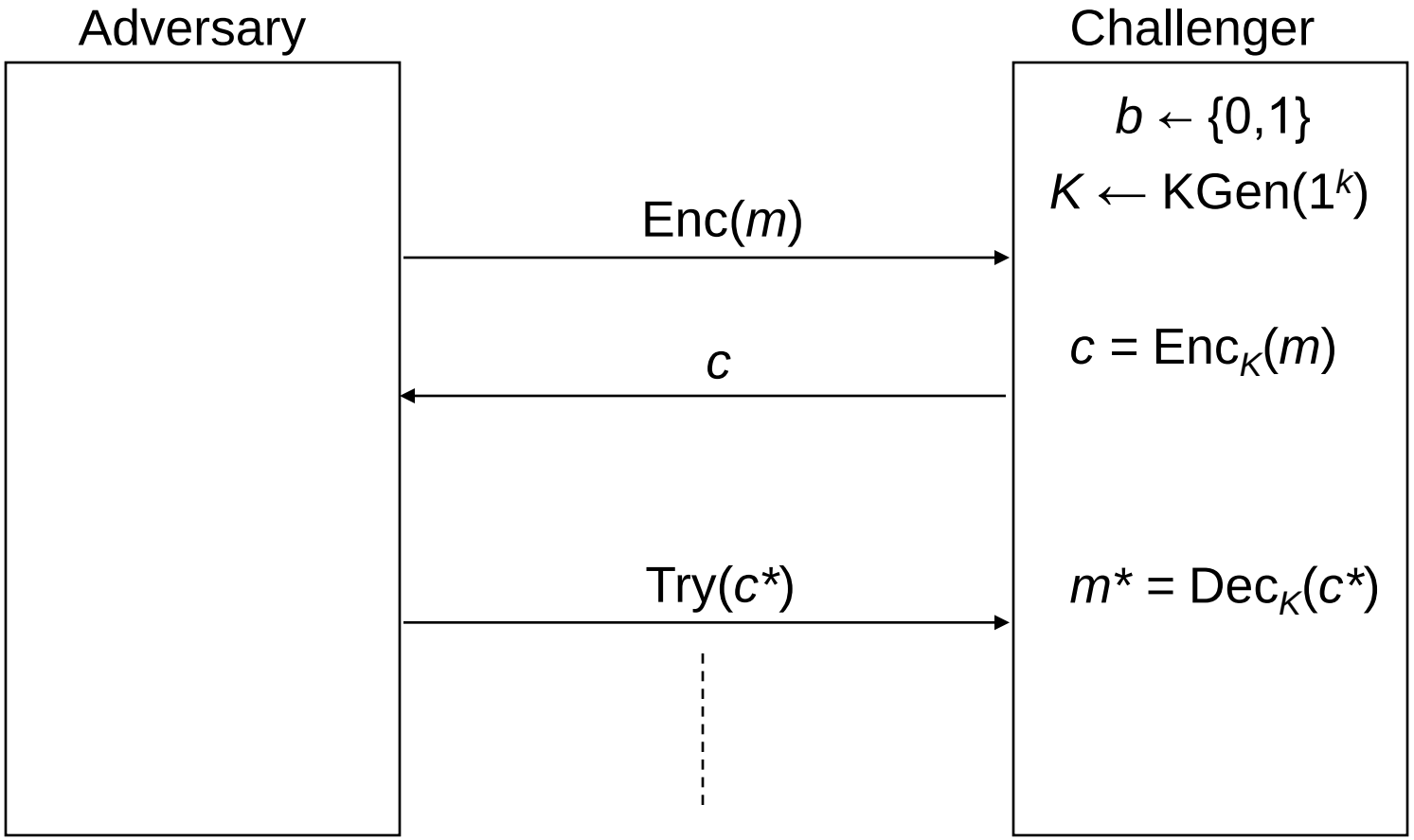
Motivating Stronger Security

- These kinds of attack do not break IND-CPA security, but are clearly undesirable for building secure channels.
  - Modified plaintext may result in wrong message being delivered to an application, or unpredictable behaviour at receiving application.
- We really want some kind of *non-malleable* encryption, guaranteeing integrity as well as confidentiality.
- Two basic security notions:
  - *integrity of plaintexts* and *integrity of ciphertexts*.

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 $\text{Enc}_K(m)$.
  - Try oracle takes any c* as input (and has no output).
  - Adversary's task is to submit *c** to its Try oracle such that $\text{Dec}_K(c^*)$ decrypts to message $m^* \neq \perp$ that is distinct from all *m* queried to its encryption oracle.
  - Hence adversary wins if it can create a "plaintext forgery".

INT-PTXT Security



Adversary wins if $m^* \neq m$ and $m^* \neq \perp$

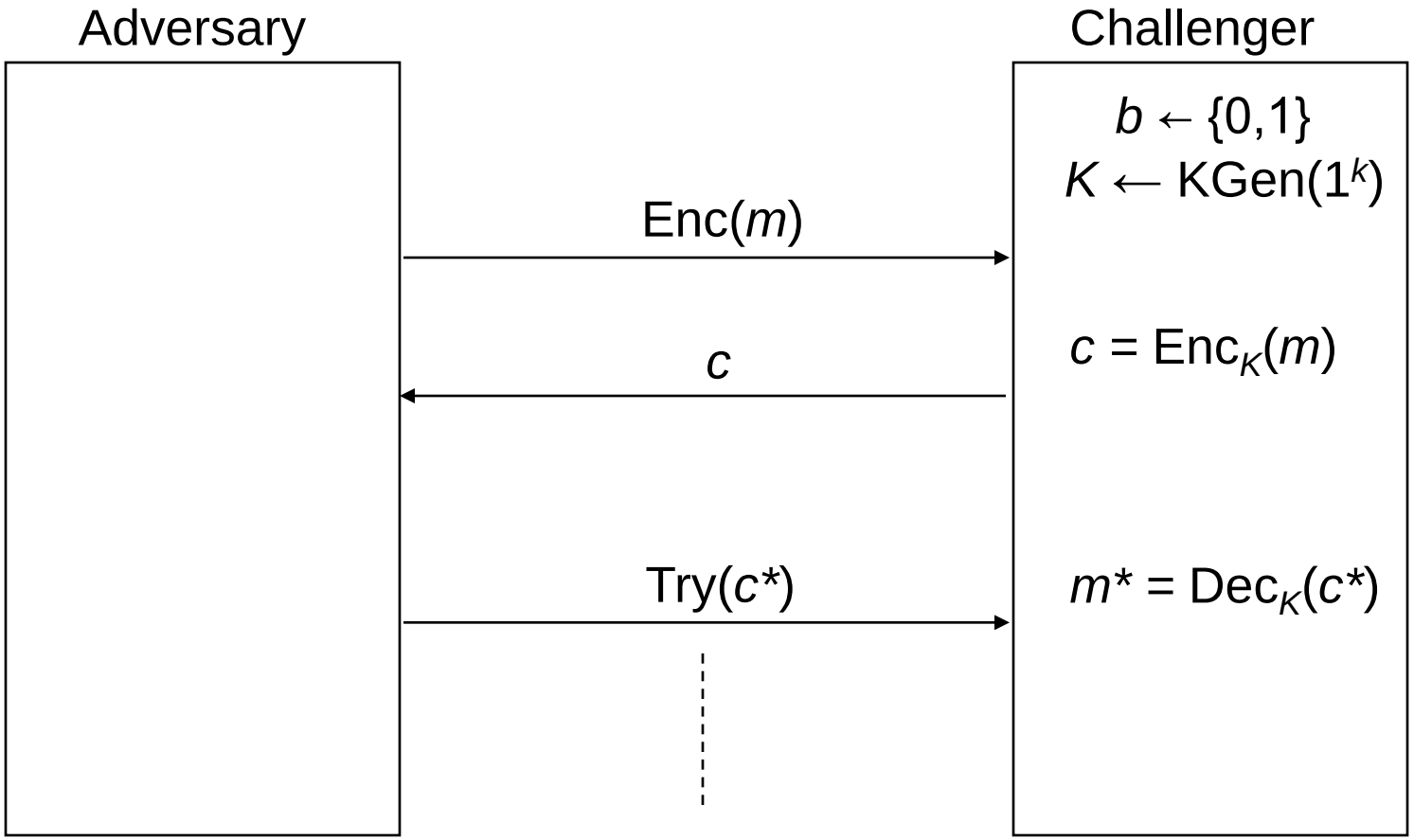
INT-PTXT Security

- INT-PTXT security:
  - An SE scheme is INT-PTXT secure if no such efficient adversary exists.
  - Clearly INT-PTXT security is a desirable property of an encryption scheme used for building a secure channel, as it prevents (plaintext) message injection.
  - Slightly different strength of security notion depending on:
    - whether adversary has one or many queries to Try; and
    - whether Try modified to output ciphertext validity.

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* queried to encryption oracle).
  - Hence win if adversary creates “ciphertext forgery”.
  - (Application to secure channels not immediately clear.)
- Clearly, INT-CTXT security implies INT-PTXT security.
- Quiz question: does CTR mode provide INT-CTXT or INT-PTXT security?

IND-CTXT Security



Adversary wins if $c^* \neq c$ and $m^* \neq \perp$

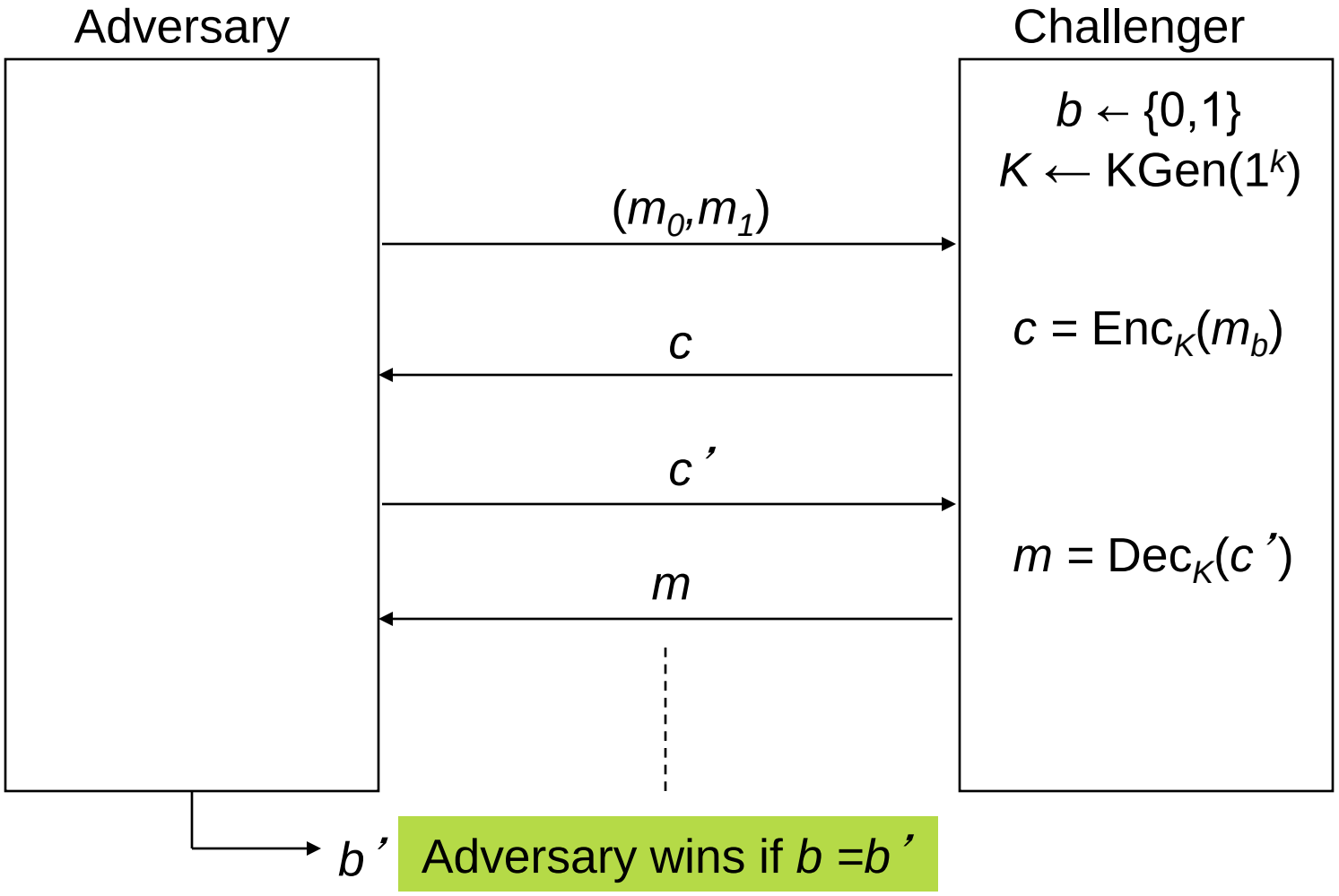
CCA Security

- We may also want to consider chosen-ciphertext attacks, in which the adversary can get ciphertexts of his choice decrypted.
  - Lazy reasoning: because this is what we did in the public key setting.
  - In extreme cases an attacker may actually have this capability in practice!
  - Or this capability may be *approximated* in practice.
    - Adversary may be able to observe the *reaction* of the decrypting party after processing an adversarially chosen ciphertext and thereby infer something about the plaintext.
    - Adversary may learn when decryption *fails*, and possibly the cause of failure, by analysing timing, error messages, or other behaviour.
    - This is particularly so for secure network protocols like IPsec, SSL/TLS, SSH.
    - It provides powerful attack opportunities!

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 $\mathrm{Dec}_K(c)$, 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!
  - Exercise for CTR mode.

IND-CCA Security



A Fundamental Relation [BN00]

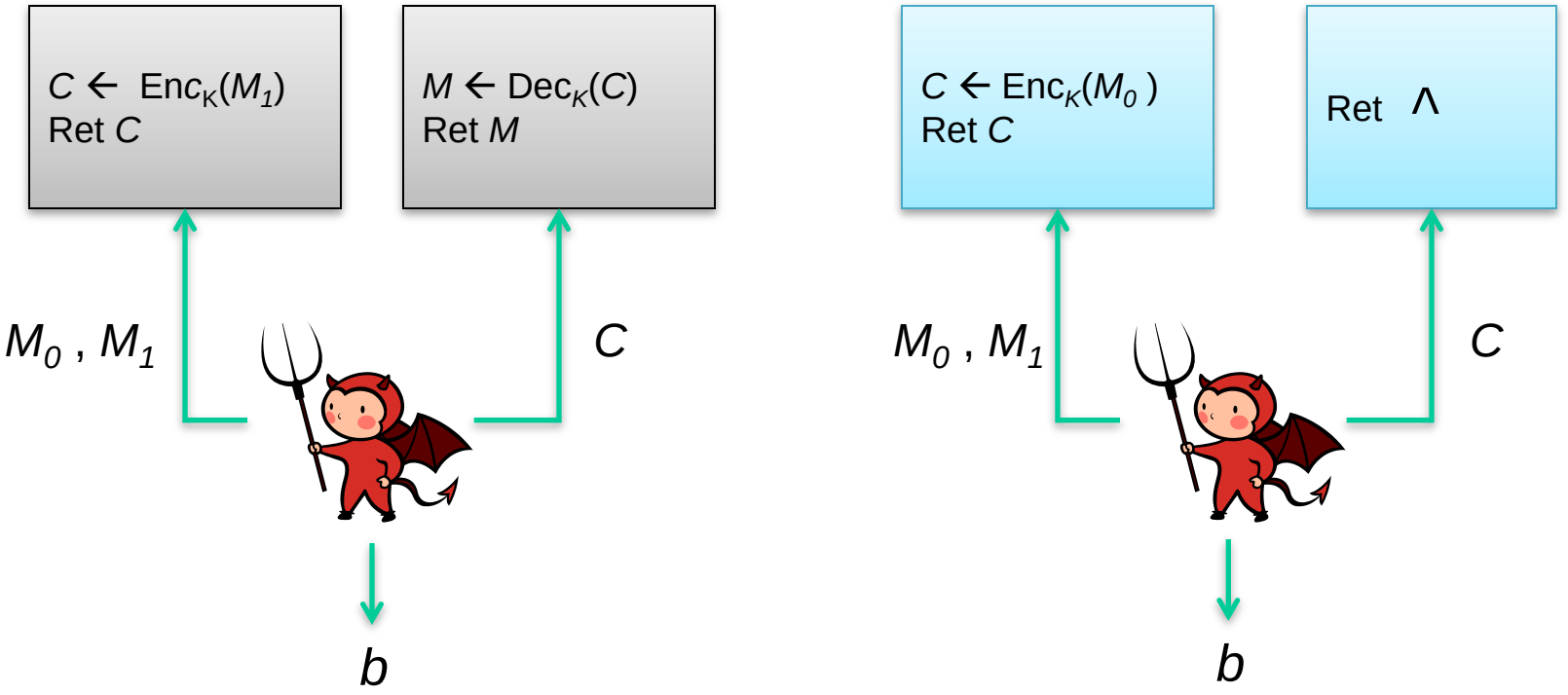
- IND-CPA + INT-CTXT → IND-CCA

- Proof intuition:
  - Game 0: IND-CCA security game against SE.
  - Game 1: replace decryption oracle with "⊥".
  - Games 0 and 1 identical unless related adversary wins INT-CTXT game.
  - Game 1 can be simulated perfectly by IND-CPA adversary (no decryption oracle to simulate any longer).
- NB: proof breaks down if decryption can return more than one error message.
  - See [BDPS13] for development of models and relations in this setting (which is important for practice).

Authenticated Encryption Security

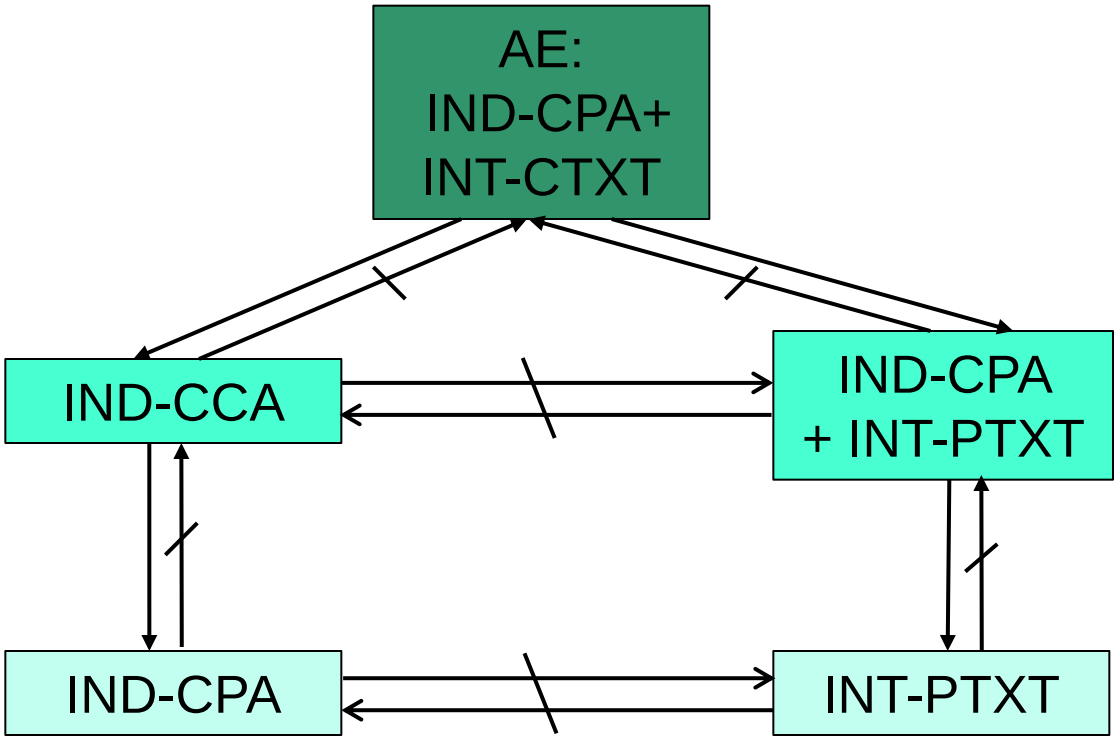
- We *define* AE := IND-CPA + INT-CTXT
- Often easier to prove IND-CPA and INT-CTXT separately than to prove IND-CCA directly.
- AE security has become the accepted security target for SE schemes.
  - In part (I think) because of the relation to IND-CCA security.
  - Note that IND-CPA + INT-PTXT does *not* imply IND-CCA.
    - Example separation: MAC-then-encrypt with redundant ciphertext bit.
  - Note also that IND-CPA + INT-CTXT is *strictly stronger* than IND-CCA.
    - Example separation: ?
    - In fact IND-CCA does not imply either of our integrity notions!

All-in-one Security Notion for AE



AE $\rightleftarrows$ IND-CPA + INT-CTXT

Relations Between SE Security Notions



AEAD

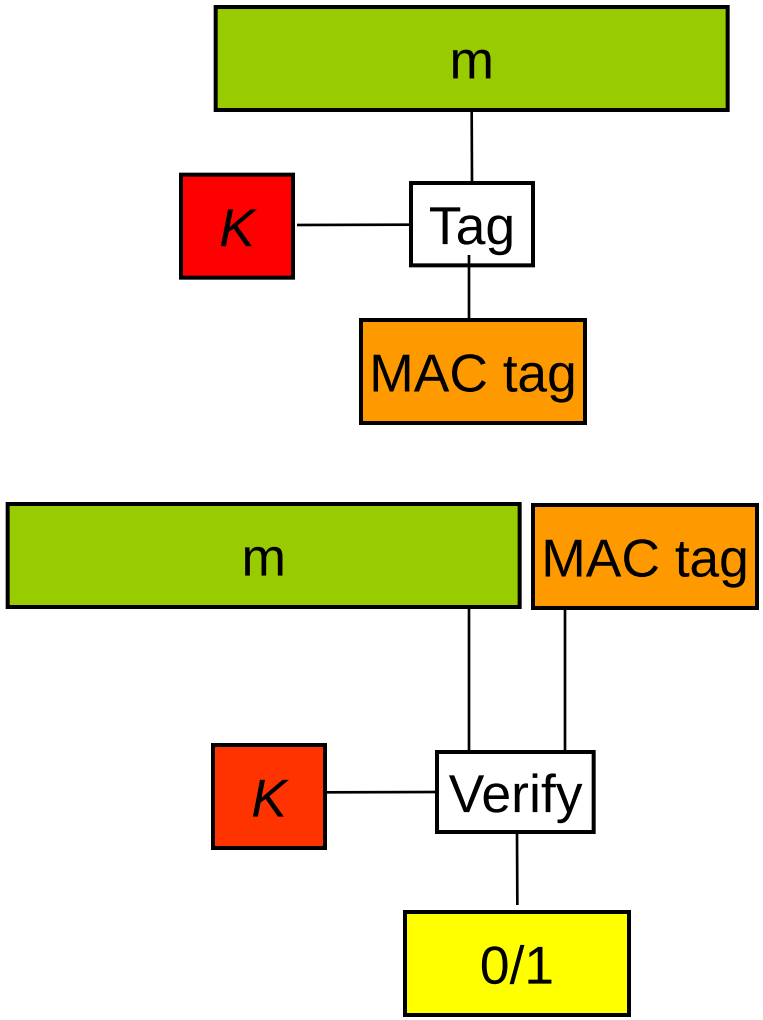
- AEAD = “AE with Associated Data”.
- Extension to AE allowing some data to be encrypted and remainder to be authenticated/integrity protected.
- Sample applications:
  - TLS Record Protocol data: header is integrity protected, rest of payload is encrypted *and* integrity protected.
  - IPsec ESPv3 protocol for encrypting IP payload and integrity protecting (selected) IP header fields.
- We omit security definition for AEAD.
  - Can define by extension of AE notion.

MACs

- Message Authentication Codes (MACs) provide authenticity/integrity protection for messages.
  - Symmetric analogue of a digital signature.
  - Important for achieving security for SE beyond IND-CPA.

- Syntax: MAC = (KGen,Tag,Verify).
  - KGen takes security parameter is input and outputs key *K.*
  - 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.*

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

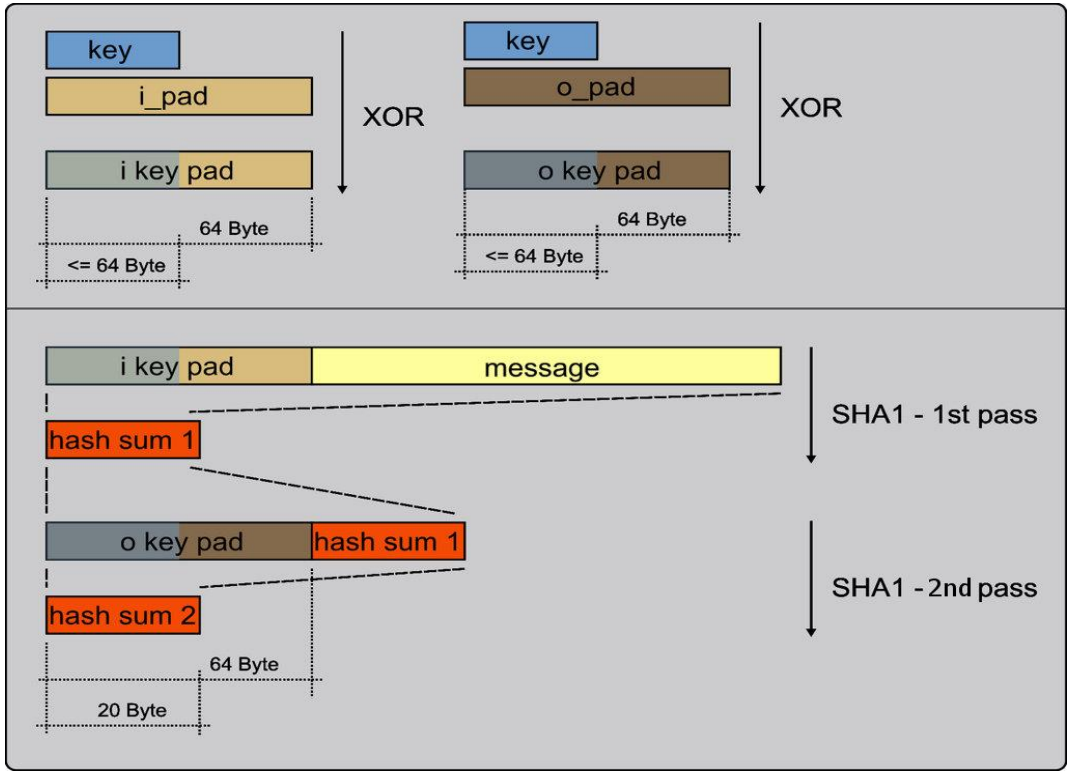
PRFs and MACs

- Unpredictability of PRF output means that a PRF *is* a MAC.
- More formally:
  - Tag(K,m) = $PRF_K(m)$; and
  - Verify(K,m,τ) outputs 1 if and only if $PRF_K(m) = \tau$.

- It's often assumed (implicitly or explicitly) that the security definition of a MAC is that it acts as a PRF.

HMAC

- HMAC is a general purpose method for building a MAC from a hash function H.
- Illustration for SHA-1:



Source: Wikipedia

HMAC

- HMAC is fairly efficient: cost of tag computation/verification is that of hashing message plus small overhead (3 x H's compression function).
  - But slow compared to more modern algorithms based on universal hashing like UMAC, poly1305.
- HMAC design and security proof in [BCK96].
  - PRF security relies (roughly) on H's compression function being a PRF and on collision resistance of H.
  - Refined analysis in [B06].
- HMAC standardised in RFC 2104.
  - Widely adopted in secure network protocols, e.g. SSL/TLS.
  - HMAC is an early triumph for provable security.

Generic Composition: EtM

- [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 (EtM): achieves AE security
  - Very easy proof:
  - INT-CTXT security follows from MAC on ciphertext;
  - IND-CPA security follows from IND-CPA security of base SE scheme.
  - Needs SUF-CMA MAC.
    - As provided by PRF-based construction, HMAC, etc.

Generic Composition: E&M

- Encrypt-and-MAC (E&M): **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].
  - At least need MAC to not leak plaintext information (e.g. PRF assumption).

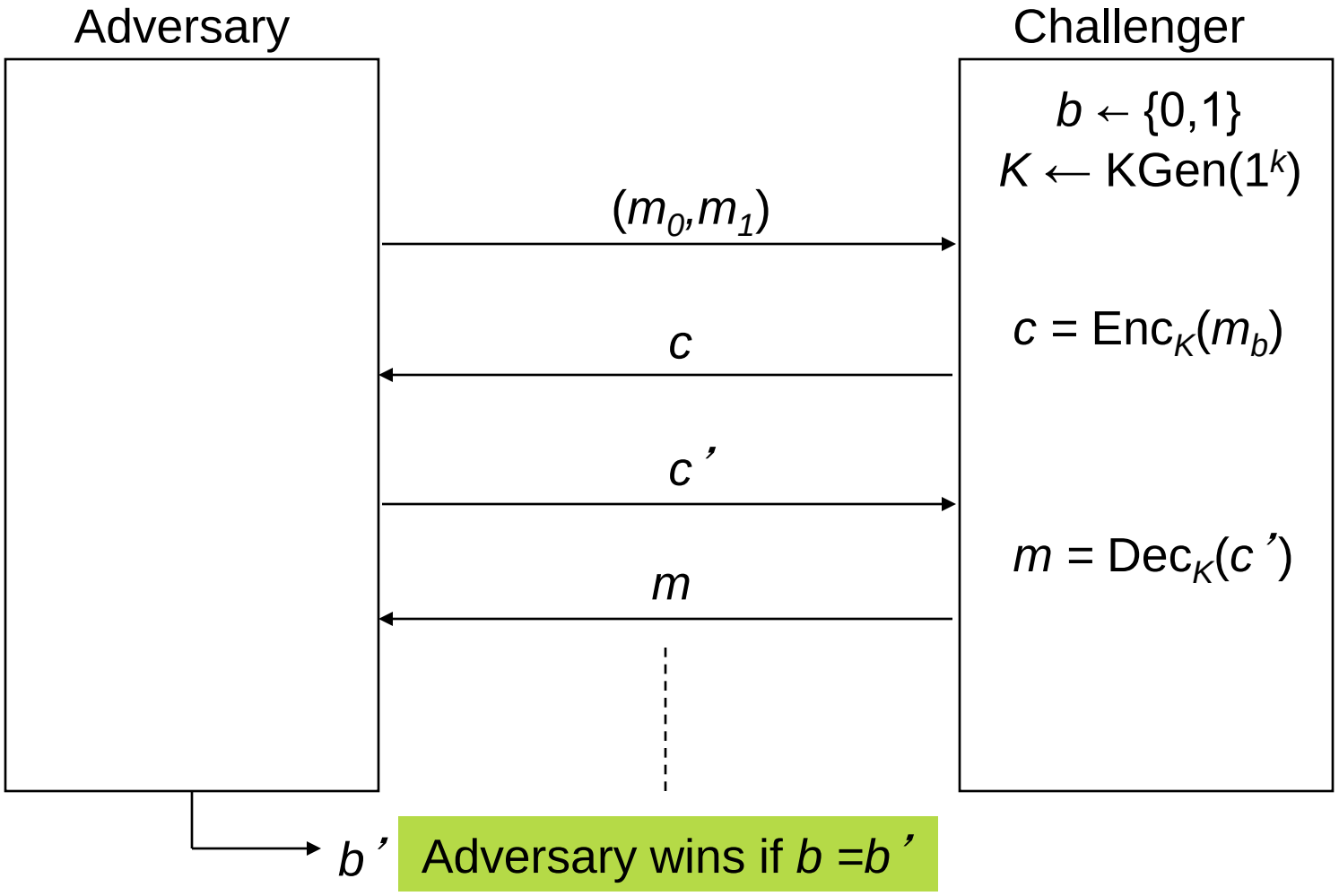
Generic Composition: MtE

- MAC-then-Encrypt (MtE): **Not** CCA secure in general.
  - Construct separating example (hint: redundant bits again).
  - But easy to show IND-CPA and INT-PTXT security for this composition.
    - Good enough for secure channel applications?
  - Extension of [K01] shows MtE is IND-CCA secure when encryption scheme is CBC mode or secure stream cipher.
  - Proof for CBC needs SPRP property for block cipher.
  - Real instantiations are rarely pure MtE (more later).

Some Philosophy

- Models are just models.
- And reality is hard to model.
- General approach is to build conservative models with strong adversaries.
- Realising assumed adversarial capabilities may be hard in practice.
  - Consequently, it can be hard to convince practitioners that your chosen plaintext distinguishing attack should be of serious concern (to them)!
  - They tend to need to see plaintext and a working exploit.
  - Countering this: attacks only get better (worse!) with time (examples to follow).
  - Every practitioner seems to need to learn this the hard way.

IND-CCA Security (recap)



Discussion

Now look back at IND-CCA security model.

- What, if anything, is overkill?
- What, if anything, is missing?

Lecture 2
Kenny Paterson

Bar-Ilan Winter School on Symmetric Cryptography

Royal Holloway
University of London

Information Security Group

Lecture 2 Outline

- **Why integrity protection really matters: IPsec case study.**
- Why details really matter: predictable IVs, TLS, and the BEAST.

Introduction to IPsec

- IPsec provides security at the IP layer.
  - IP packets get encrypted and/or integrity protected.
- Defined in IETF RFCs 2401–2412 (v2) and 4301-4309 (v3).
- Implemented in all major OSes and in networking hardware.
- Applications:
  - Virtual Private Networking.
  - Remote Access Solutions.
  - Protection of inter-network management data in UMTS.

IPsec Basic Features

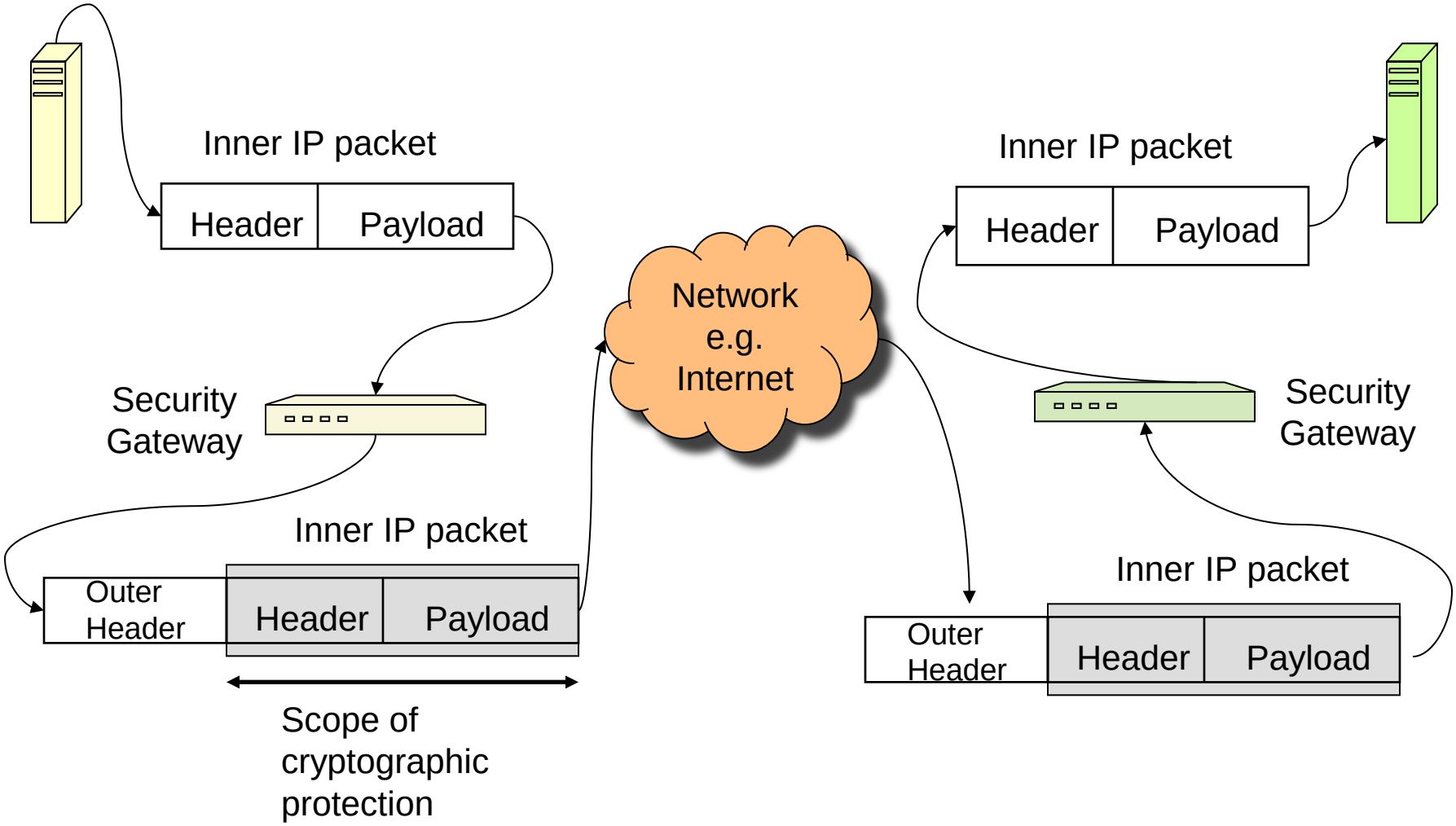
- IPsec provides two basic modes of use: transport and **tunnel**.

- IPsec provides authentication/integrity protection and/or confidentiality services for data.
  - AH and **ESP** protocols.

IPsec Tunnel Mode

- Cryptographic protection for entire IP packet.
- Entire packet plus security fields encapsulated as payload of new ‘outer’ IP packet.

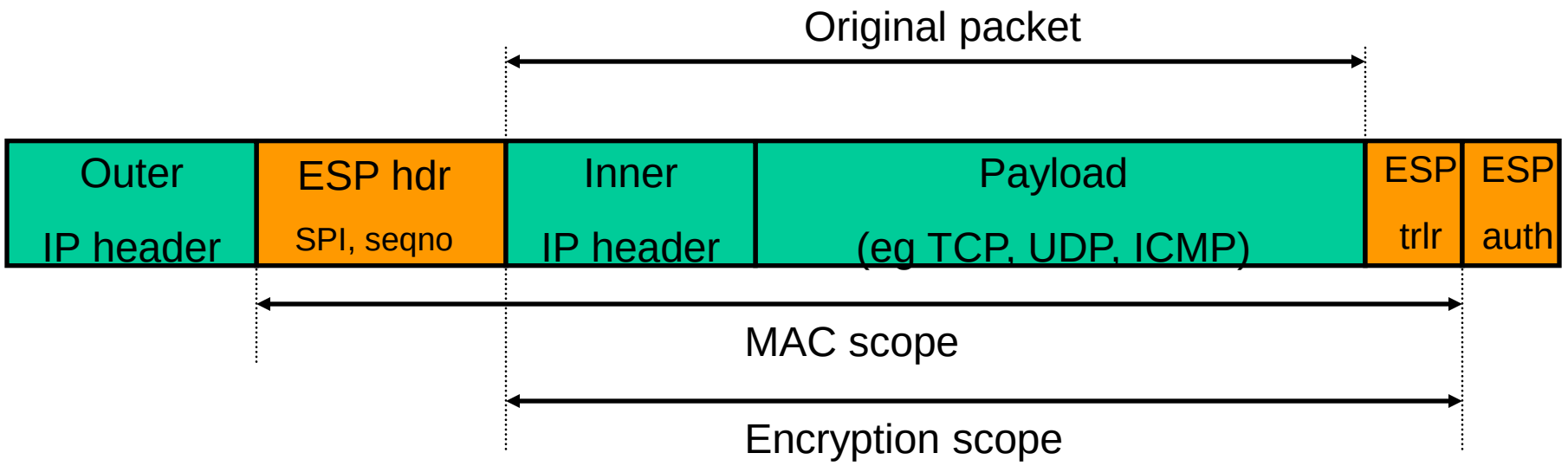
IPsec Tunnel Mode Deployment



ESP Protocol

- ESP = Encapsulating Security Protocol.
  - v1, v2, v3 in IETF RFCs 1827, 2406, 4303.
  - IPsec's "encryption workhorse".
- ESP provides one or both of:
  - Confidentiality for packet/payload (v1, v2, v3).
  - Integrity protection for packet/payload (v2, v3).
- ESP uses symmetric encryption and MACs.
  - Usually CBC mode of block cipher for encryption.
    - With random, per packet IVs.
  - HMAC-SHA1 or HMAC-MD5 for integrity protection.

ESP in Tunnel Mode



When both MAC and encryption are used,
IPsec employs an EtM construction

History of Encryption in IPsec

- ESPv1 (1995) provided no integrity protection.
  - Reliant on separate AH protocol to provide this.
  - Bellovin [B97] sketched a series of attacks on ESPv1 without AH.
  - Limited plaintext recovery from TCP segments, using 2^{24} chosen plaintexts.
  - Certainly breaks IPsec in IND-CCA security model.
  - Theoretically interesting, but no attacks demonstrated to work in practice.
  - Sufficiently serious to influence development of v2 RFCs.

Integrity protection and ESPv2

- IETF response to Bellovin attacks:
  - ESPv2 (1998) includes integrity protection as an option.
  - But implementations **must** still support “encryption-only” mode.
- ESPv2 represents a compromise between improving security and maintaining backwards-compatibility.
  - This is very common in real-world cryptography!

Integrity protection and ESPv3

- ESPv3 (2005):
  - Still allows encryption-only ESP.
  - But no longer *requires* support for encryption-only.
  - Gives strong warnings about Bellovin-Wagner attack and refers to theoretical cryptography literature to motivate need to use integrity protection.
  - *"ESP allows encryption-only … because this may offer considerably better performance and still provide adequate security, e.g., when higher layer authentication/integrity protection is offered independently."*

IPsec in **Theory** and Practice

- Back in the 2000's, the theoretical cryptography community was well aware of the need to carefully combine integrity protection with encryption.
  - To prevent active attacks against encryption.
- Already plenty of high-profile, real-world examples.
  - Kerberosv4, WEP, SSHv1,…
- It was also well-known amongst IPsec experts that encryption-only configurations should be avoided.
  - Clear warnings against their use in the RFCs.
- So was there really any problem here?

IPsec in Theory and **Practice**

- From an historical administration guide from a well-known vendor:

> *"If you require data confidentiality only in your IPSec tunnel implementation, you should use ESP without authentication. By leaving off the authentication service, you gain some performance speed but lose the authentication service."*

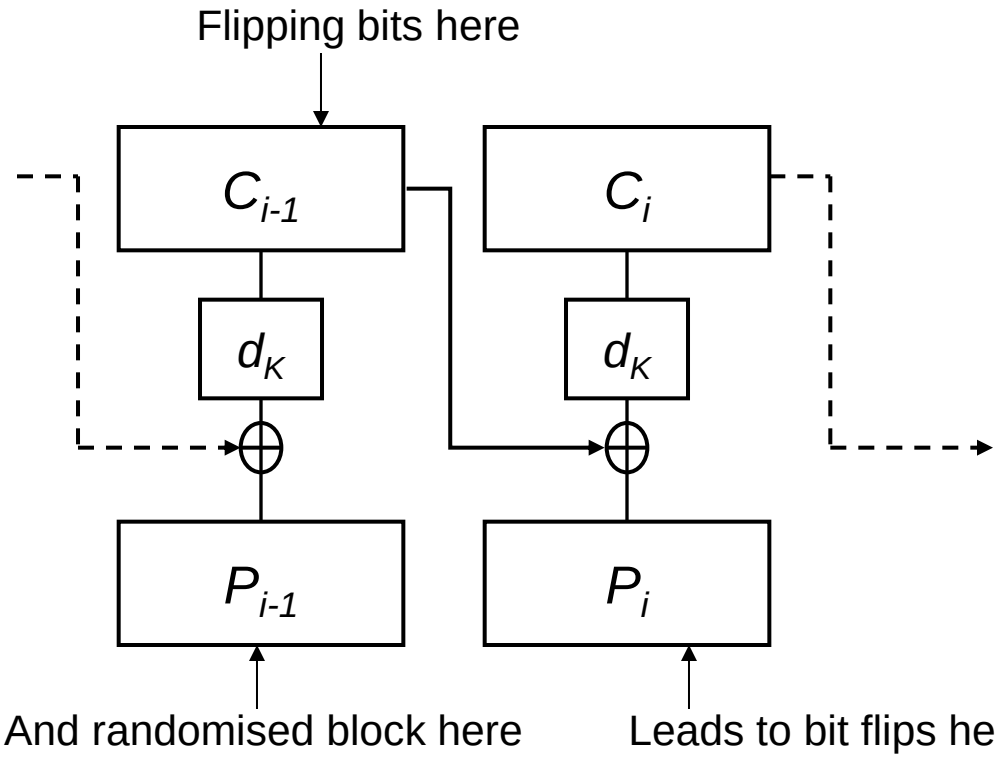
http://www.cisco.com/en/US/docs/security/security_management/vms/router_mc/1.3.x/user/guide/U13_bldg.html#wp1068306, cicra 2008).

Attacking Encryption-only ESP

- If we want to demonstrate beyond doubt that encryption-only ESP is *disastrously* weak, we need to:
  - Produce attacks that consume reasonable resources.
  - *Implement* attacks that are as *realistic* as possible
    - Operating under normal network conditions.
    - Ideally, ciphertext-only attacks.
    - Hand over plaintext in a demo!

Reminder: Bit Flipping in CBC Mode

- Flipping bits in ciphertext block C_{i-1} leads to controlled changes in plaintext block P_i.
- But block P_{i-1} is randomised.

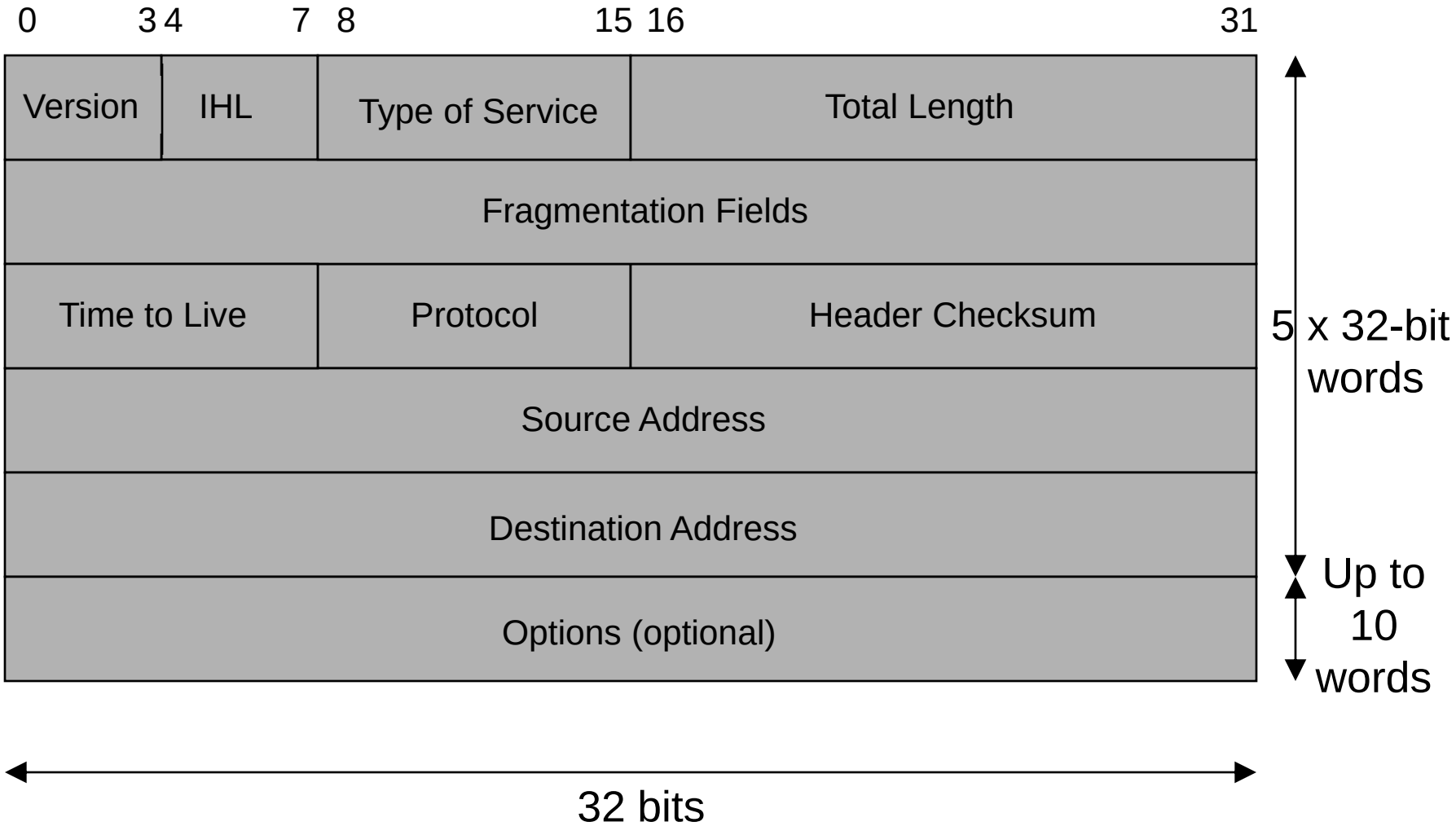


Attacking Linux ESP

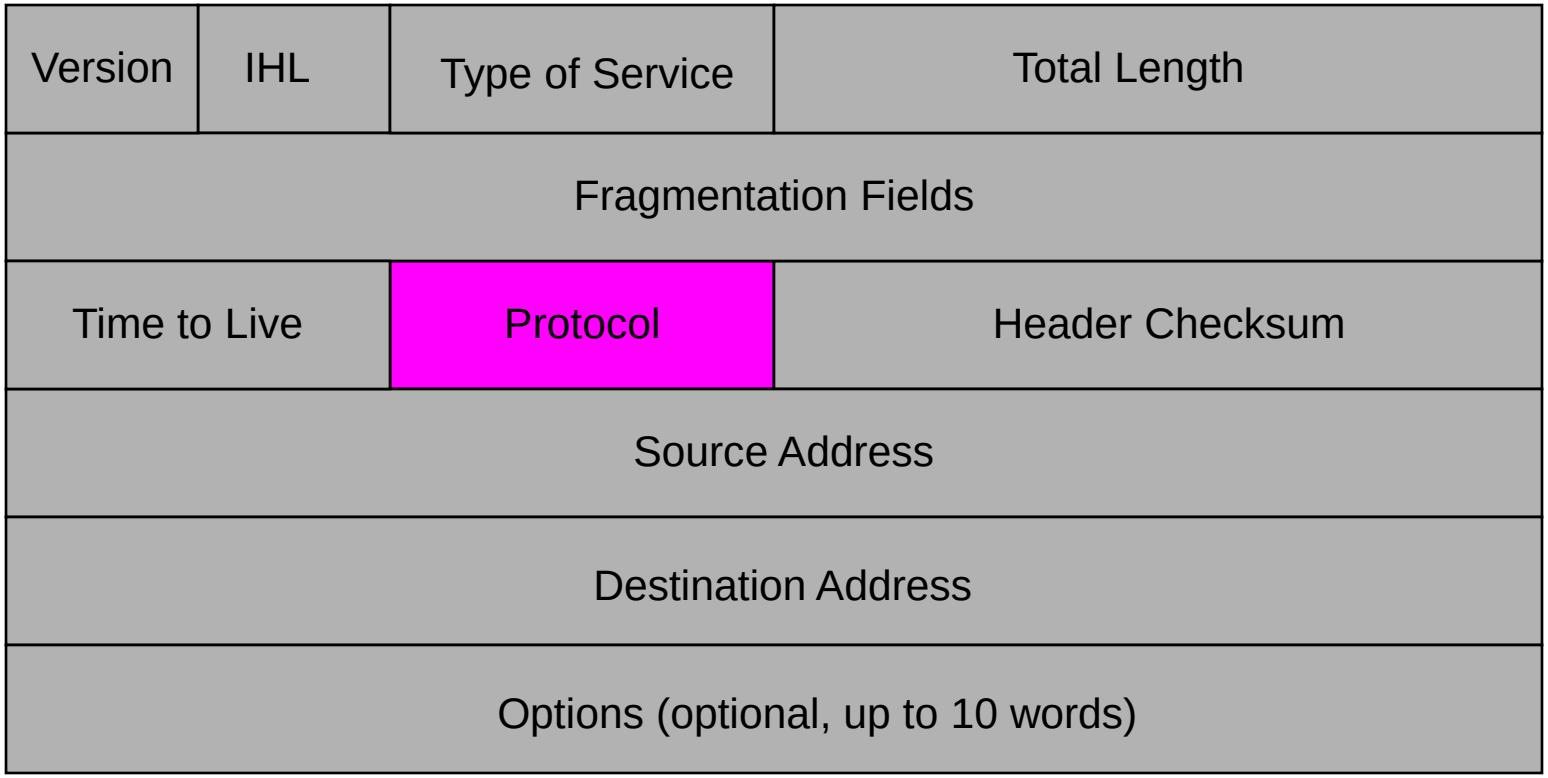
[PY06]:

- Three different (but related) attacks on Linux kernel implementation of encryption-only ESP in tunnel mode.
- Exploit bit flipping weakness of CBC mode encryption.
  - Flipping creates modified headers for inner packets that produce “unusual” effects when processed
- Resulting in either error messages or in packet re-direction for inner packets
  - Error messages are carried by ICMP protocol and reveal (some) plaintext data.
  - Packet redirection can send inner packet to attacker’s machine.

IP Header Format



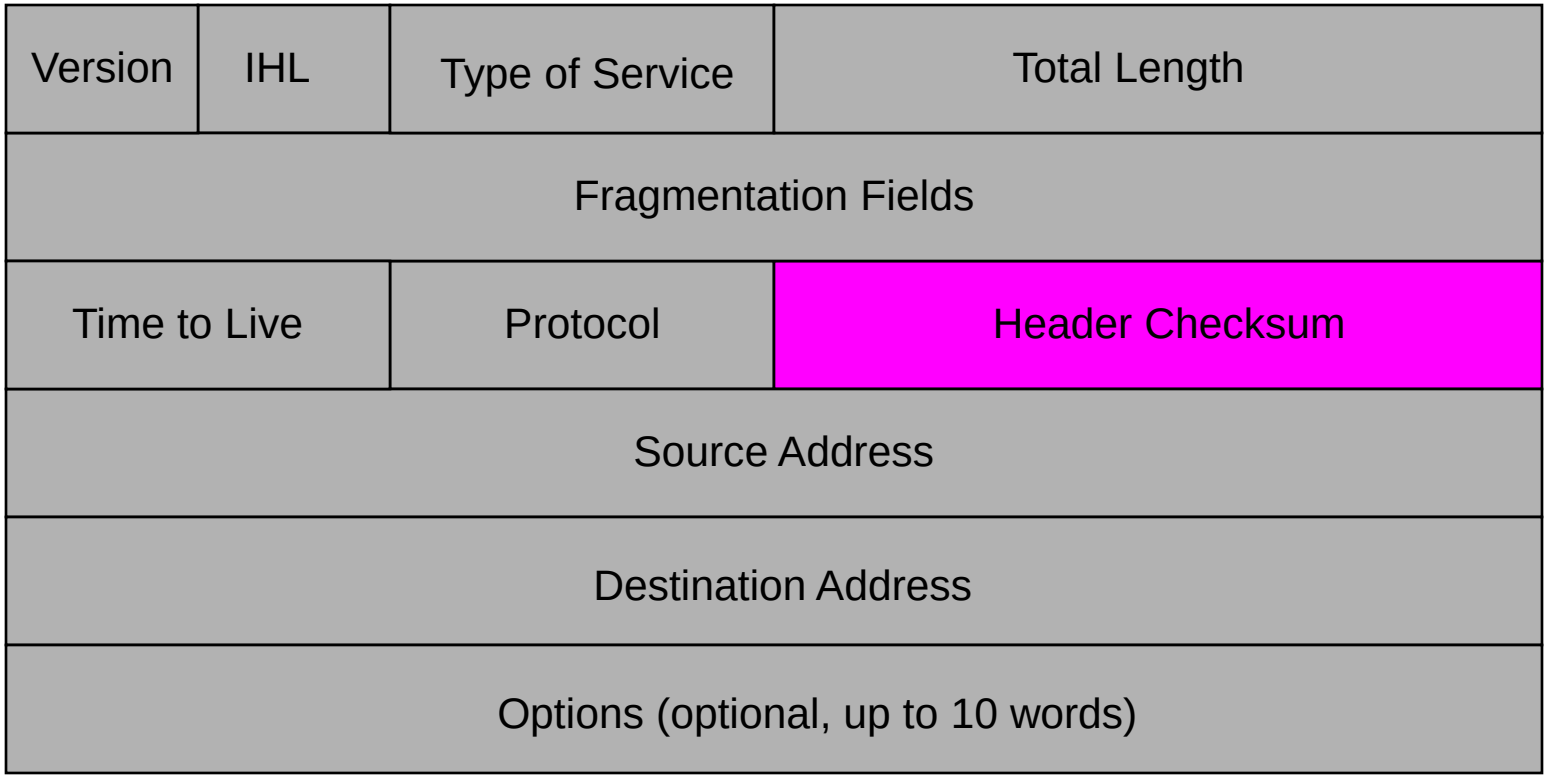
IP Header Format



Protocol field (8 bits):

- Indicates upper layer protocol in IP payload.
- Possible values are dependent on IP implementation and protocols it supports.
- Typical values: 0x01 for ICMP, 0x06 for TCP, 0x17 for UDP.

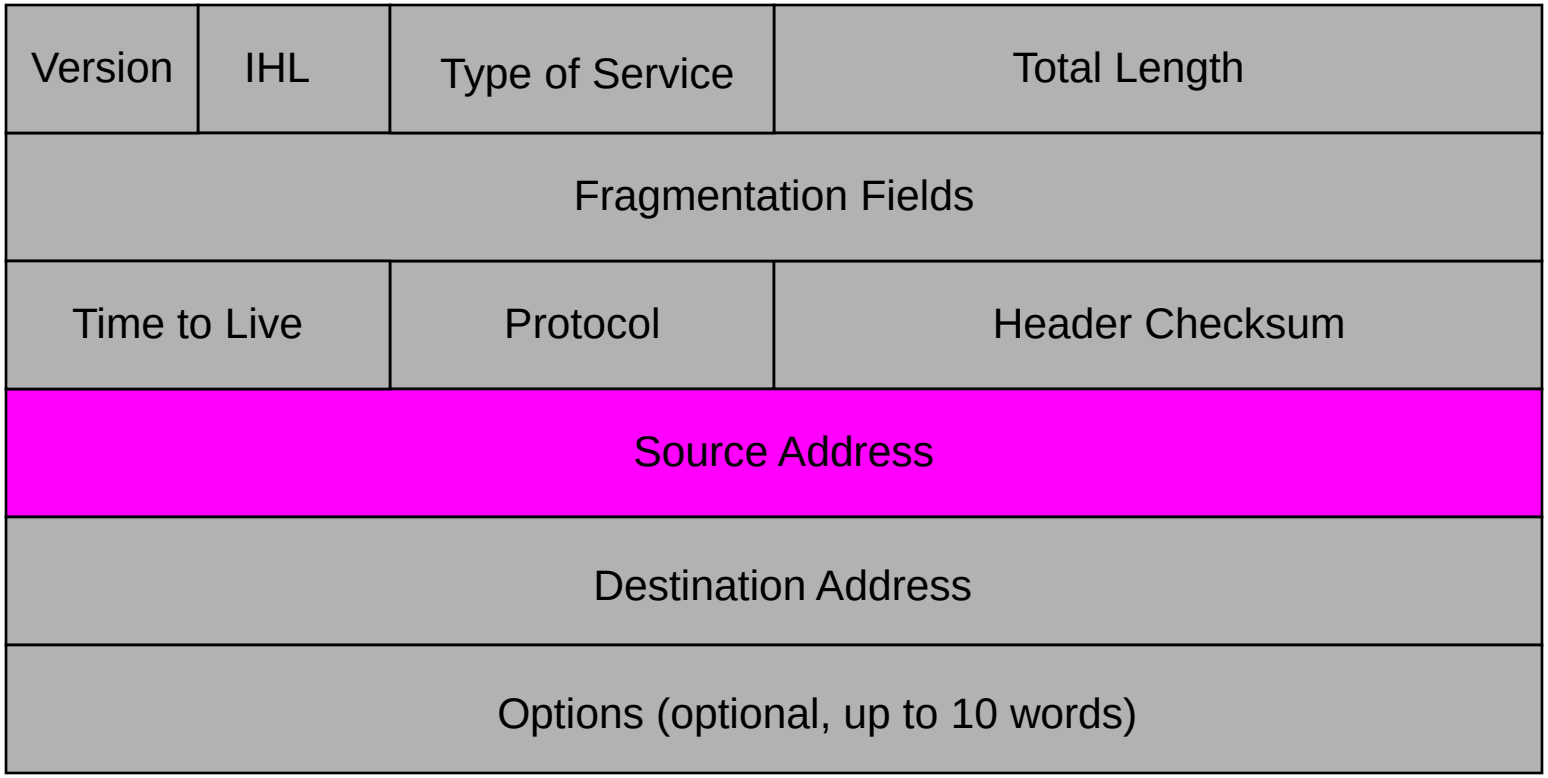
IP Header Format



Header checksum (16 bits):

- 1's complement sum of 16 bit words in header (inc. any options).
- Incorrect checksum leads to datagram being silently dropped.
- Provides error detection for IP headers.

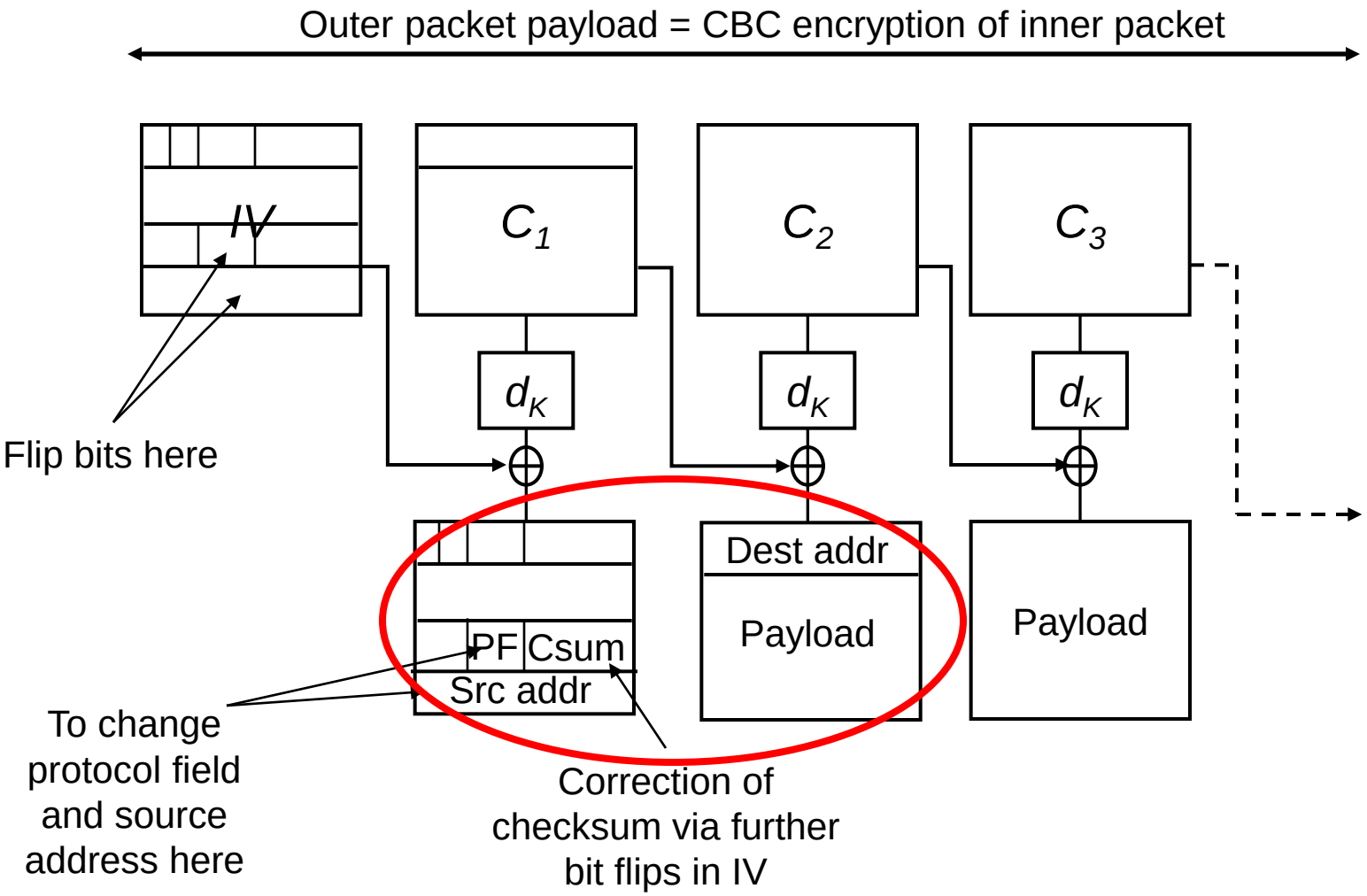
IP Header Format



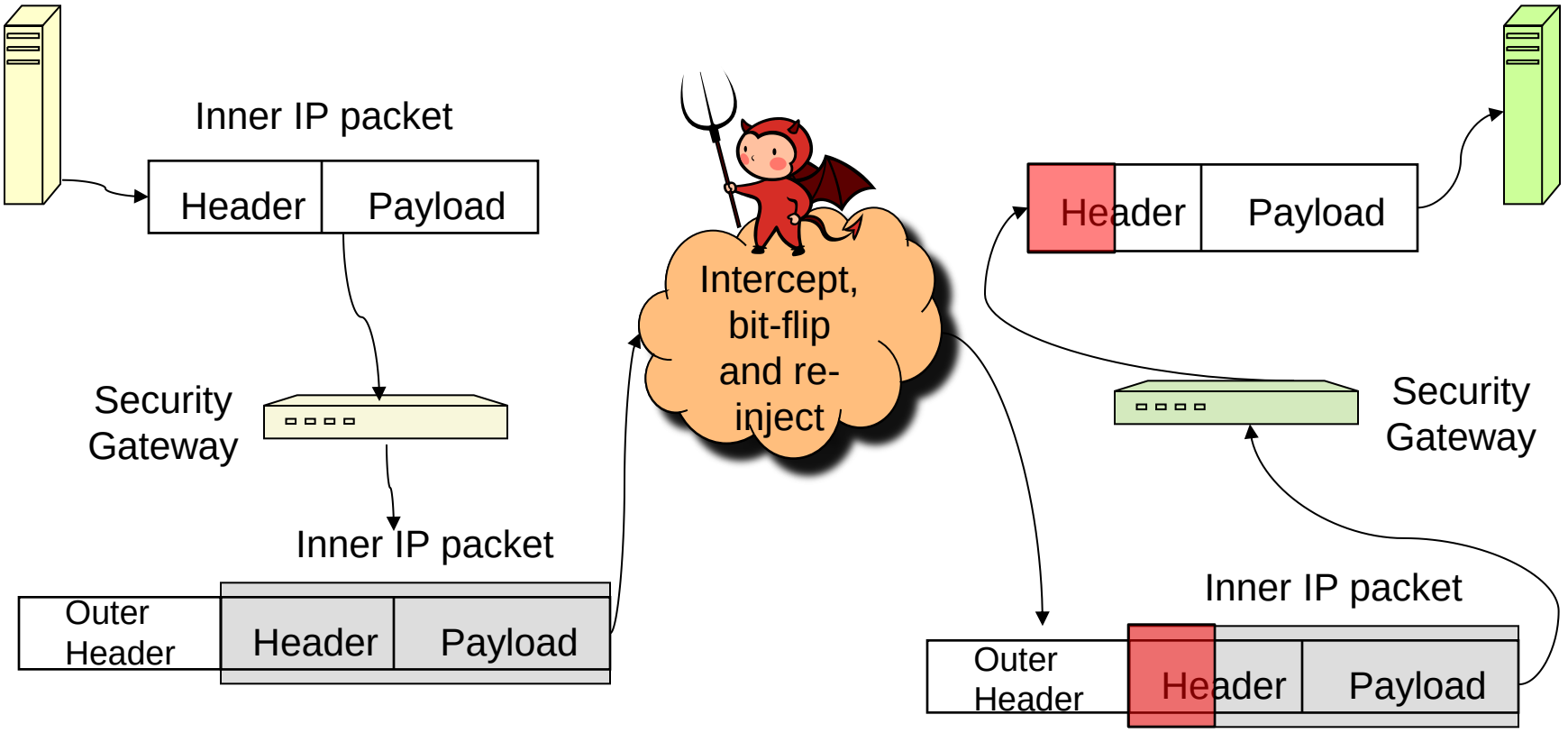
Source Address (32 bits):

- Contains the IP address of the host originating the datagram.
- Needed so any replies or error messages can be delivered back to source.

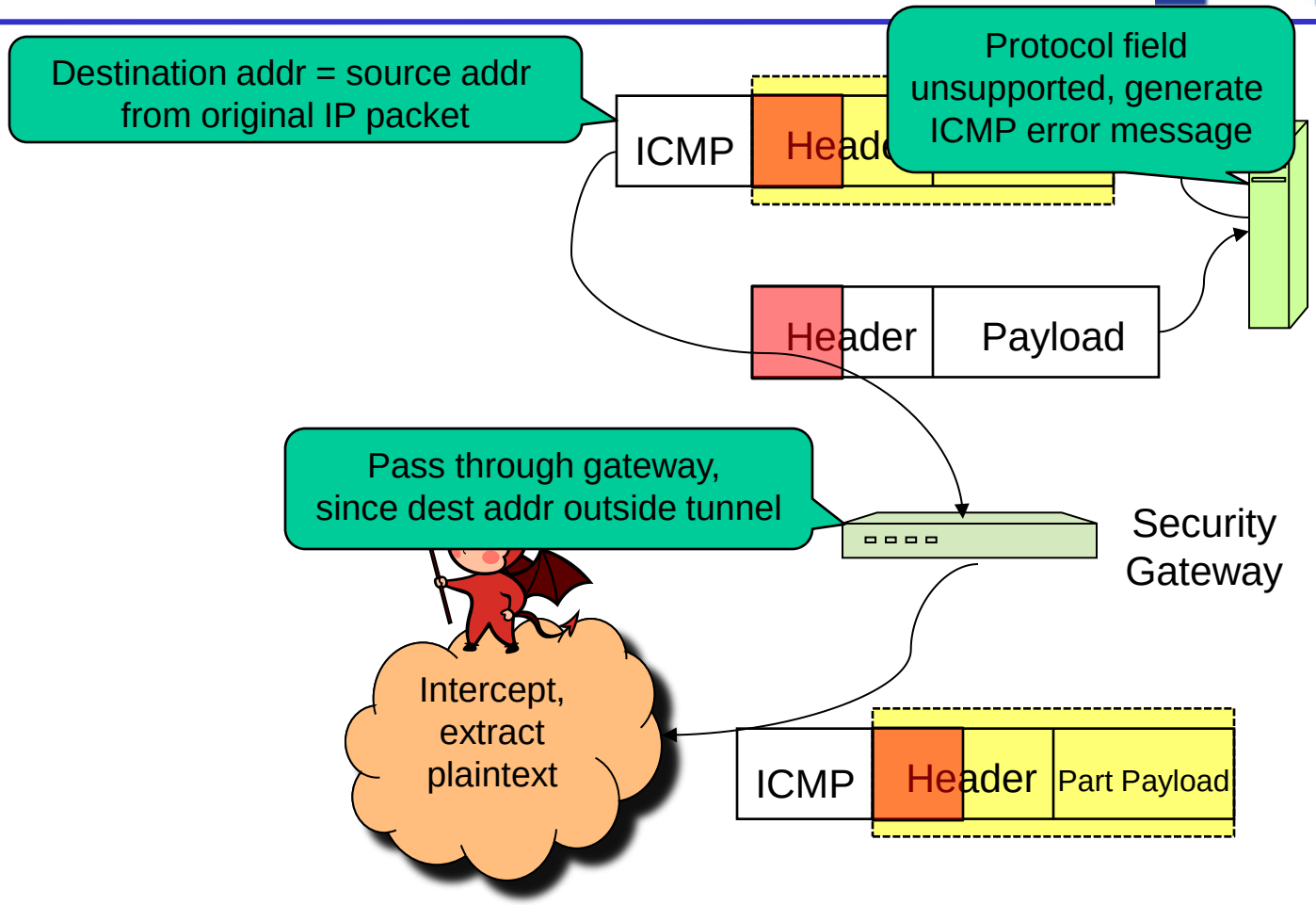
Example Attack on ESP



Attack Visualisation



Attack Visualisation



The Attack in Words

- Attacker intercepts packet, does bit flipping needed to manipulate protocol field and source address, and to correct checksum.
  - Can do better than random bit flipping for checksum.
- Attacker than injects modified datagram into network.
- Inner packet decrypted by gateway and forwarded to end-host.
- End-host generates ICMP “protocol unreachable” message in response to modified protocol field in header.

The Attack in Words

- ICMP payload carries inner packet header and 528 bytes of inner packet's payload.
  - Payload now in plaintext form!
  - ICMP message is sent to host indicated in source address
  - And we have modified this address so that ICMP message does not pass through IPsec tunnel.
- Attacker intercepts ICMP message to get plaintext bytes.
- These ideas were used in [PY06] to build an attack client that can efficiently extract *all* plaintext from an IPsec encryption-only tunnel.

Characteristics of IPsec Attacks

- The attacks recover plaintext (i.e. contents of inner datagrams), but not encryption keys.
- The attacks are efficient.
  - Even against triple DES or AES.
  - Can be run in near real-time against an IPsec tunnel.
- Attacks are ciphertext-only.
- The attacks do not require special operating conditions.
  - Attacker needs to capture packets from network, inject packets into network.
  - But they need ability to monitor gateway for ICMP responses.
- All three attacks worked in practice against Linux implementation of IPsec.
  - Attacks fail if post-processing policy checks specified in RFCs are properly implemented.
  - But Linux did not implement these ☺

Attacking Encryption-only ESP

- Some reactions to attacks in [PY06]:
  - *"…the possibility of active attacks on unauthenticated but encrypted ESP packets is well known, and we advise against such use in the most recent set of IPsec documents. These documents have been approved for publication by the IESG and are in the queue to be published as RFCs. As a result, no further, substantiative changes will be made."*
  - *"This is all very well understood among the IPSec community, and is not news."*
  - *"I think the spec is clear about the dangers of encryption without authentication. If anyone built implementations that negotiate encryption without authentication, then maybe they weren't paying attention closely enough."*
- So is there really any problem if the RFCs still allow use of encryption-only ESP?

Why the Attacks Matter(ed)

- Recall:

> *"If you require data confidentiality only in your IPSec tunnel implementation, you should use ESP without authentication. By leaving off the authentication service, you gain some performance speed but lose the authentication service."*

Why the Attacks Matter(ed)

- Also recall:

> *“‘ESP allows encryption-only … because this may offer considerably better performance and still provide adequate security, e.g., when higher layer authentication/integrity protection is offered independently.”*

- But these attacks work without any higher layer even getting to see the data.
- So no higher layer integrity protection can stop them!

Follow-up Work

- [DP07]:
  - Attacks against any RFC-compliant implementation of encryption-only ESP.

- [DP10]:
  - Extending [DP07] attacks to the situation where integrity protection via AH is applied *before* encryption.
  - Breaking all MtE configurations of IPsec!
  - Rendering AH pretty useless.
    - Since ESP offers integrity too, though with different scope of protection.

Lessons

- **Encryption on its own does not provide confidentiality in the face of active attacks.**
  - IND-CPA security is not enough.
  - AE security would have prevented the attacks.
- Attacks can exploit interaction between crypto layer and the layer(s) above.
  - In this case, the layer above was IP because of protocol tunnelling.
  - Information leakage from IP layer error messages.
- Practical attacks are needed to convince “experts” of the need for change.

Lecture 2 Outline

- Why integrity protection really matters: IPsec case study.
- **Why details really matter: predictable IVs, TLS, and the BEAST.**

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.

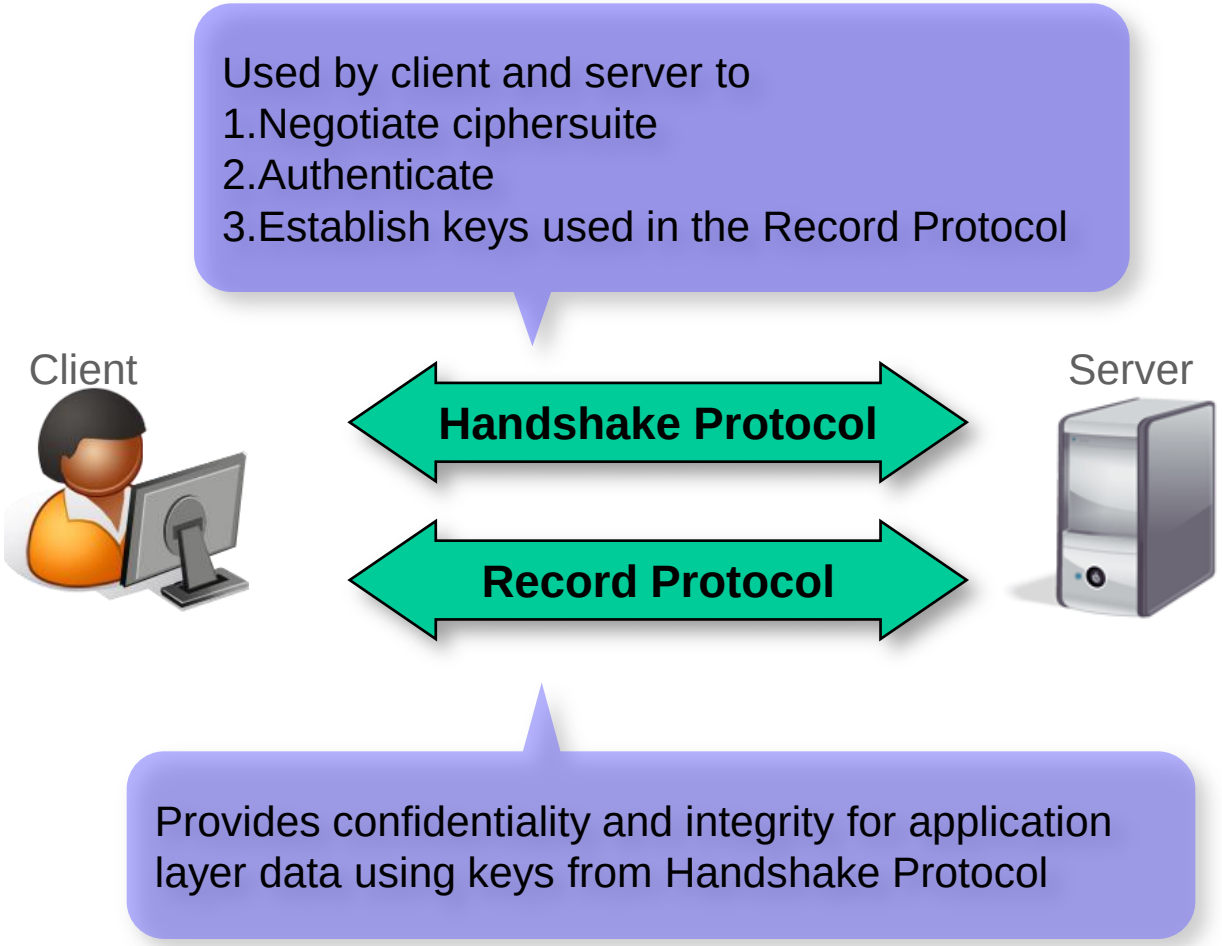
- TLS = Transport Layer Security.
  - IETF-standardised version of SSL.
  - TLS 1.0 in RFC 2246 (1999).
  - TLS 1.1 in RFC 4346 (2006).
  - TLS 1.2 in RFC 5246 (2008).

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 channel protocol of choice.
  - Used by hundreds of millions of people and devices every day.

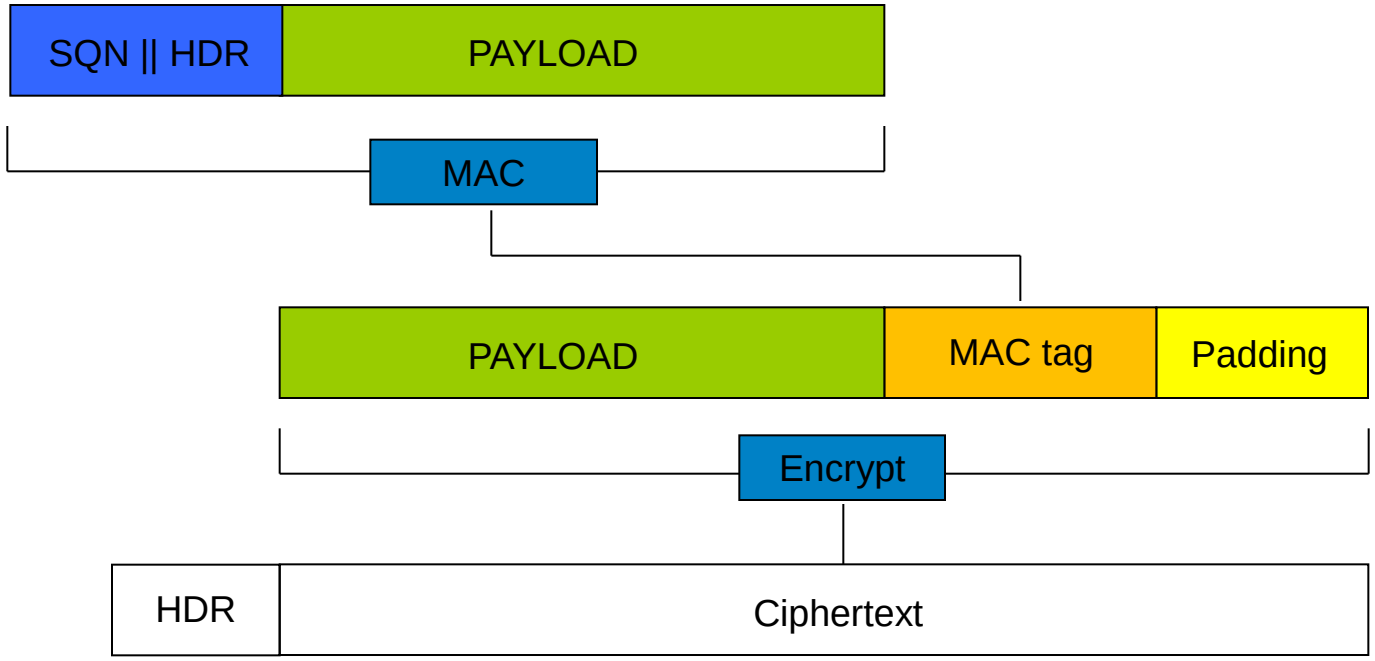
Simplified View of TLS



TLS Record Protocol

- TLS Record Protocol provides:
  - Data origin authentication and integrity using a MAC.
  - Confidentiality using a symmetric encryption algorithm.
  - Anti-replay service 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 is no compression.
- Calculate MAC on SQN, 5-byte HDR, and PAYLOAD.
- Append MAC to PAYLOAD.
- Pad (if needed by encryption mode), then encrypt.
- Prepend HDR 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

Receiver processing steps reverses these steps:

1. Receive message, of length specified in HDR.
2. Decrypt.
3. Remove padding to recover PAYLOAD and MAC.
4. Check MAC on SQN, HDR, PAYLOAD.
5. (Decompress PAYLOAD.)
6. Pass PAYLOAD to upper layer (no fragment reassembly).

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

All of these are *fatal* errors, leading to error message and connection termination.

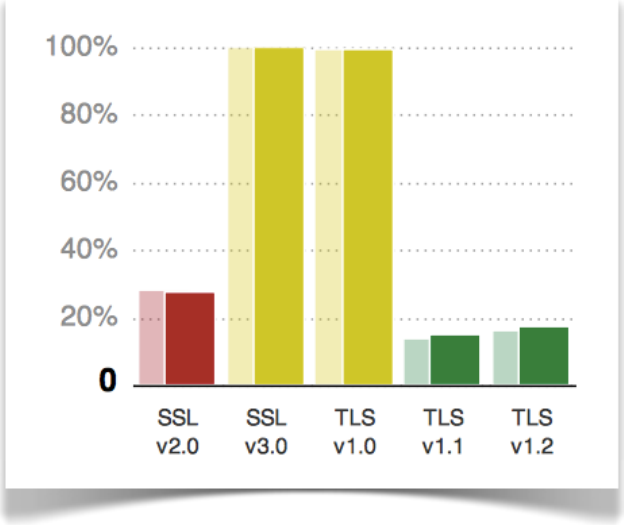
TLS Sequence Numbers

- SQN is 8 bytes in size and is incremented for each new Record Protocol message.
- SQN not transmitted as part of message.
  - Each end of connection maintains its own view of the current value of SQN.
  - TLS is reliant on TCP to deliver messages in order.
- Using wrong SQN leads failure of MAC verification
  - A fatal error leading to TLS connection termination.
- Use of SQN creates a stateful encryption scheme.
  - Intention is to prevent replay, insertion, reordering attacks.
  - Order in the TLS secure channel matters.
  - We have not yet introduced security notions for this (see later).

AE in the TLS Record Protocol

- TLS 1.2 additionally supports authenticated encryption modes.
  - AES-GCM in RFC 5288
  - AES-CCM in RFC 6655
- Support for TLS 1.2 recently added in major browsers.
  - Mostly as a consequence of recent attacks.
- However, TLS 1.2 is only now becoming supported in servers.

SSL Pulse: Webserver TLS support



Browser TLS support (out-of-the-box, Nov. 2013)


TLS v1.2


TLS 1.2


TLS v1.2


TLS v1.2

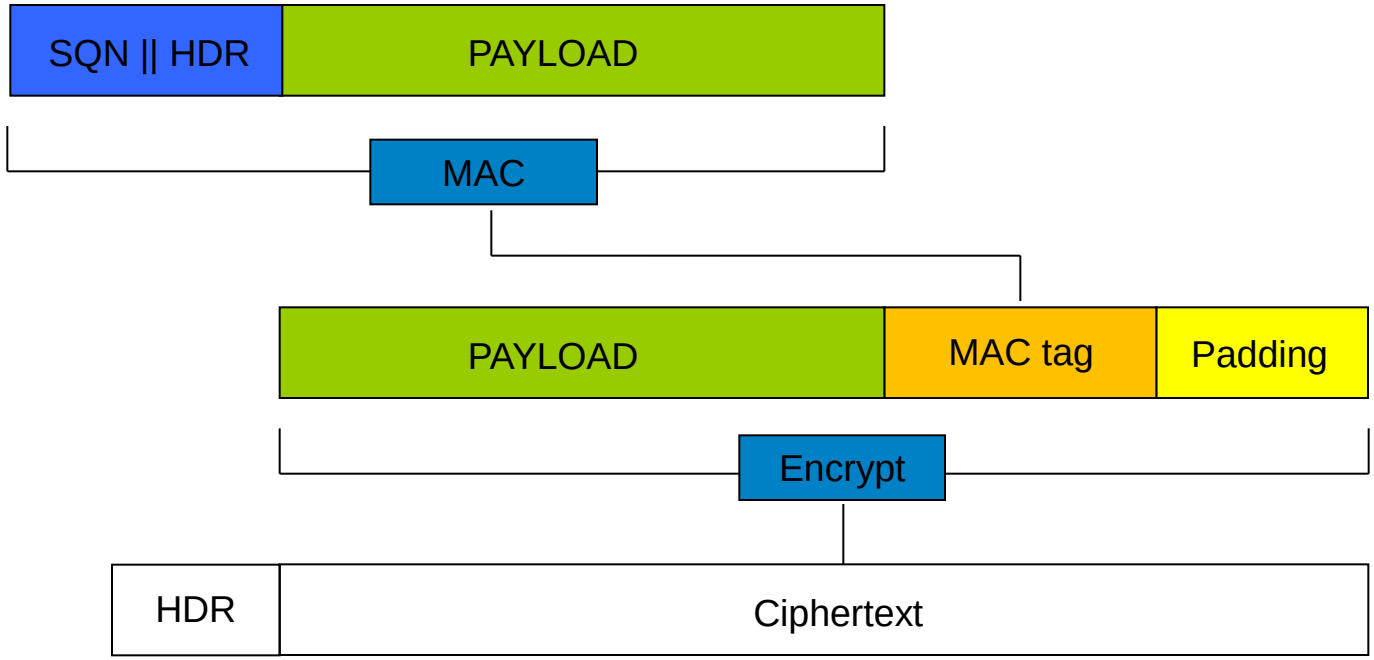

TLS v1.2

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.

Reminder: 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”

Theory for TLS Record Protocol?

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

- This is known to be **not** generically secure, according to the results of [BN00].
  - But it is INT-PTXT and IND-CPA secure

Theory for TLS Record Protocol?

- Building on results of [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.
- So are we done?

Theory for TLS Record Protocol?

- Analysis of [K01] assumes random IVs for CBC mode.
  - SSL v3.0 and TLS 1.0 use *chained* IVs.
  - TLS 1.1 and 1.2 recommend use of random IV.
- 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.
  - And padding is not integrity protected.
- RC4 has known statistical weaknesses.
- We'll show that these gaps between theory and reality do matter.

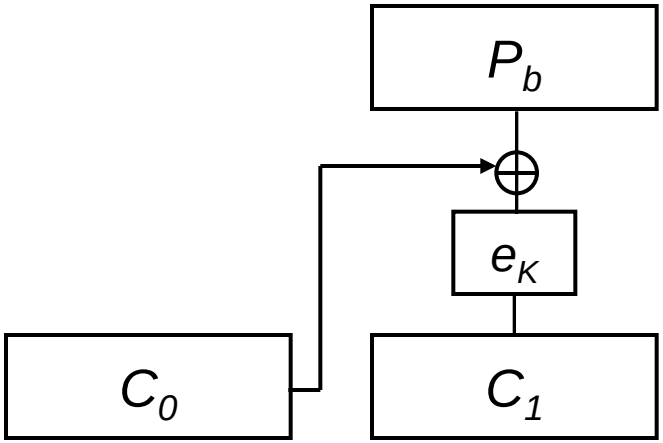
Attacking Chained IVs

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.
- Application to TLS noted 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.
- 16-year demonstration that attacks do get better.

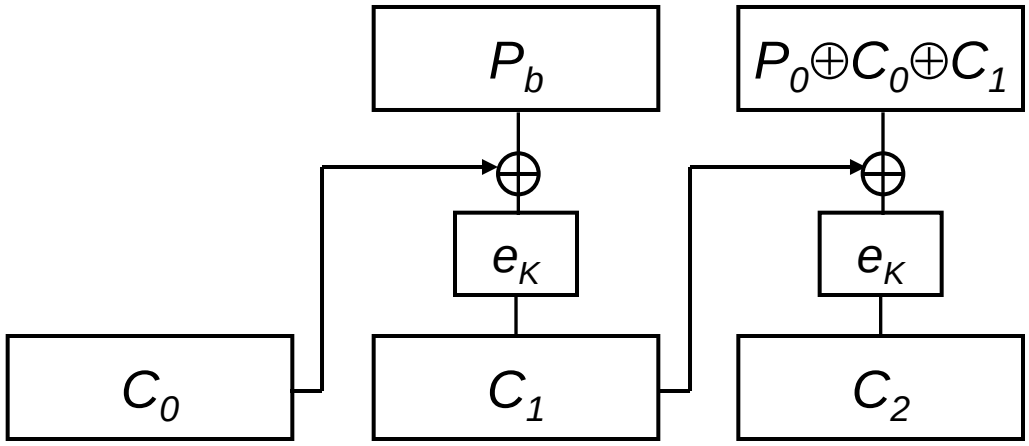
Attacking Chained IVs

- Suppose attacker wishes to distinguish encryptions of single blocks P_0 , P_1.
- Attacker makes LoR query for messages P_0, P_1.
- Attacker receives ciphertext $C = C_1$ for message P_b *w*here some known, previous block C_0 was used as the IV.

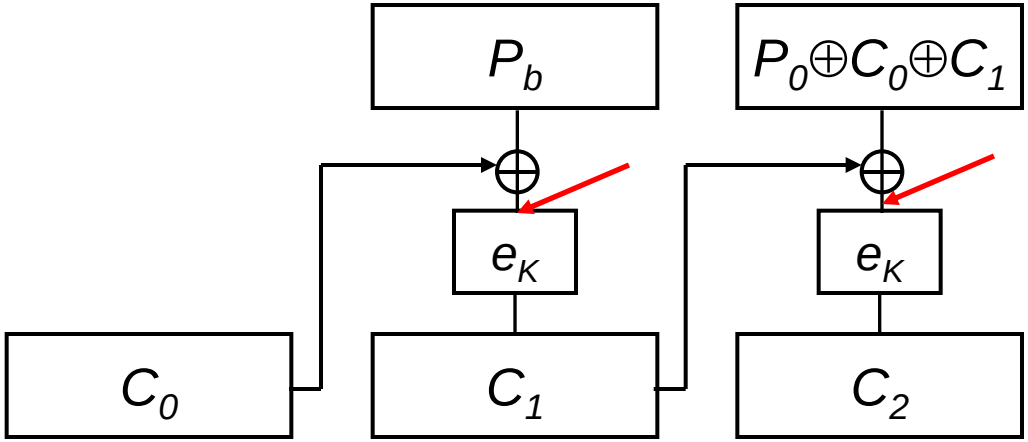


Attacking Chained IVs

- C_1 will be used as the IV for the *next* encrpytion.
- Attacker now makes LoR query on block $P_0 \oplus C_0 \oplus C_1$.
- Attacker receives single block ciphertext C_2.



Attacking Chained IVs

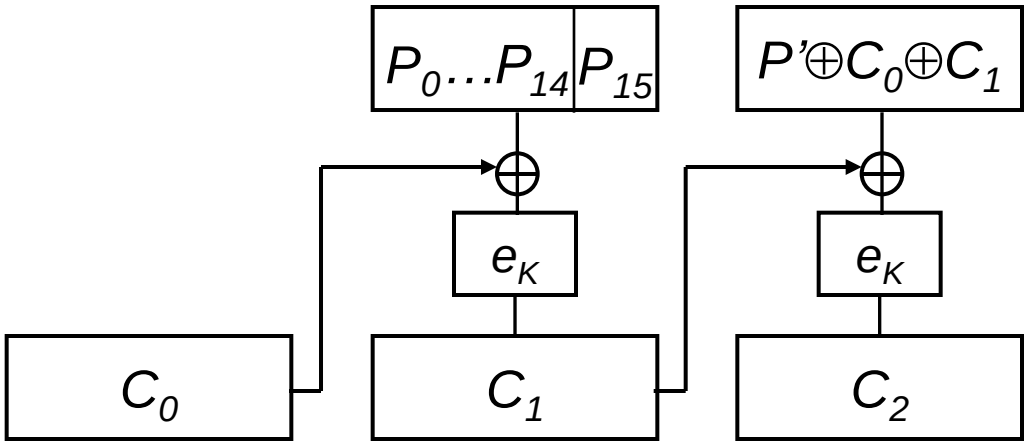


- If $P_b = P_0$, then inputs to block cipher are the same in both encryptions.
- Hence, if $P_b = P_0$, then $C_1 = C_2$.
- Otherwise, if $P_b = P_1$, then $C_1 \neq C_2$.
- So looking at C_1 and C_2 gives us a test to distinguish encryptions of P_0 and P_1.

Attacking Chained IVs

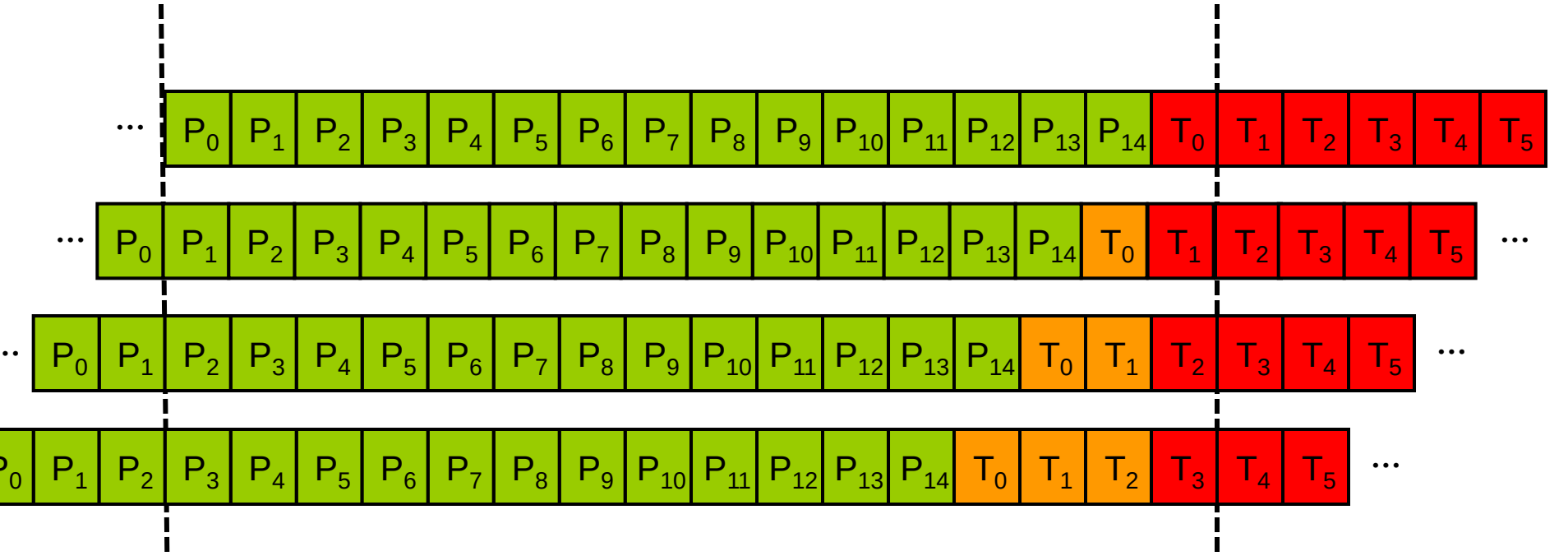
- Attack extends easily to multi-block messages.
- So IV chaining for CBC mode is broken in theory.
- How can we turn this into a practical attack on TLS?
- We want plaintext recovery rather than a distinguishing attack.
- We need to realise the chosen plaintext requirement.

The BEAST – Part 1



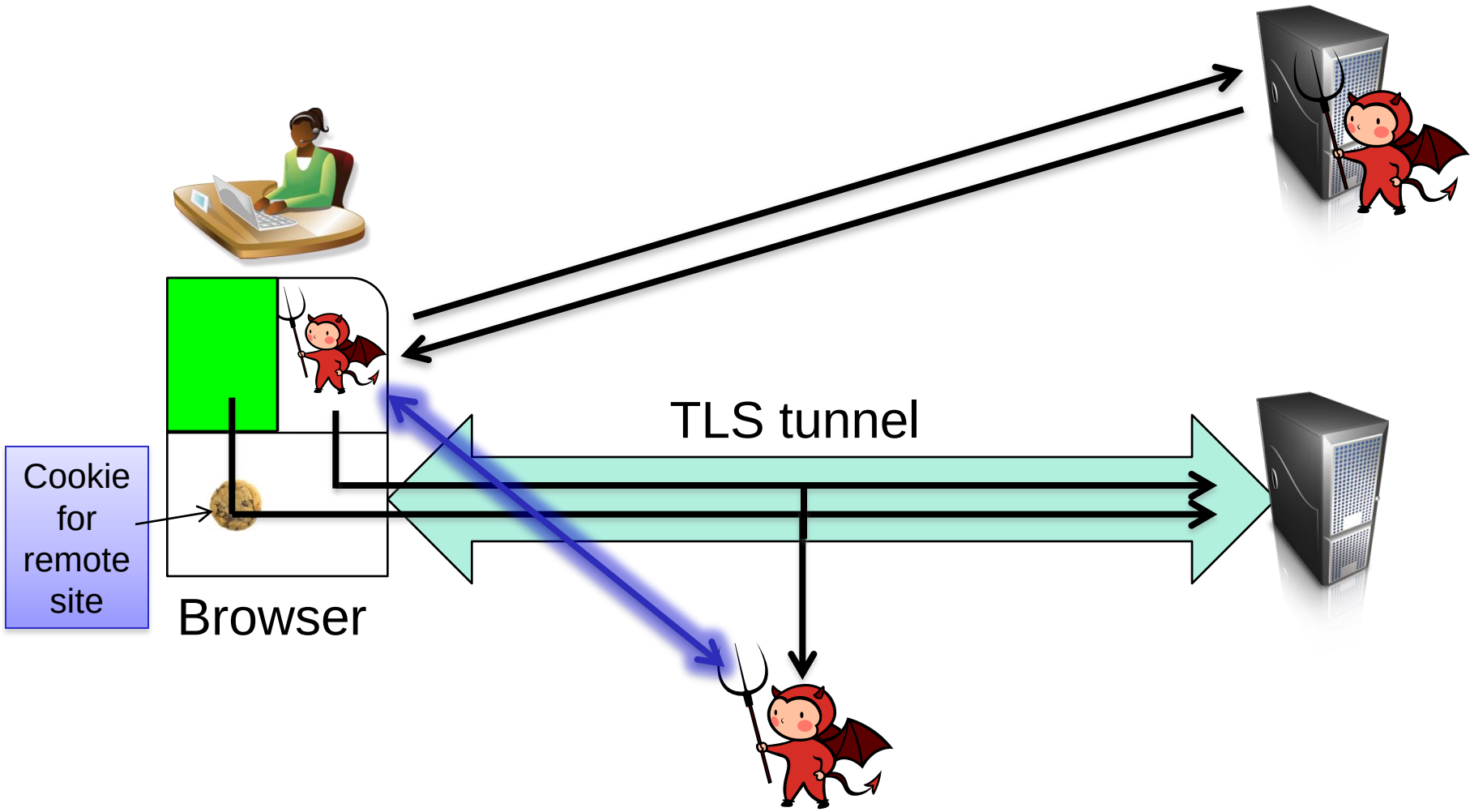
- Assume bytes P_0, P_1,… P_{14} are known, try to recover P_{15}.
- Use $P_0P_1 \ldots P_{14}$ as first 14 bytes of P'.
- Iterate over 256 possible values in position 15 in P'.
- $P'_{15} = P_{15}$ if and only if $C_1 = C_2$.
- So average of 128 trials to extract P_{15} when remaining bytes in block are known.

The BEAST – Part 2



- Now assume attacker can control position of unknown 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 – Part 3



The BEAST – Key Features

- BEAST JavaScript loaded ahead of time into client browser from compromised or malicious wesbite.
- Provides chosen-plaintext capability.
- Attack target is HTTP secure cookie.
- JavaScript uses HTTP padding to control positions of unknown bytes (chosen boundary privilege).
- Difficult to get fine control over byte/block positions.
  - Need to be able to inject chosen plaintext block at the very start of Record Protocol messages.
- JavaScript also needs to communicate with MITM attacker.

Summary: it's complicated, but it can be made to work.

- Techniques useful in later TLS attacks too.

The BEAST – Impact

- The BEAST was a major headache for TLS vendors.
  - Perceived to be a realistic attack.
  - Most client implementations were “stuck” at TLS 1.0.
- Best solution: switch to using TLS 1.1 or 1.2.
  - Uses random IVs, so attack prevented.
  - But needs server-side support too.
- For TLS 1.0, various hacks were done:
  - Use 1/n-1 record splitting in client.
    - Now implemented in most but not all (?) browsers.
  - Send 0-length dummy record ahead of each real record.
    - Breaks some implementations.
  - Or switch to using RC4?
    - As recommended by many expert commentators.

Lessons

- A theoretical vulnerability pointed out in 1995 became a practical attack in 2011.
  - Attacks really do get better (worse!) with time.
  - Practitioners really should listen to (some) theoreticians.
  - And, in this case, they did: TLS 1.1 and 1.2 use random IVs.
  - Problem was that no-one was using these versions.
- Ideas from the wider security field were needed to make the attacks headline news.
  - Man-in-the-browser via Javascript.
  - Importance of demo/youtube video and showing people the plaintext.

Lecture 3
Kenny Paterson

Bar-Ilan Winter School on Symmetric Cryptography

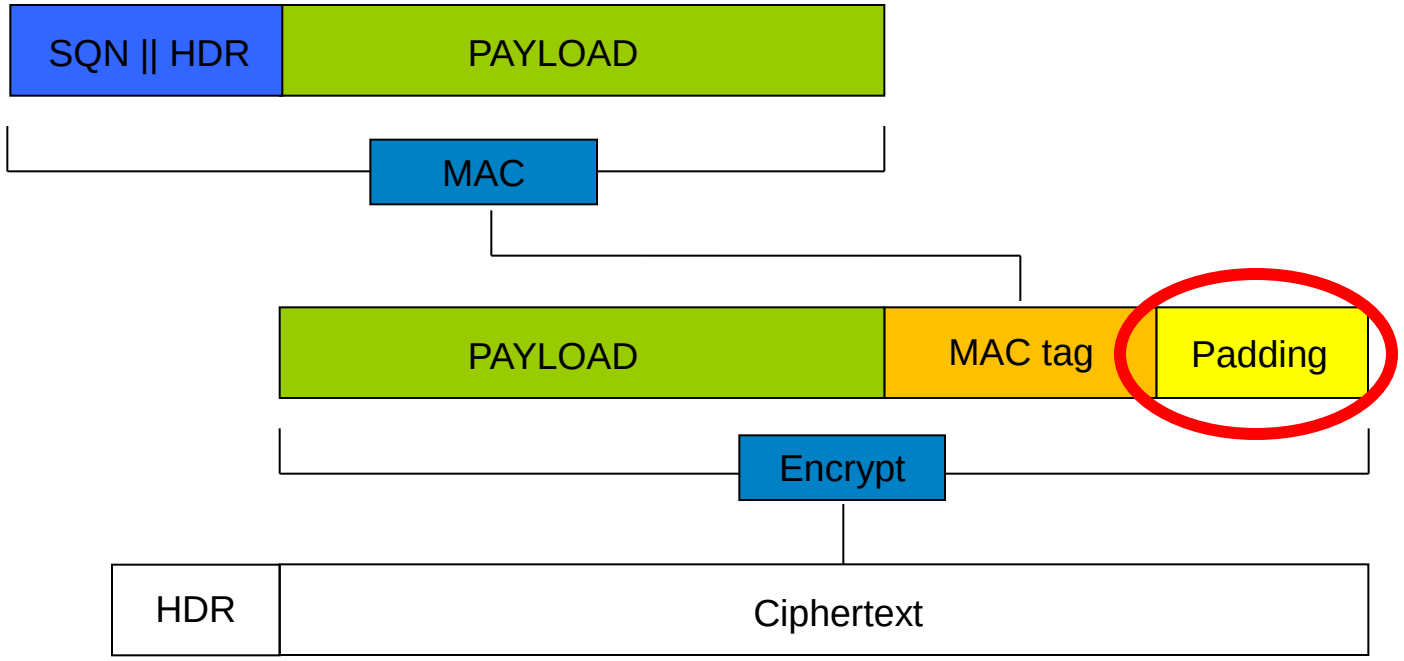
Royal Holloway
University of London

Information Security Group

Outline

- Padding oracle attacks on TLS
- Lucky 13
- TLS security proof

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, so longest possible padding pattern is:

  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.
- This “goal” has interesting theoretical implications.

  -Recall that, in IND-CPA/IND-CCA models, m_0 and m_1 always have the same length.

Handling Padding During Decryption

- TLS 1.0 error alert:

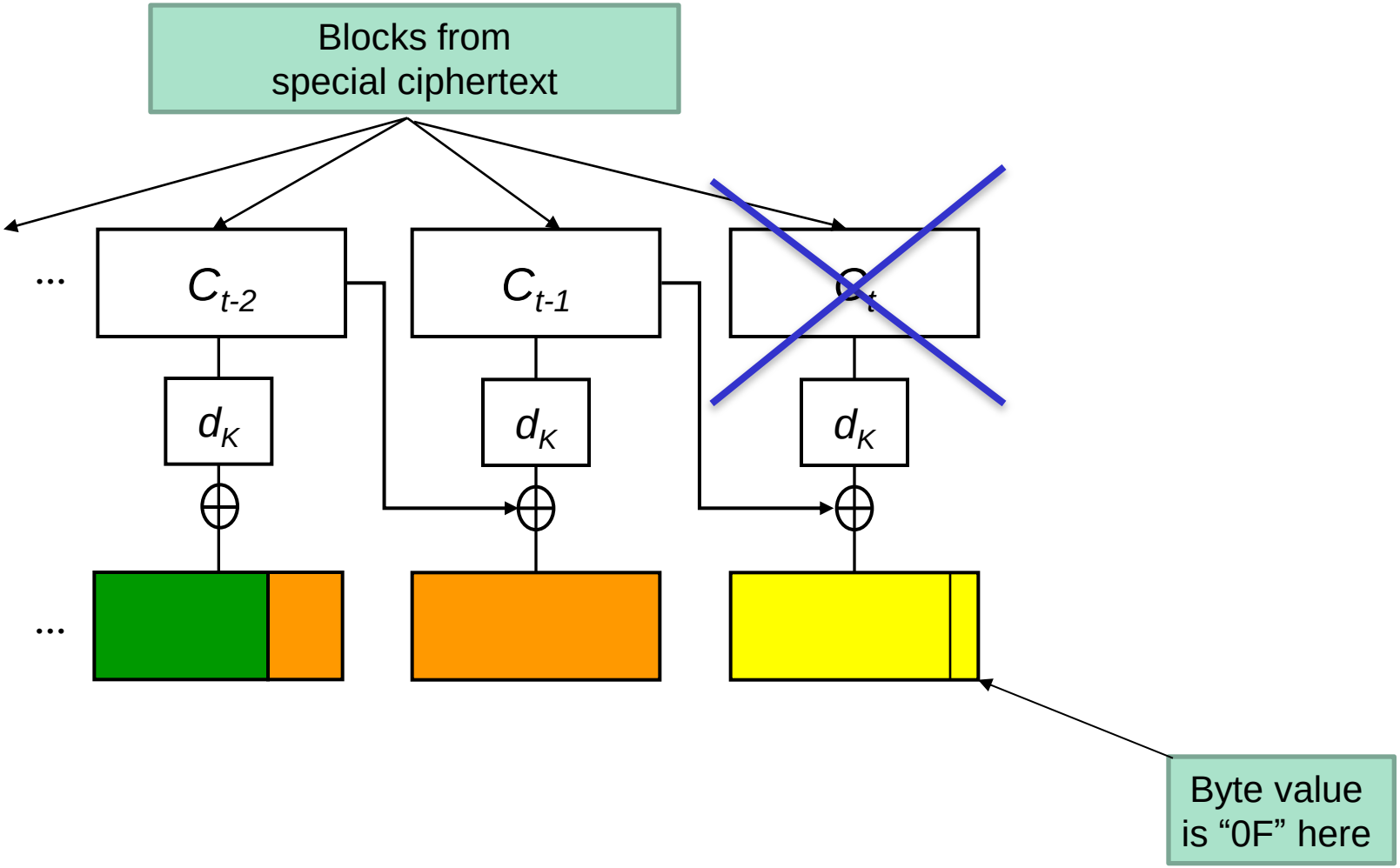
  decryption_failed: A TLSCiphertext decrypted in an invalid way: either it wasn`t an even multiple of the block length or its padding values, when checked, weren't correct. This message is always fatal.

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

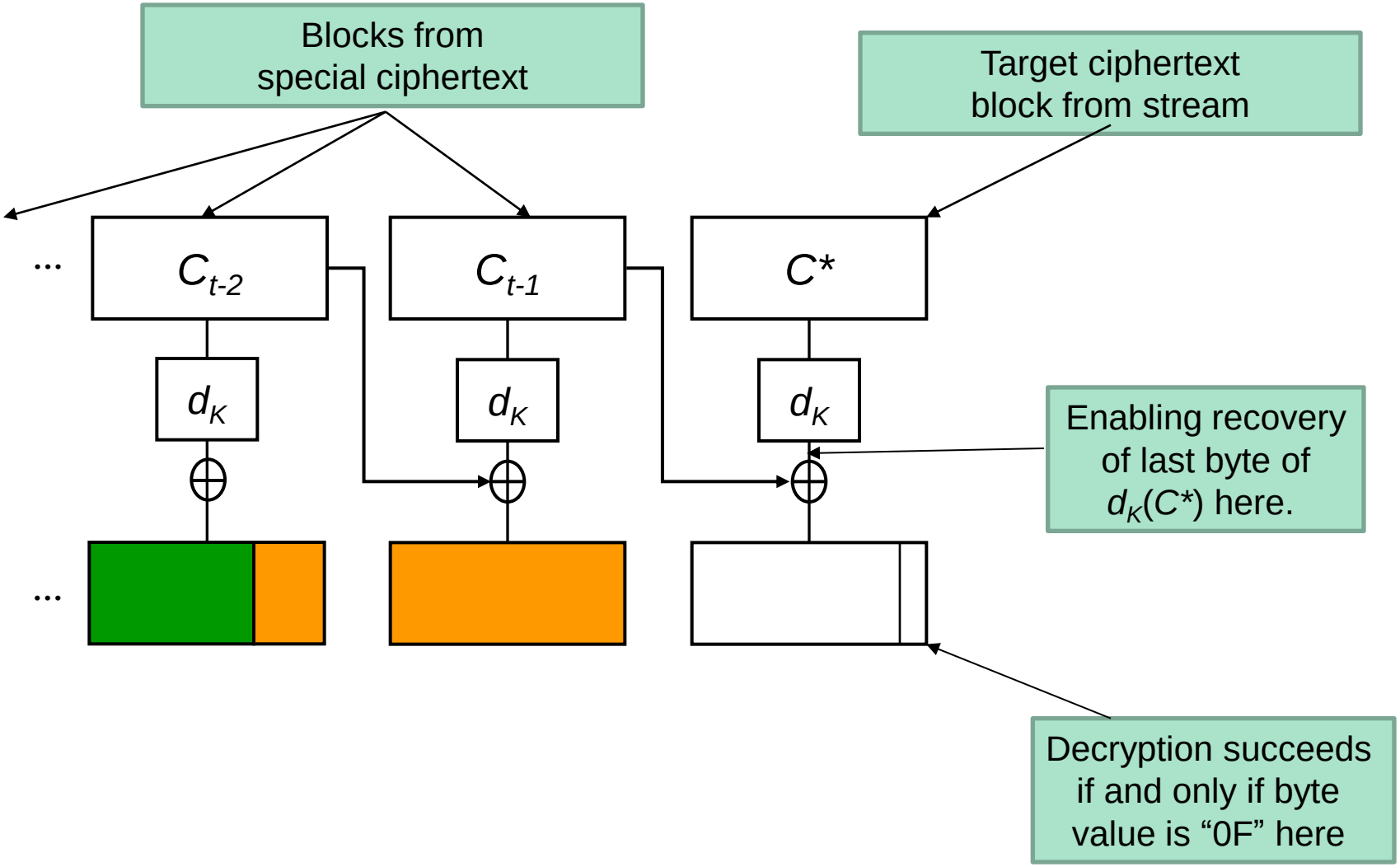
Insecurity of Weak Padding Checks

- Recall 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:

  $(C_{t-1})_{15} \oplus (d_K(C^*))_{15}$ = "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

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

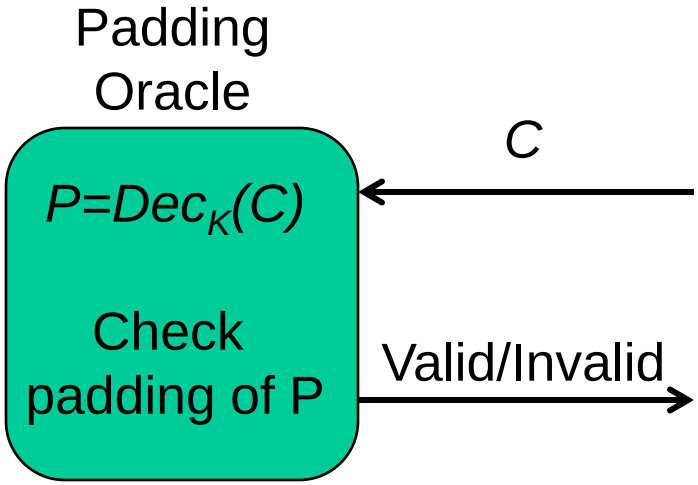
00;
01, 01;
02, 02, 02;

FF, FF,……..FF

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

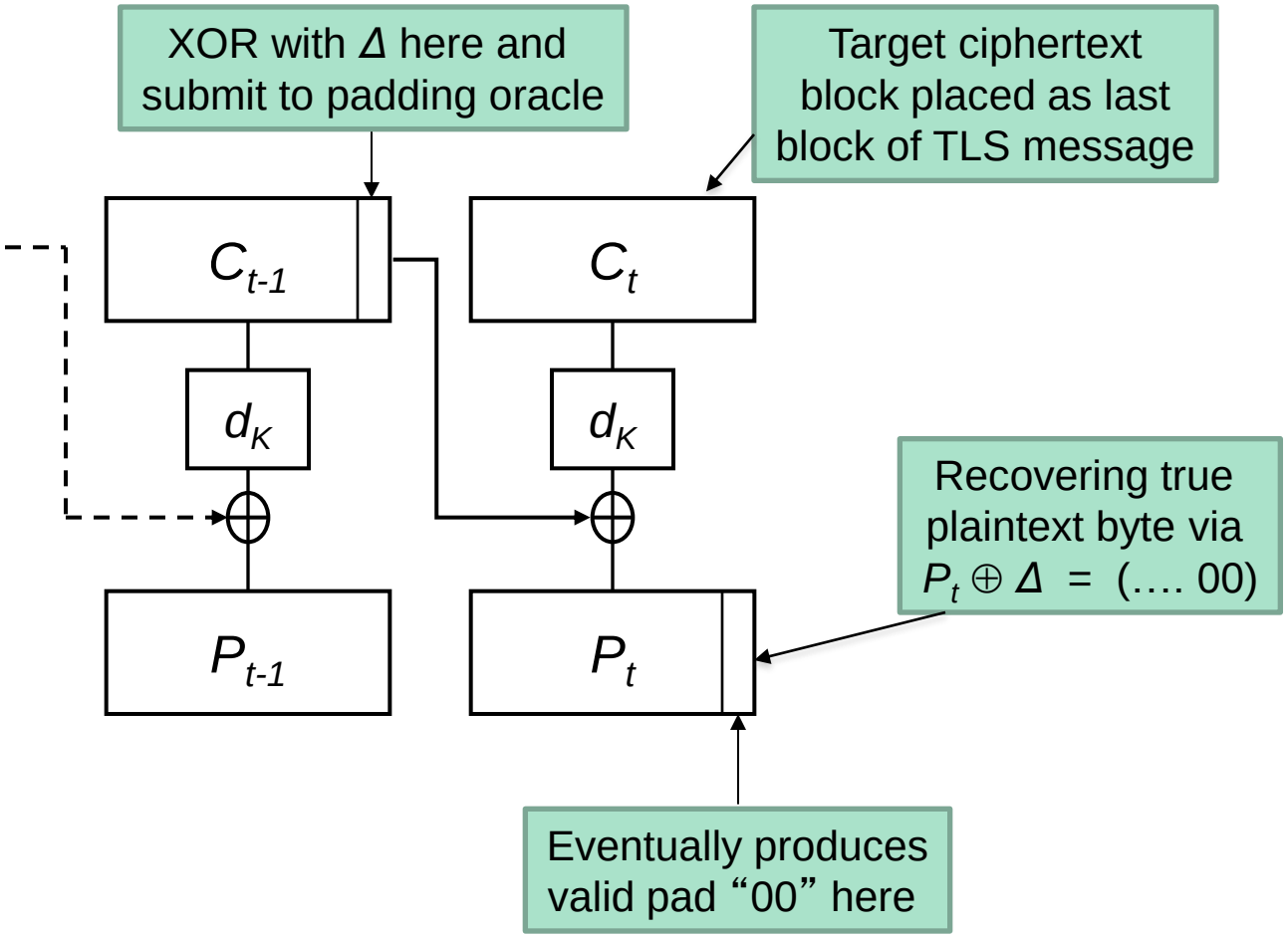
Padding Oracles

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

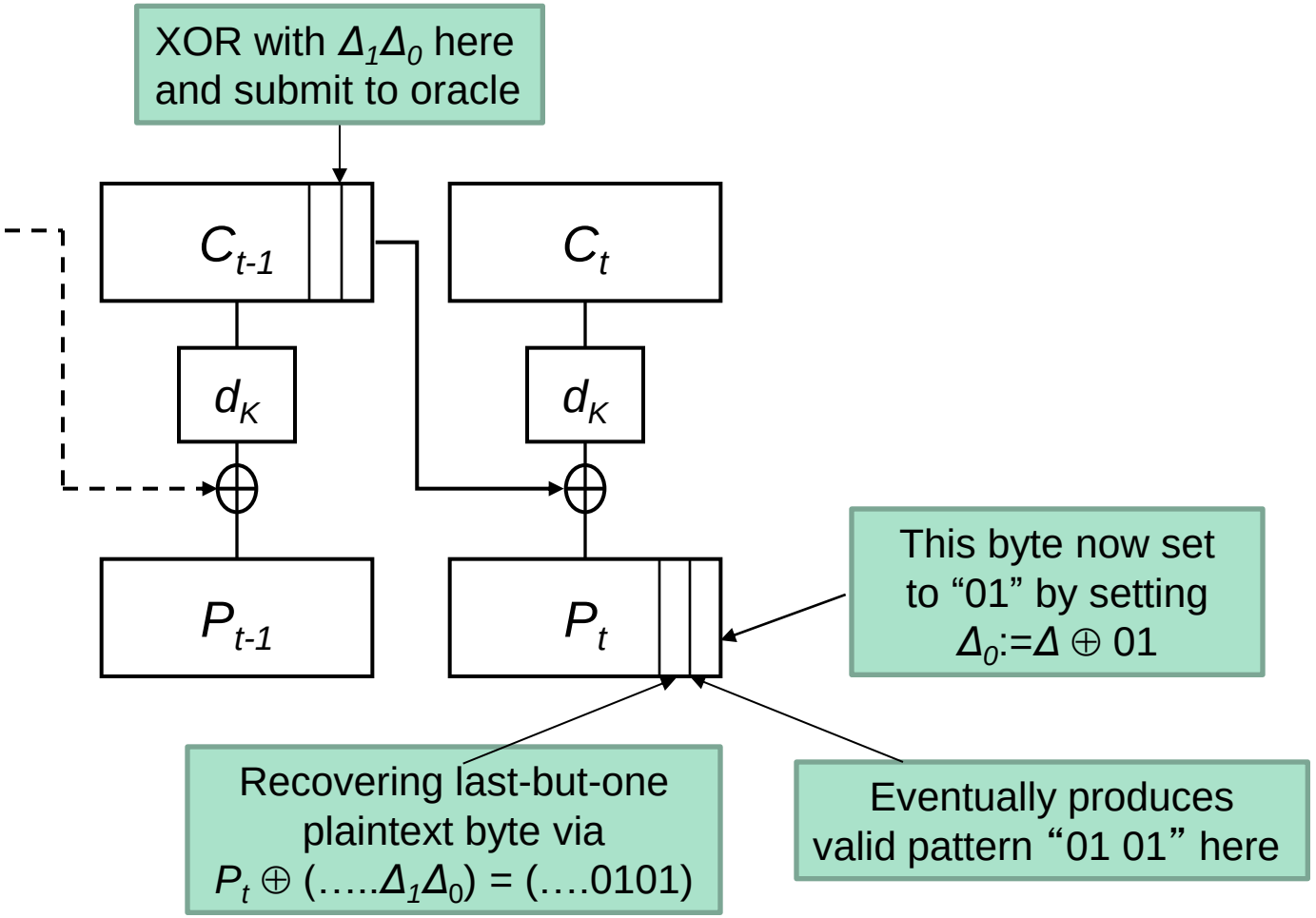


- Vaudenay showed that, for CBC mode and for certain padding schemes, a padding oracle can be used to build a decryption oracle!
- We'll focus on TLS, but padding oracle attacks have been widely deployed, e.g. DTLS, ASP.NET, XML encryption.

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.
- Attack extends 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] :

- With the natural implementation, a MAC failure error message will appear on the network **later** than a padding failure error message.
- Why?
- Recall the sequence of processing steps:
  - Decrypt
  - Check pad (abort if wrong)
  - Check MAC (abort if wrong)
- Hence MAC check only done if padding is good.
- And if padding is bad, processing terminates quickly (MAC check is relatively slow).

TLS Padding Oracles In Practice?

Canvel *et al.* [CHVV03] :

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

- **But** the modified ciphertexts always fail the MAC check (or the padding check).
- And the errors are fatal.
- So the attacker only gets query to padding oracle before try before connection is lost.
- Attacker can learn one byte of plaintext, with probability only 1/256.
  - Chances of being correct on first query.

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 approach: use BEAST-style malware.

- OpenSSL had a detectable timing difference.
  - Difference is time taken to compute HMAC on message.
  - Roughly 2ms difference for 2^{14} byte messages.
  - Enabling recovery of TLS-protected Outlook passwords in about 3 hours.

DTLS and Padding Oracles

- Recall that DTLS is basically TLS over UDP.
- UDP is not reliable like TCP, so DTLS has to tolerate packet drops, replays, etc.
- This means that the connection is *not* terminated in the event of an error.
- But there are no error messages to time.

Breaking DTLS in OpenSSL

- [AP12]: Can we apply padding oracle ideas to DTLS?

- But surely DTLS implementations would have learned lessons from old TLS attacks?
  - DTLS 1.0 is based on the TLS 1.1 specification.
  - So we should not expect a timing-based side channel to exist…

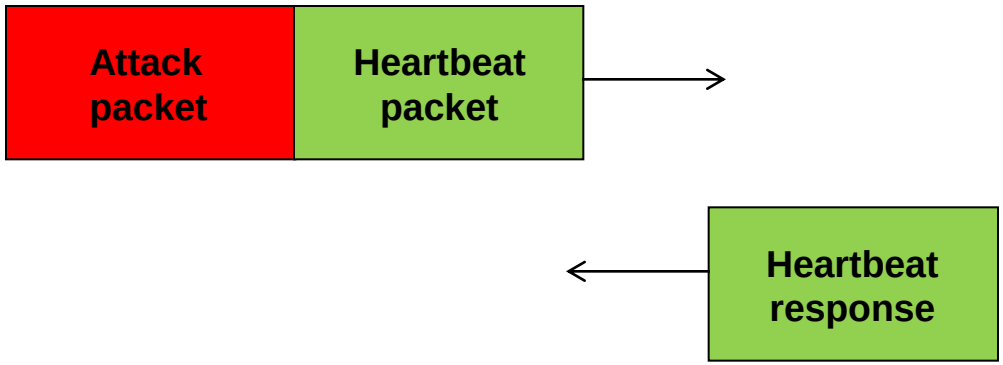
Breaking DTLS in OpenSSL

- OpenSSL implementations of DTLS prior to versions 0.9.8s/1.0.0f did *not* check the MAC if the padding check fails.

- Hence the timing difference observed in [CHVV03] should still be present!

Breaking DTLS in OpenSSL

- Bad news: no error messages to time.
  - Not a major hurdle:

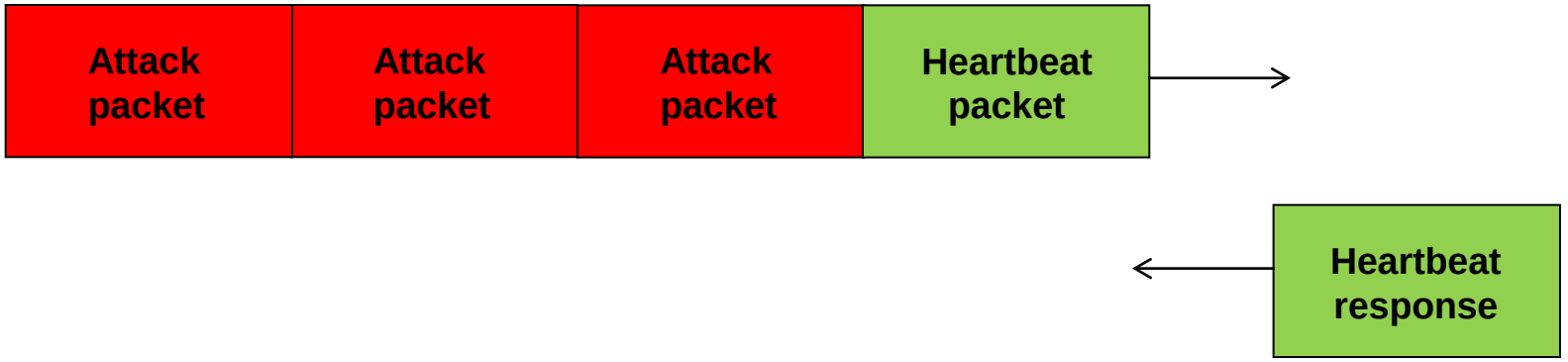


  - Attack packet takes longer to process if padding is good.
  - So measure time difference between sending attack packet + heartbeat and receiving heartbeat response.
  - This serves as a proxy for timing error messages

Breaking DTLS in OpenSSL

Good news: errors in DTLS are not fatal.

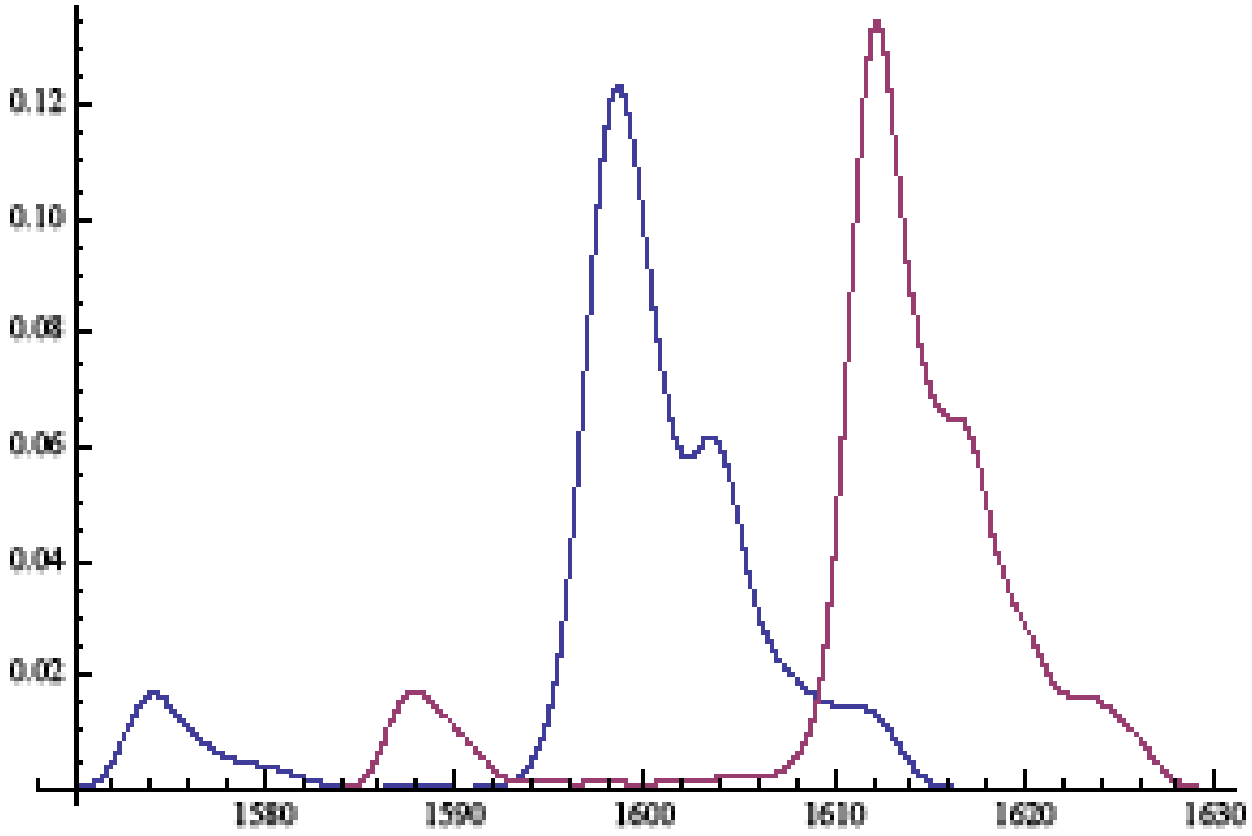
- Actually *very* good news: allows amplification of timing difference using *packet trains*.



- With care, the timing difference arising from the attack packets can be made cumulative!
- Repeat over many trains and use statistical techniques to detect timing difference.

Experimental Results

- HMAC-SHA1 + CBC-AES, 10 packets per train, 1456 bytes per packet:



Experimental Results

- Example for HMAC-SHA1 + CBC-AES
  - 192 byte packets
  - 2 packets per train
  - 10 trains per byte value
- Statistical processing:
  - Get timings for each set of 10 trains; remove outliers
  - Keep *minimum* time for each byte value tried.
  - Select as correct byte the one that *maximizes* the resulting time.
- Success probabilities:
  - Per byte: 0.996
  - Per block: 0.94

Observation

- DTLS turns out to be substantially easier to attack than TLS.
  - Because of ability to amplify timing differences using packet trains.
  - This is a consequence of the choice of transport protocol: UDP instead of TCP.
  - Details in [AP12].
- This distinction does not arise in current formal security models for encryption.
  - But could easily be modelled.

Countermeasures to Padding Oracle Attacks

- Redesign TLS:
  - Pad-MAC-Encrypt or Pad-Encrypt-MAC.
  - Too invasive, did not happen.
- Switch to using RC4?
  - Seems to have been a widespread reaction.
- Or add a fix to CBC mode 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 1.1 (2006) and 1.2 (2008) specifications:

…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 was 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.

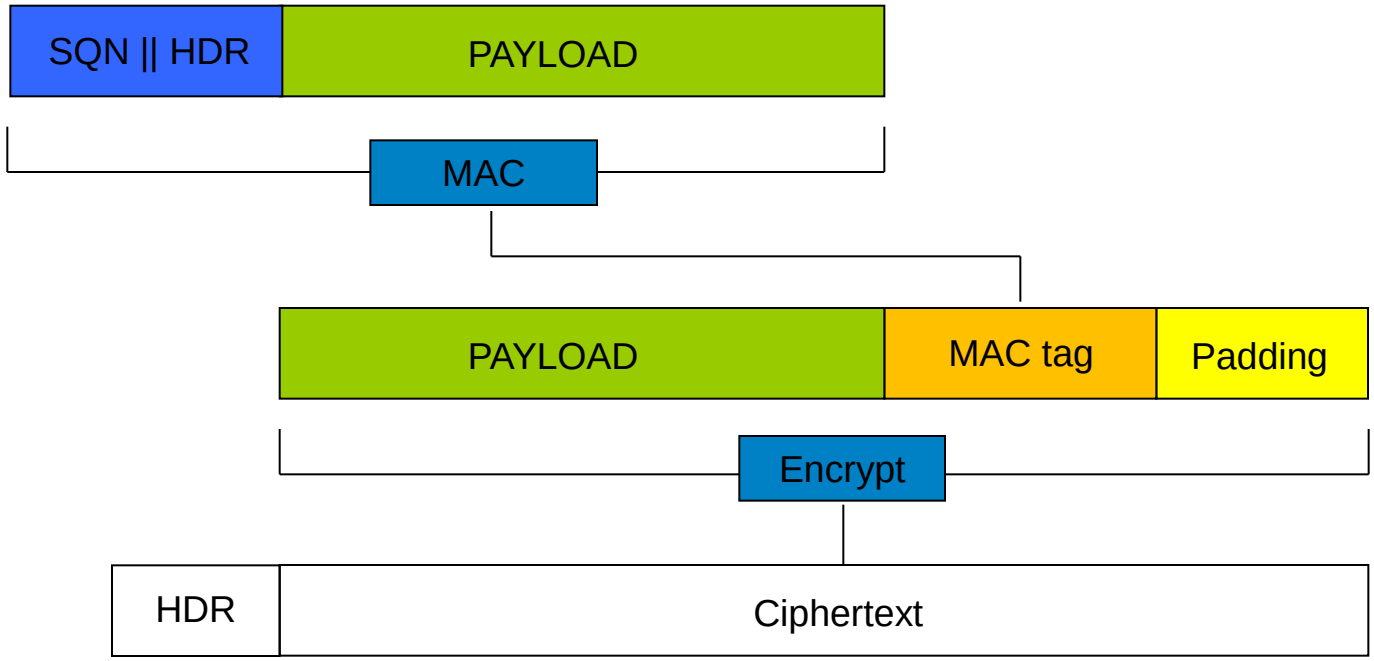
Lucky 13 [AP13]

- Distinguishing attacks and full plaintext recovery attacks against TLS-CBC implementations following the advice in the TLS 1.1/1.2 specs.
  - 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.

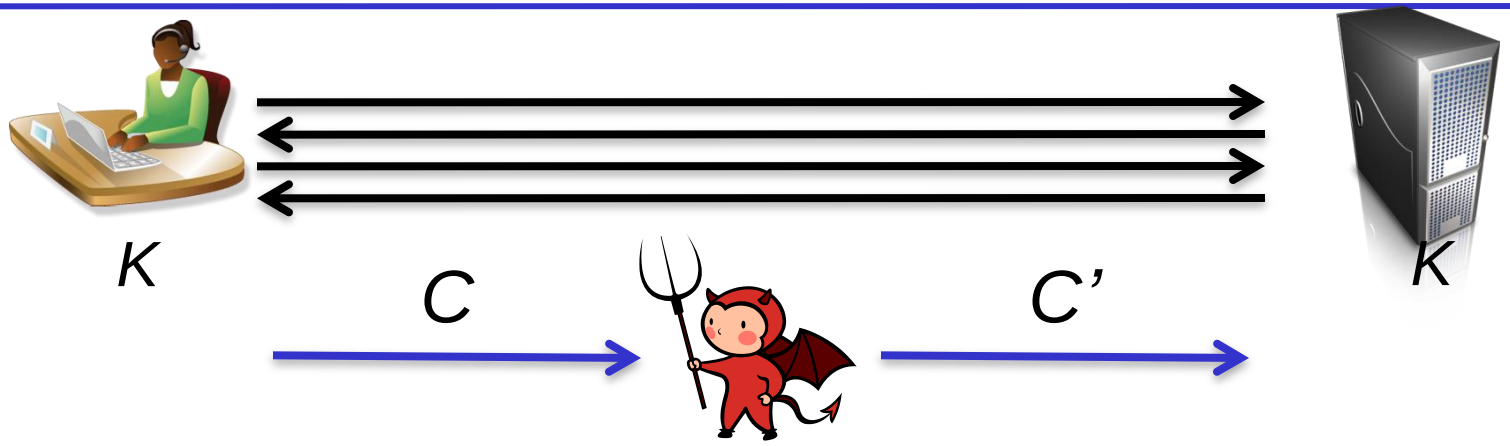
Reminder: MAC-Encode-Encrypt in TLS



Lucky 13 – Basic Idea

- TLS decryption removes padding and MAC tag to extract PAYLOAD.
- HMAC computed on SQN || HDR || PAYLOAD.
- HMAC computation involves adding ≥9 bytes of padding and 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;
  - ….

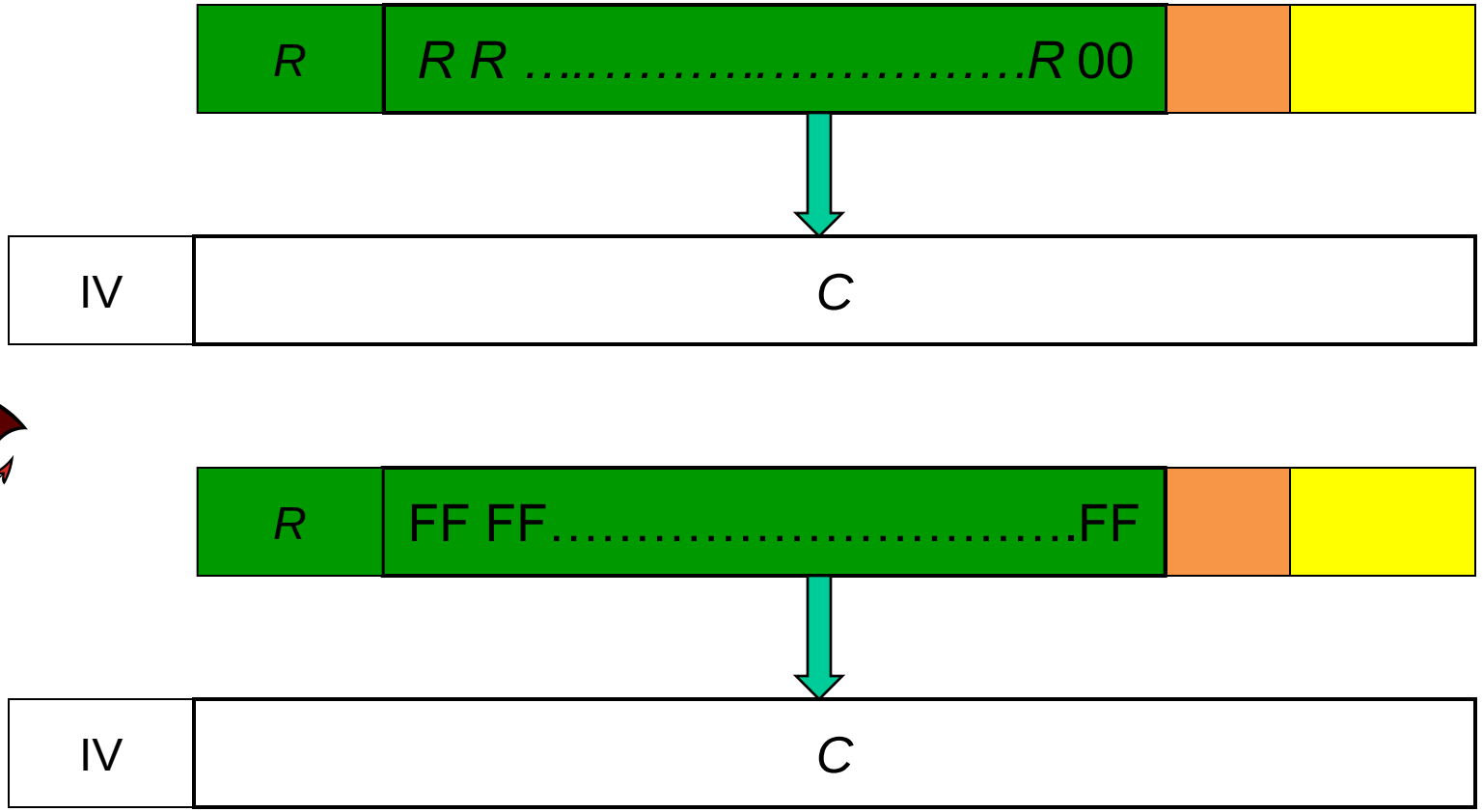
Lucky 13 Distinguishing Attack



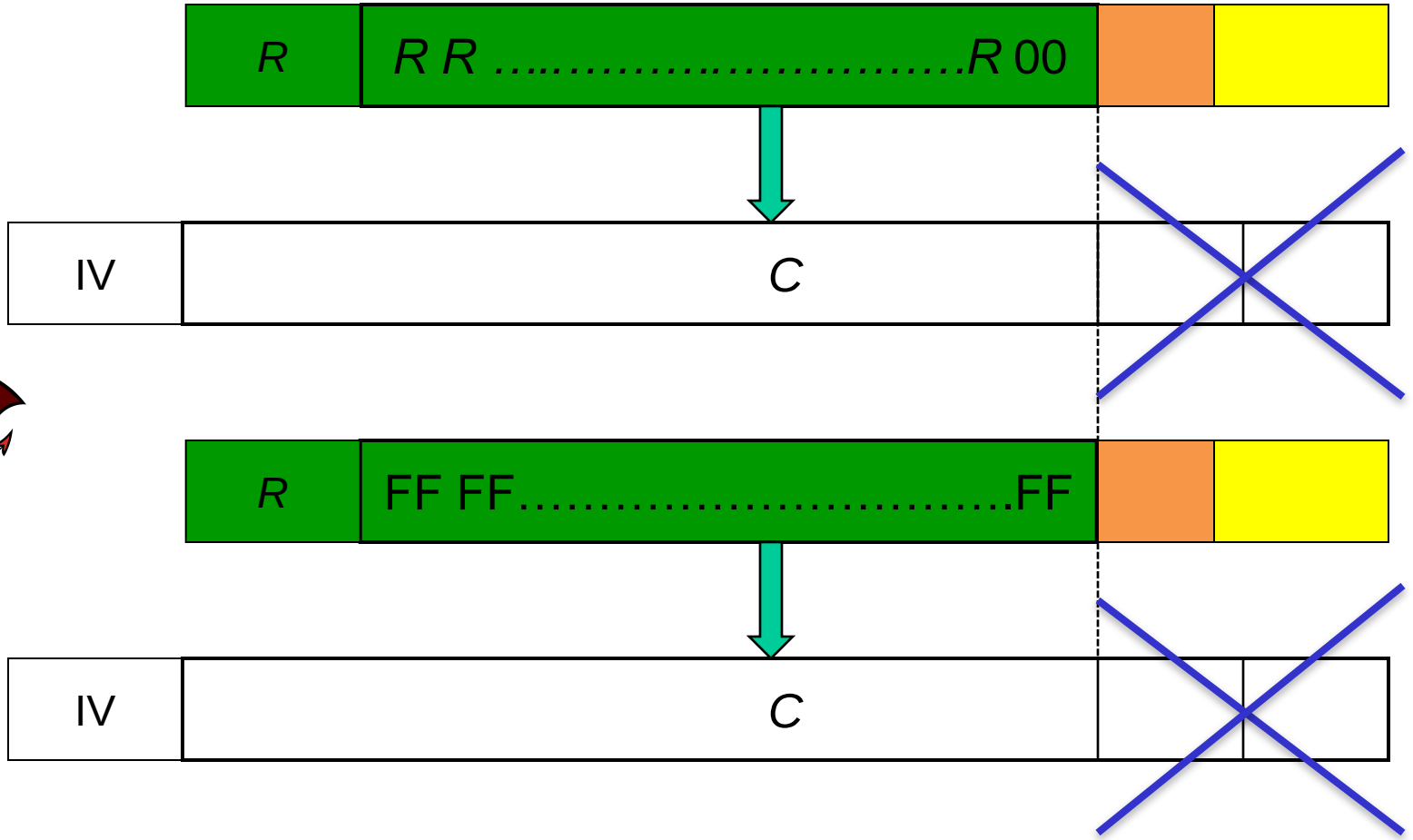
$C = \text{Enc}_K(M)$ M is either $R^{287} \| 00$ or $R^{32} \| FF^{256}$

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

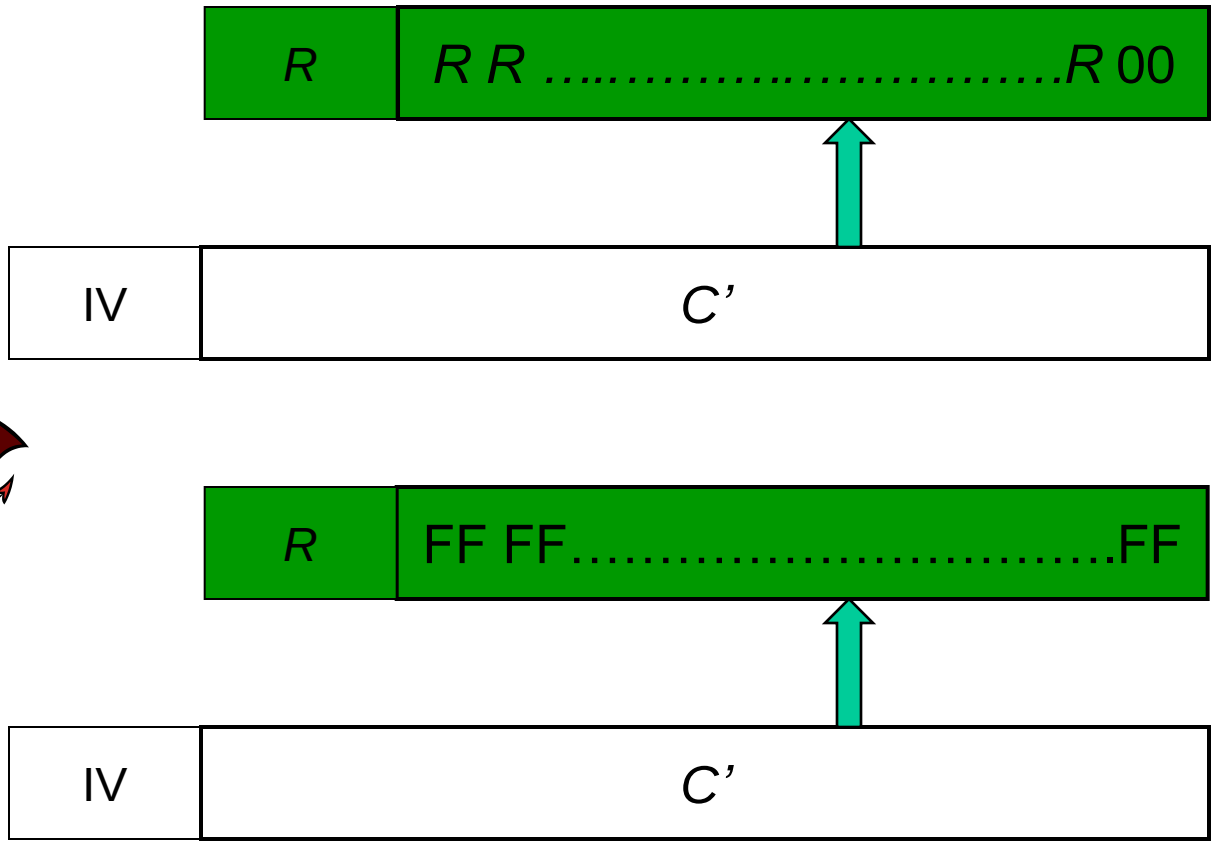
Lucky 13 Distinguishing Attack – Choose



Lucky 13 Distinguishing Attack – Maul



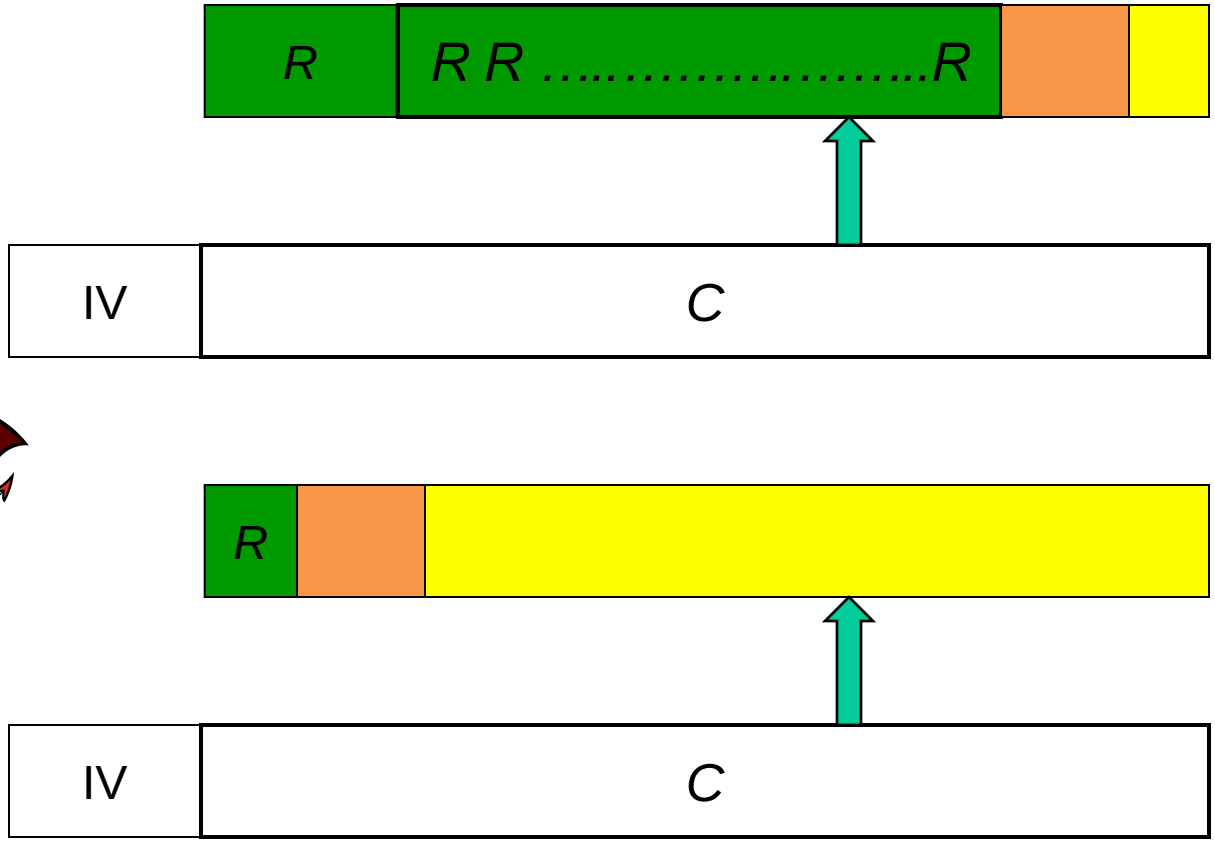
Lucky 13 Distinguishing Attack – Inject



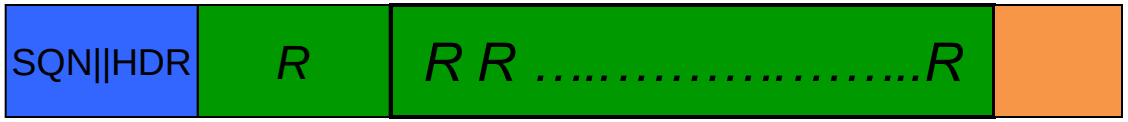
1-byte valid padding
20-byte MAC
267-byte message

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

Lucky 13 Distinguishing Attack – Decrypt

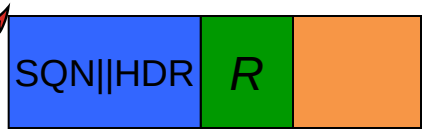


Lucky 13 Distinguishing Attack – Decrypt



280 bytes

Slow MAC verification

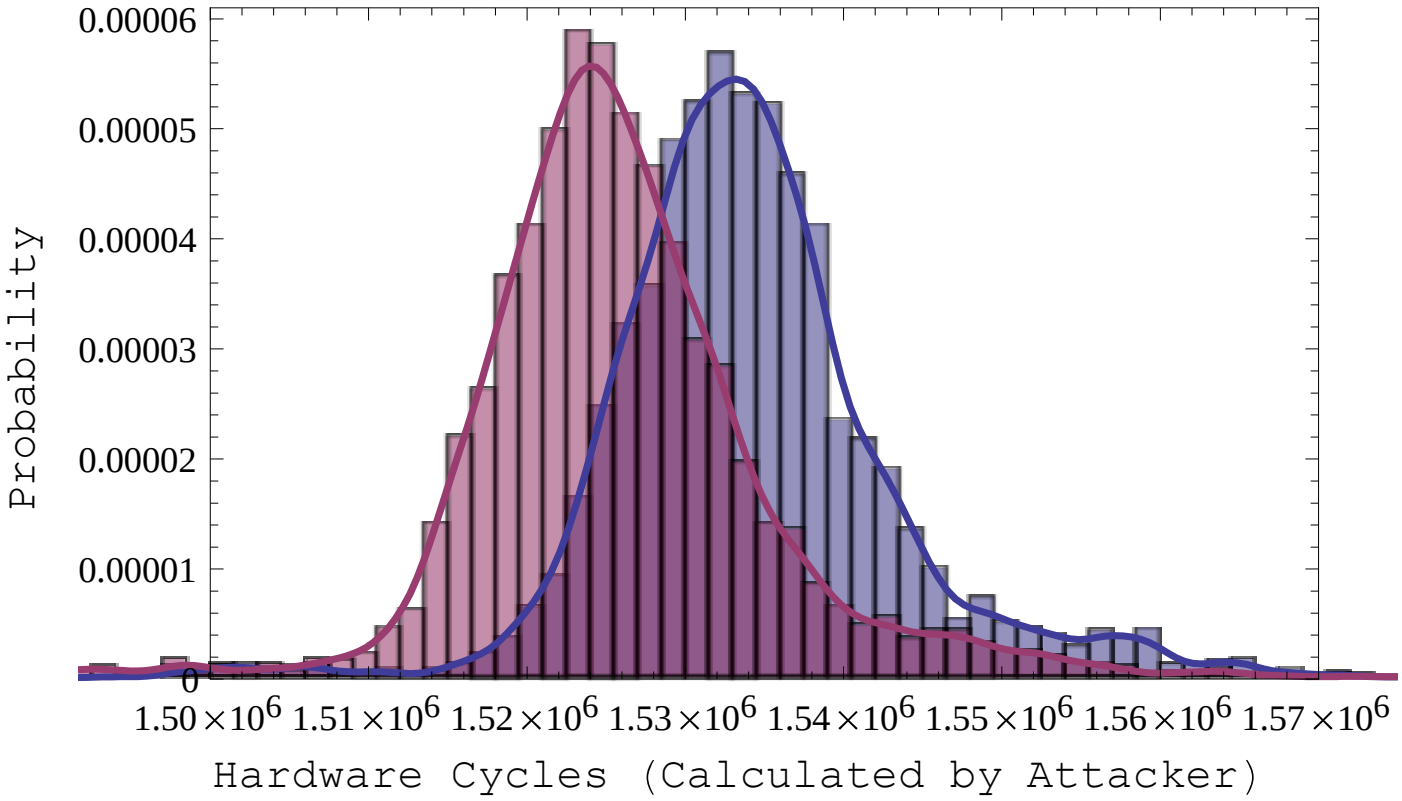


25 bytes

Fast MAC verification

Timing difference: 4 SHA-1 compression function evaluations

Experimental Results for Distinguishing Attack

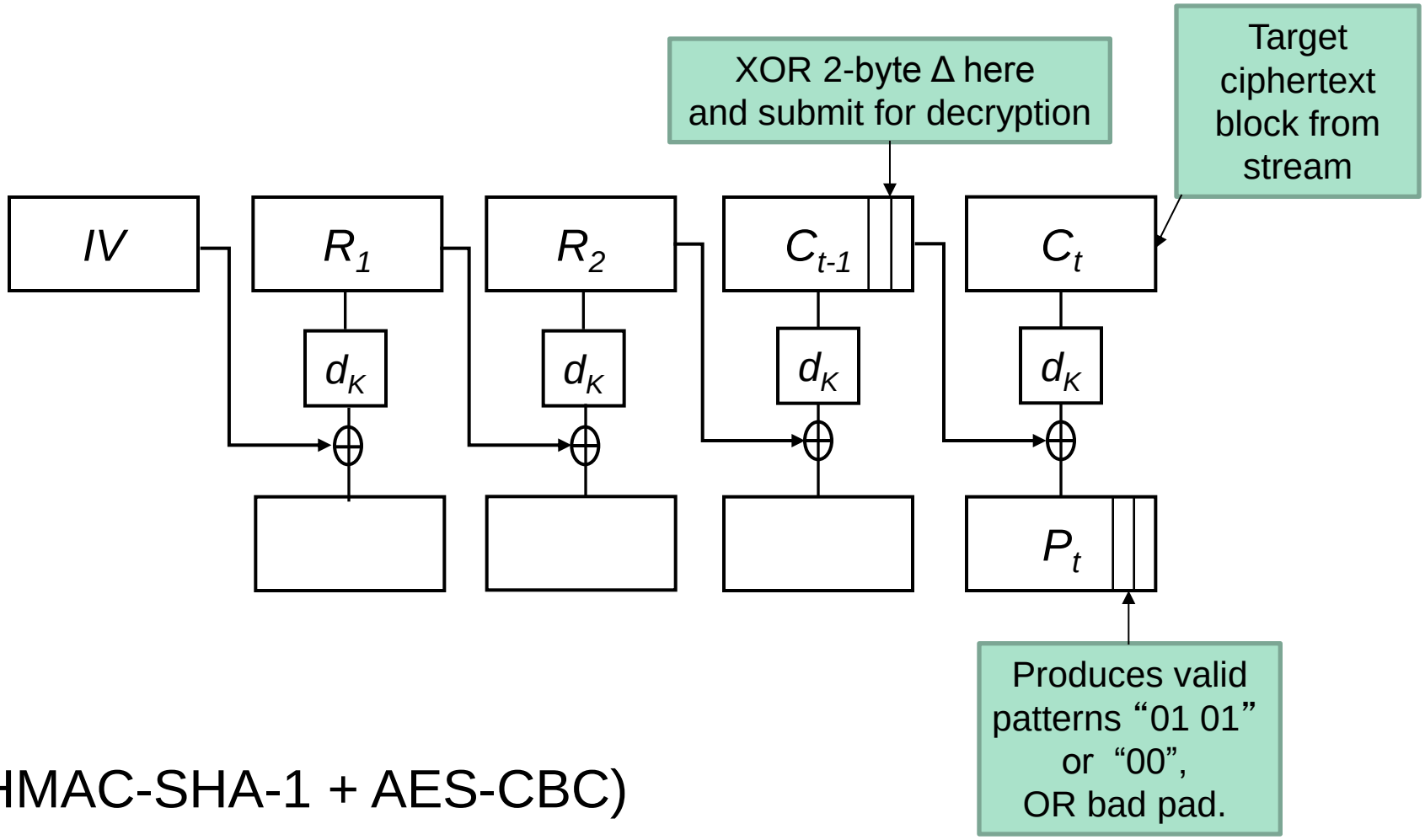


- OpenSSLv1.0.1 on server running at 1.87Ghz.
- 100 Mbit LAN.
- Difference in means is circa 3.2 μs.

Success Probability

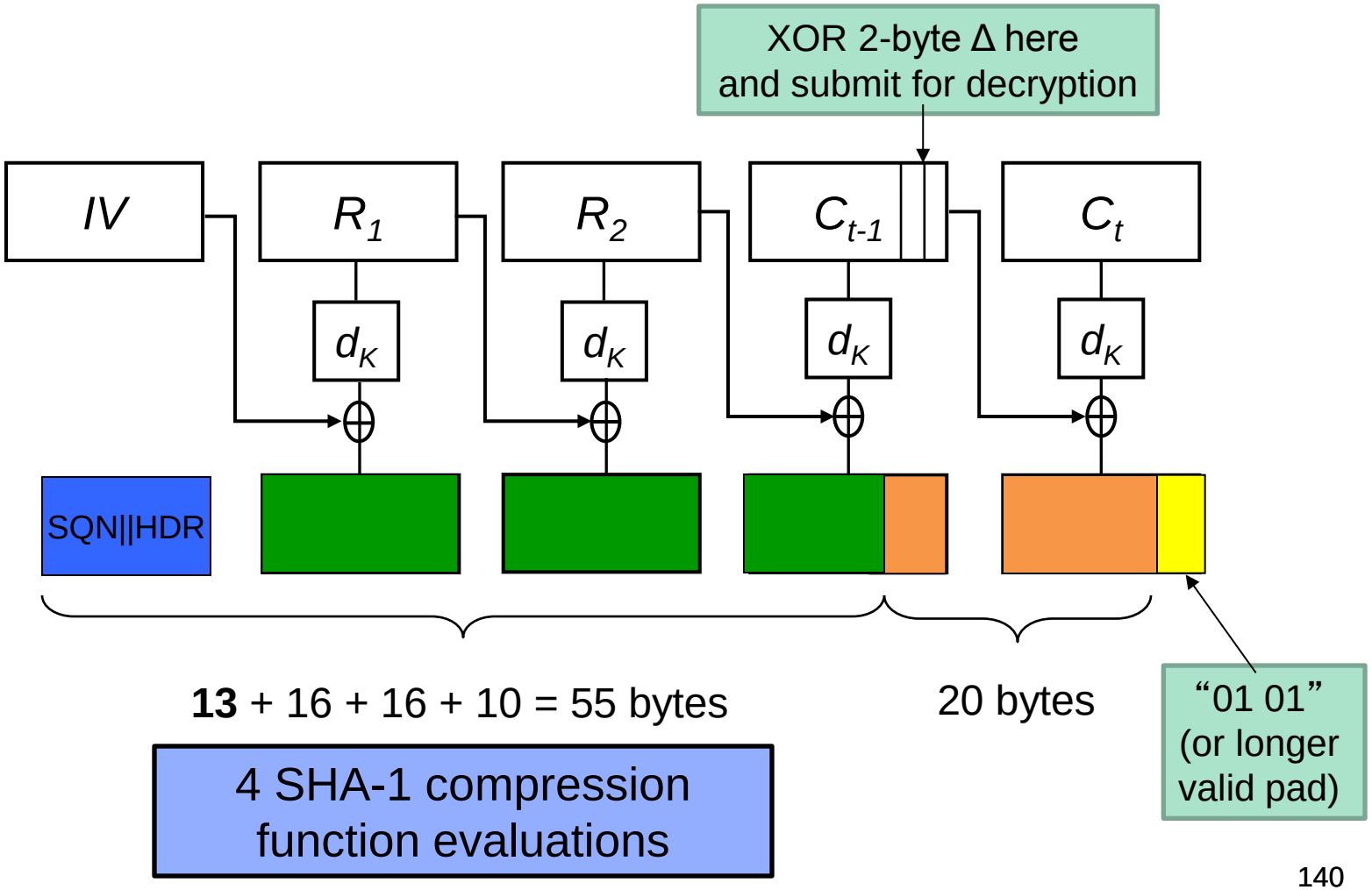
Number of Sessions	Success Probability
1	0.756
4	0.858
16	0.951
64	0.992
128	1

Lucky 13 – Plaintext Recovery

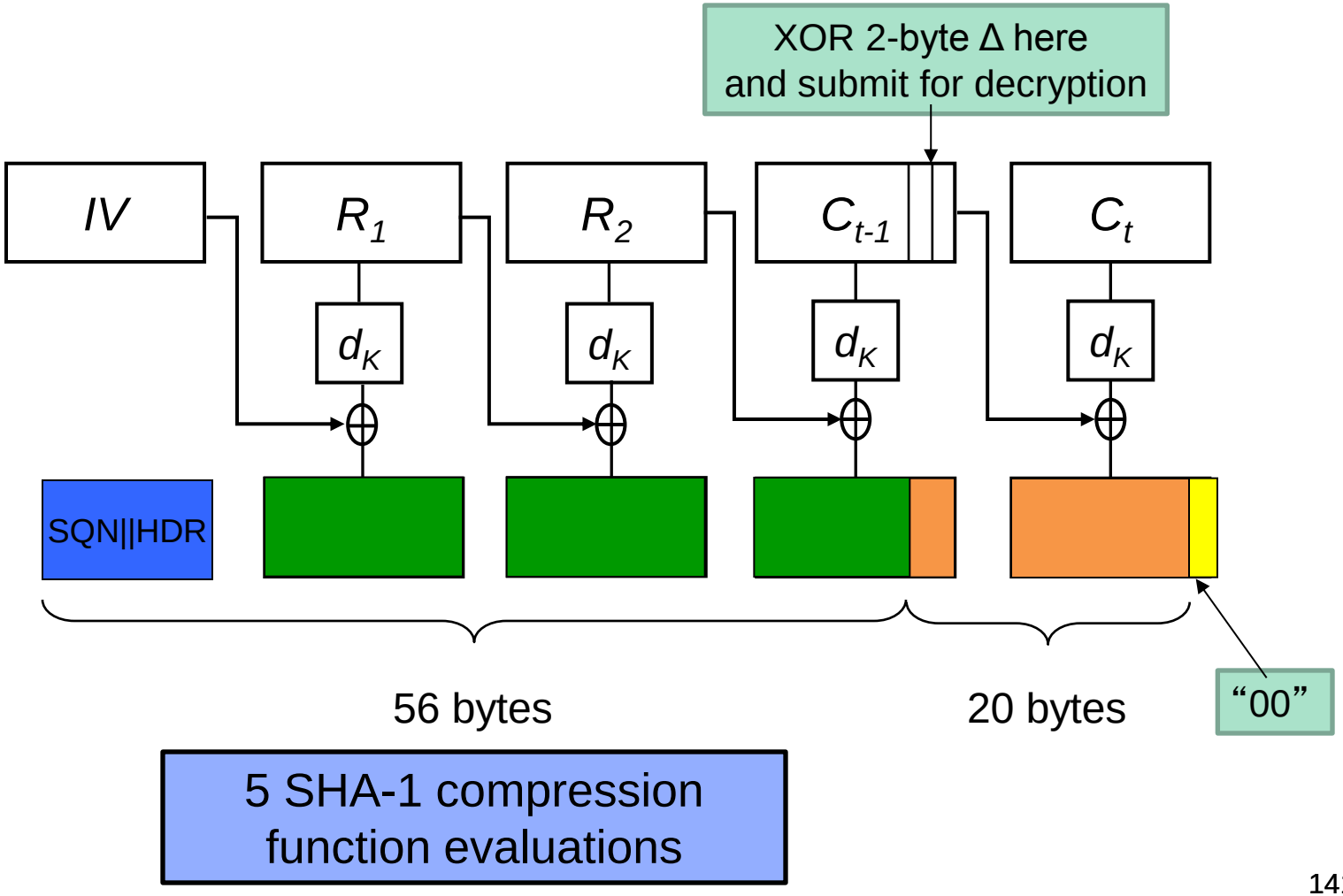


(HMAC-SHA-1 + AES-CBC)

Case 1: “01 01” (or longer valid pad)

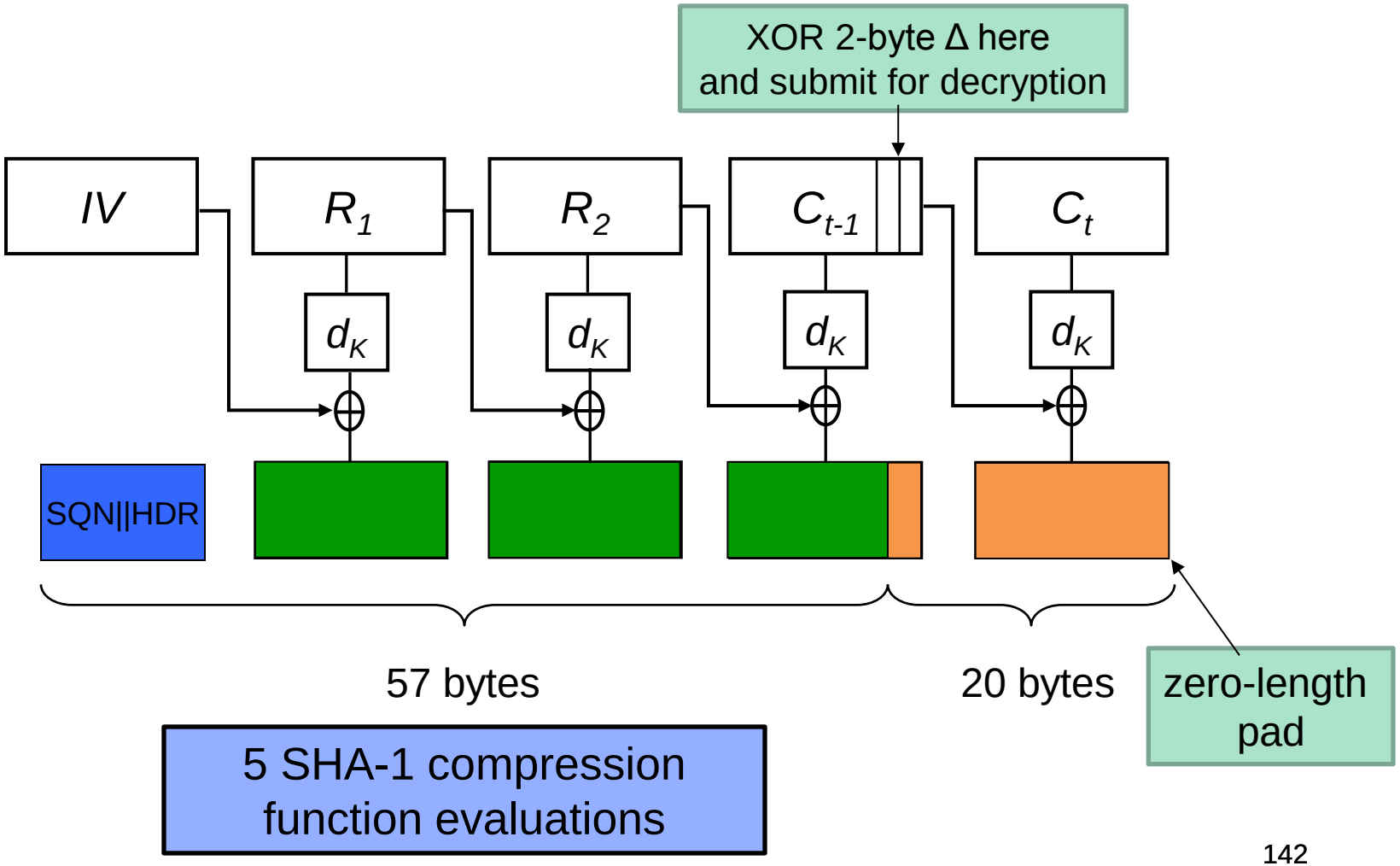


Case 2: “00”



5 SHA-1 compression function evaluations

Case 3: 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.
  - Compare with original padding oracle attack: 2ms.
- Detecting the “01 01” case allows last 2 plaintext bytes in the *target* block C_t to be recovered.
  - Using the usual CBC algebra.
  - Attack then extends easily to all bytes as in a standard padding oracle attack.

Lucky 13 – Attack Cost

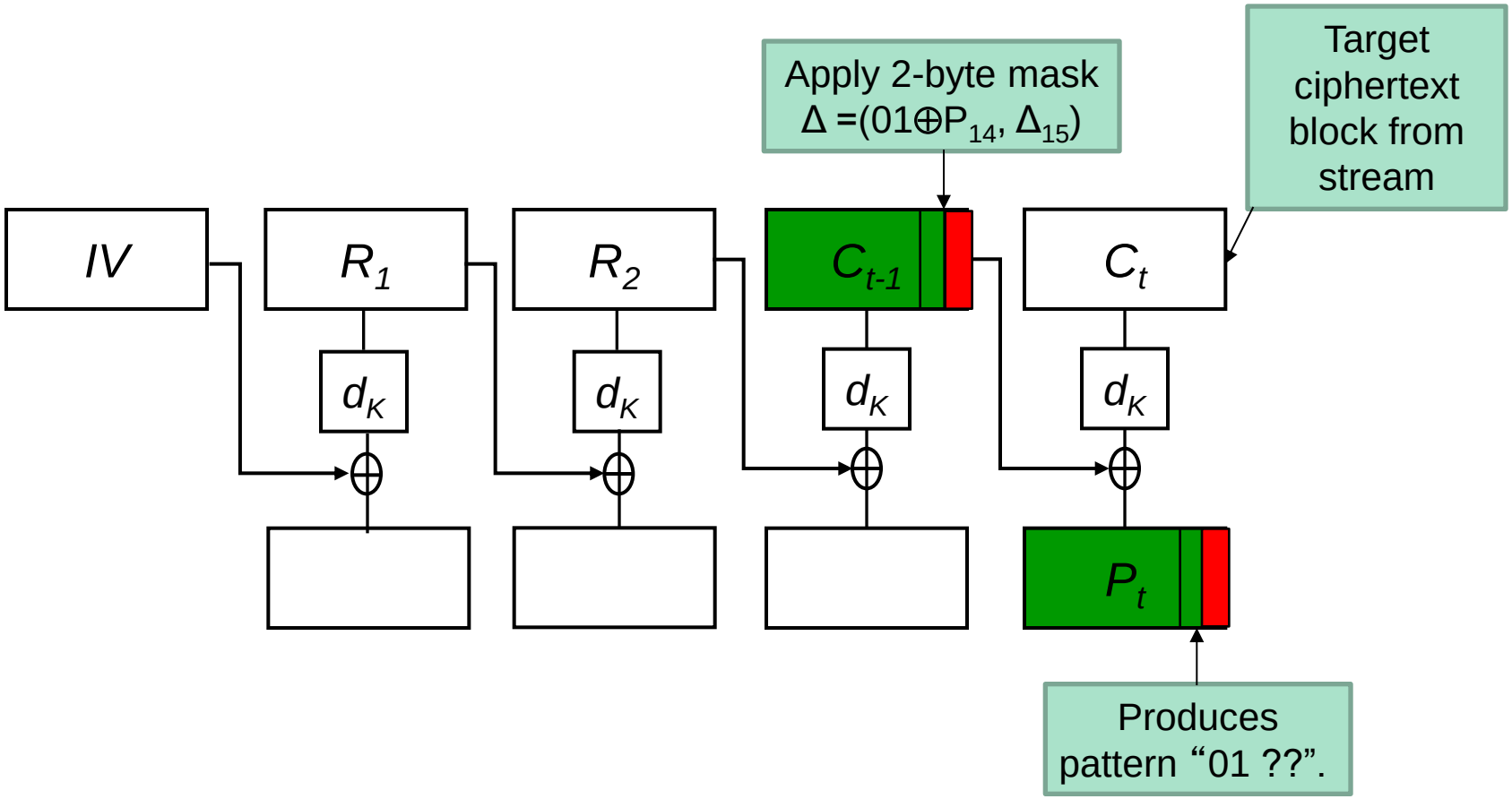
- We need 2^{16} attempts to try all 2-byte Δ values.
- And we need around 2^7 - 2^8 trials for each Δ value to reliably distinguish the different events.
  - Noise level and number of trials depends on experimental set-up.
- Each trial kills the TLS session.
- Hence the headline attack cost is 2^{23} – 2^{24} sessions, all encrypting the same plaintext.
- Looks rather theoretical?

Lucky 13 – Improvements

- If all-but-one byte of plaintext block is known, then we only need 2^8 attempts to recover the missing byte.
  - We know how to set bytes of mask Δ so that valid padding pattern is hit in all-but-one position.
  - Works for any combination of block cipher and hash function.

- 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}.

Lucky 13 – All-But-One Byte Known

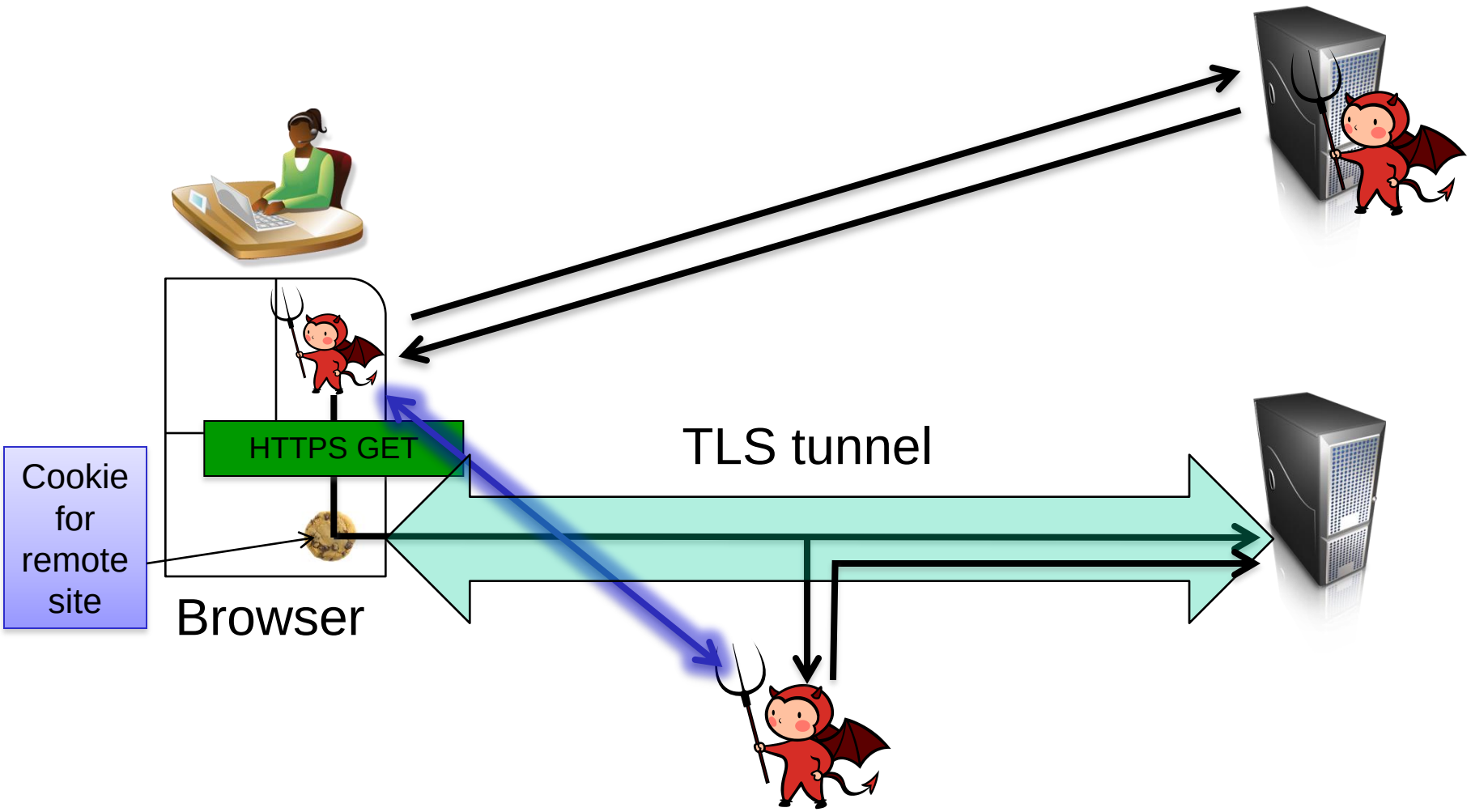


Lucky 13 + BEAST = Practical Attack

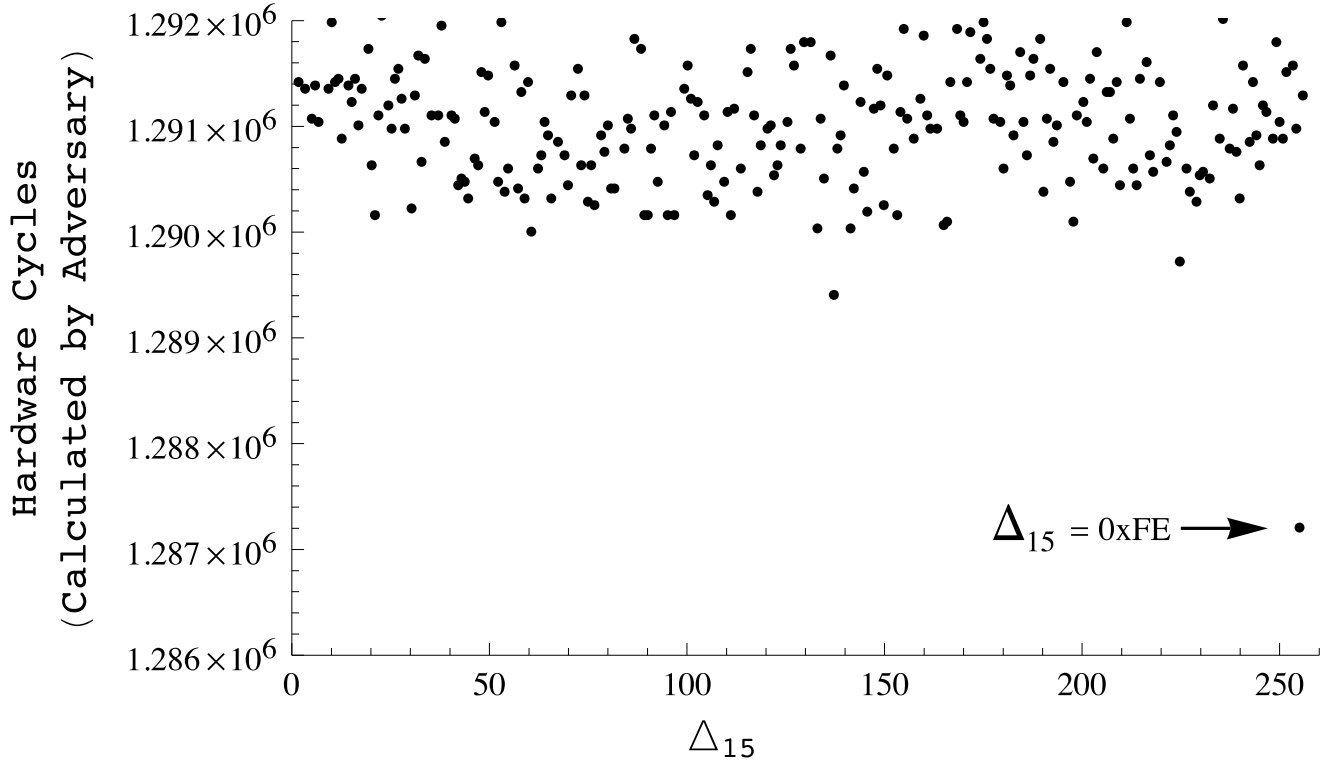
BEAST-style attack targeting HTTP cookies.

- Client-side Javascript makes repeated HTTP GET requests to target site.
- TLS sessions are automatically generated and HTTP cookies attached to outgoing GET requests.
- Javascript pads the GET requests so that all-but-one condition always holds.
  - Sliding bytes as in original BEAST attack.
- MITM modifies ciphertext.
  - Causing session crash.
- Cost of attack is around 2^{13} TLS handshakes and GET requests per byte of cookie.
- Now a practical attack!

Lucky 13 + BEAST = Practical Attack

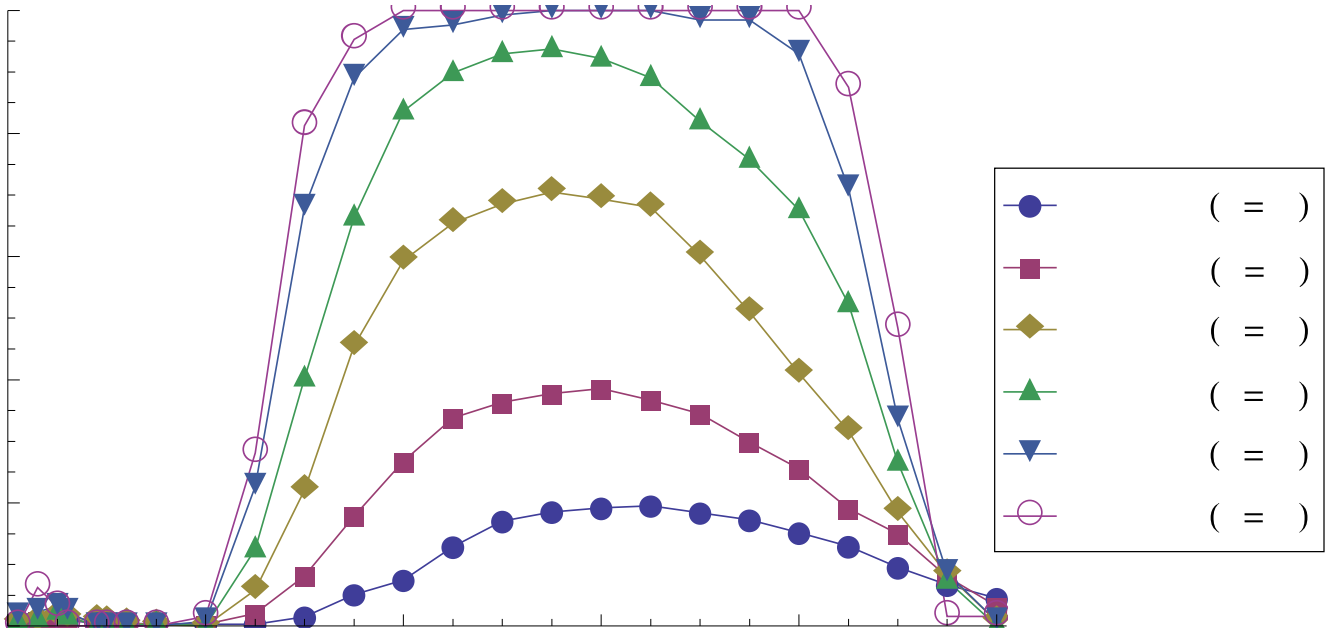


Experimental Results



- Byte 14 of plaintext set to 01; byte 15 set to FF.
- Modify Δ_{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 techniques from [AP12].]

Lucky 13 – Impact

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

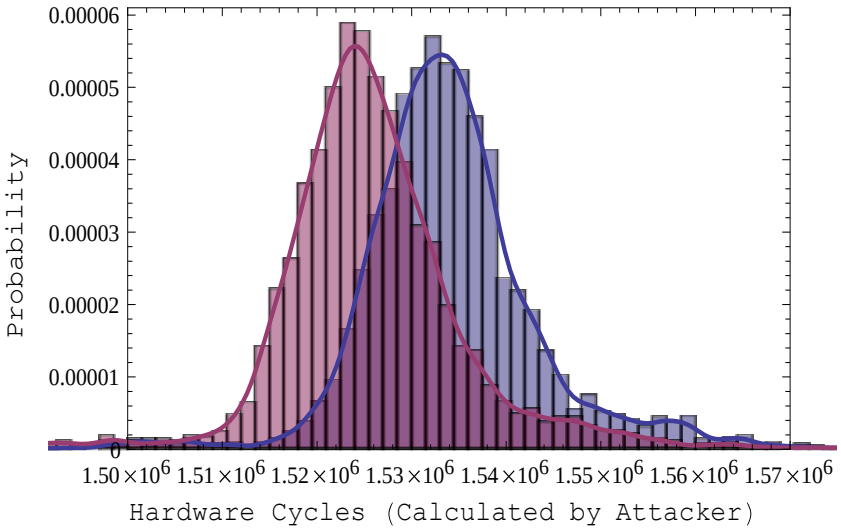
- **OpenSSL** patched in versions 1.0.1d, 1.0.0k and 0.9.8y, released 05/02/2013.
- **NSS** (Firefox, Chrome) patched in version 3.14.3, released 15/02/2013.
- **Opera** patched in version 12.13, released 30/01/2013
- **Oracle** released a special critical patch update of JavaSE, 19/02/2013.
- **BouncyCastle** 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:** patched in OS X v10.8.5 (iOS version tbd).

Lucky 13 – Countermeasures

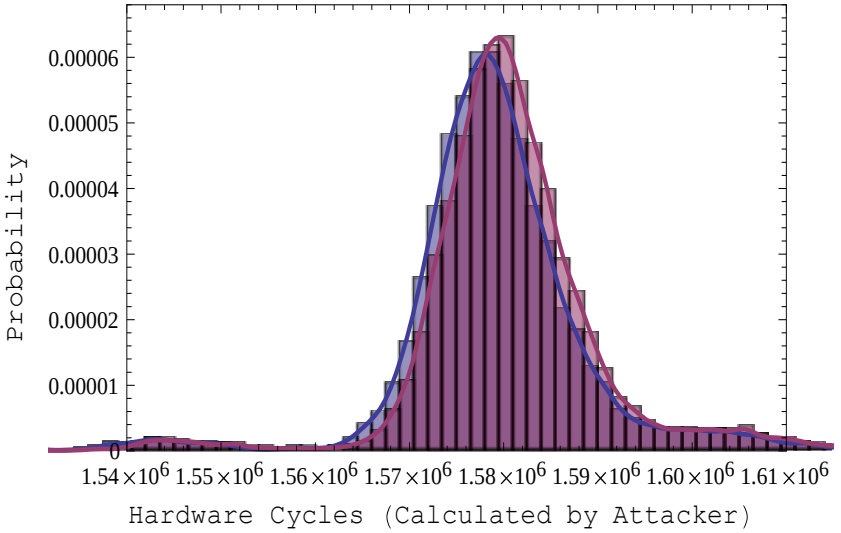
- We really need constant-time decryption for TLS-CBC.
- Add dummy hash compression function computations when padding is *good* to ensure total is the same as when padding is *bad*.
- Add dummy padding checks to ensure number of iterations done is independent of padding length and/or correctness of 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.

Performance of Countermeasures

Before



After



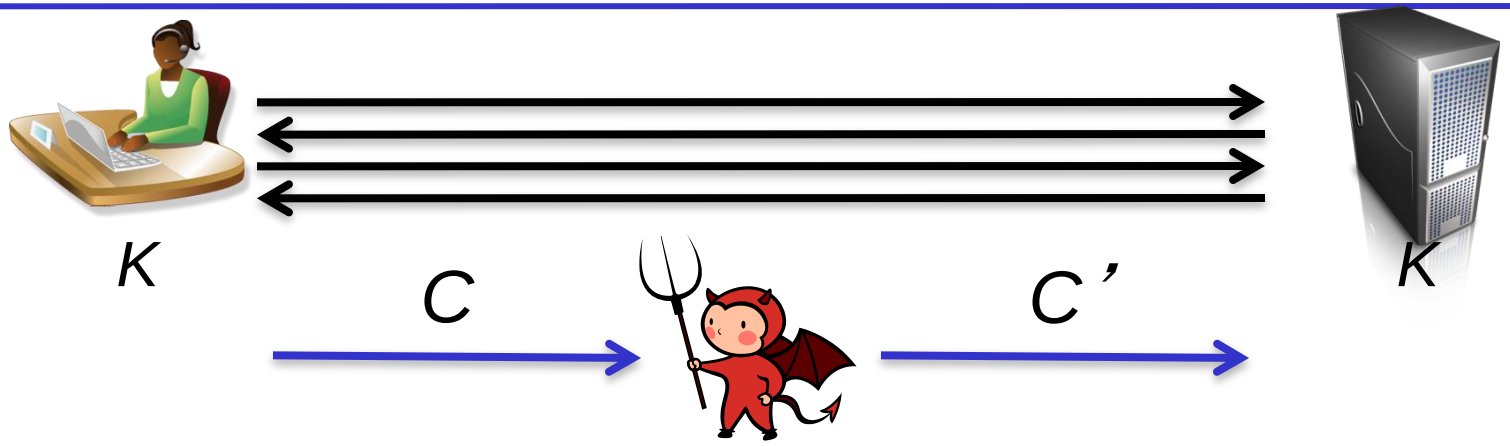
- Better but not perfect.
- Adam Langley's constant-time code in OpenSSL needed 500 lines of 'C', but completely removes difference.

Security Proofs for TLS Record Protocol (CBC mode)

Implementations of TLS in CBC mode should now have:

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

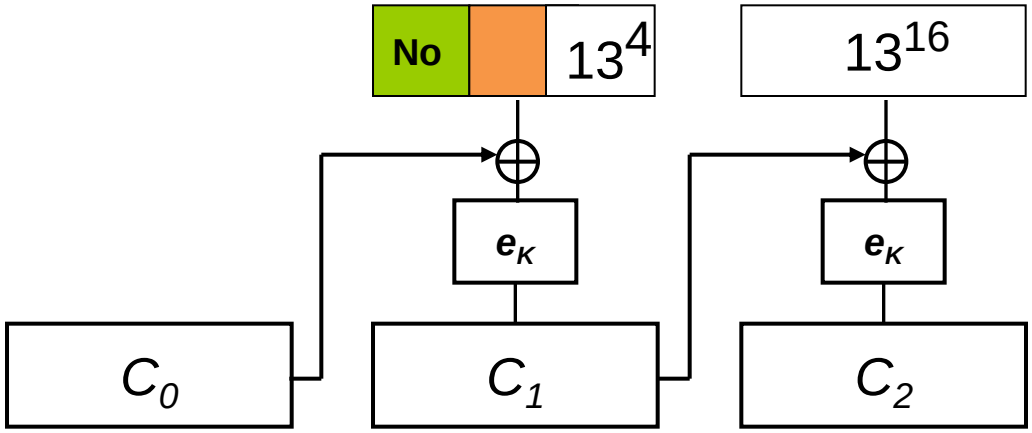
Short MAC Attack Against TLS ([PRS11])



$C = \mathrm{Enc}_K(M)$ 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”.
- A distinguishing attack.
- The attack works when MAC size < block size and when sender uses variable length padding.

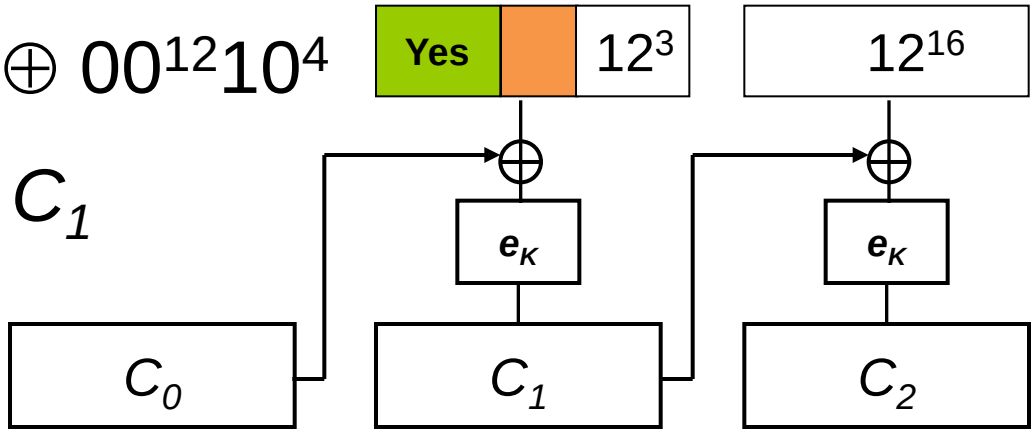
MAC length t = 80, block length n = 128



Byte 13 is hex for 19

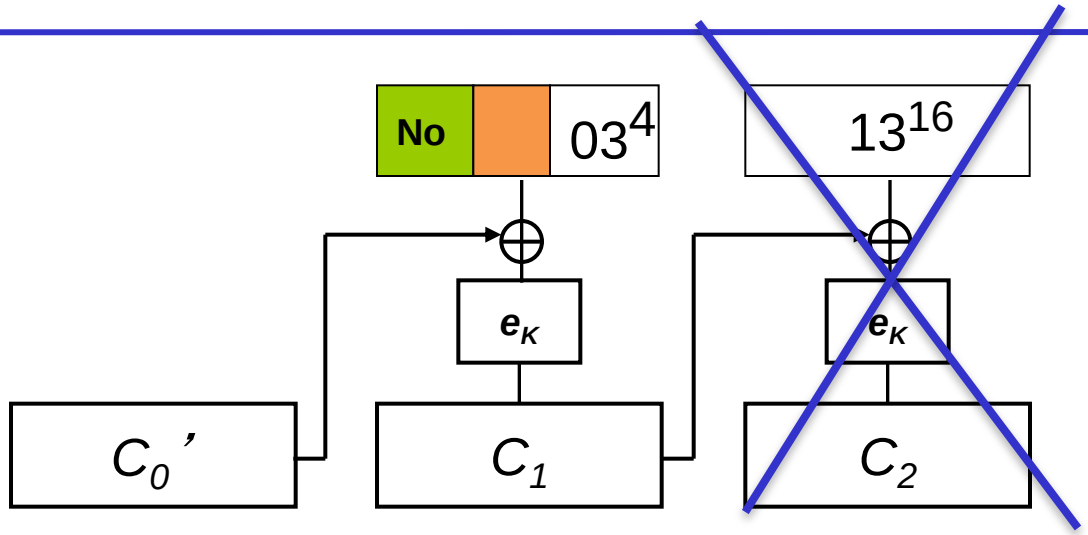
$C_0' = C_0 \oplus 00^{12}10^4$

$C' = C_0' C_1$



Byte 12 is hex for 18

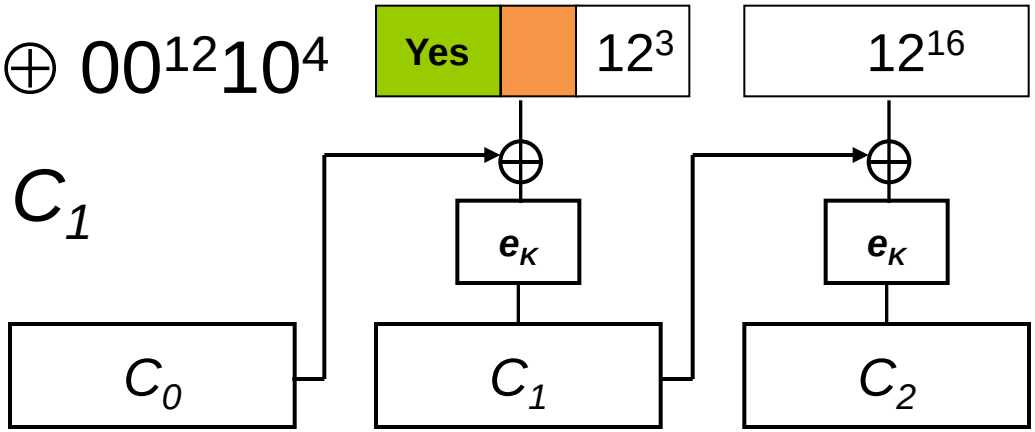
MAC length t = 80, block length n = 128



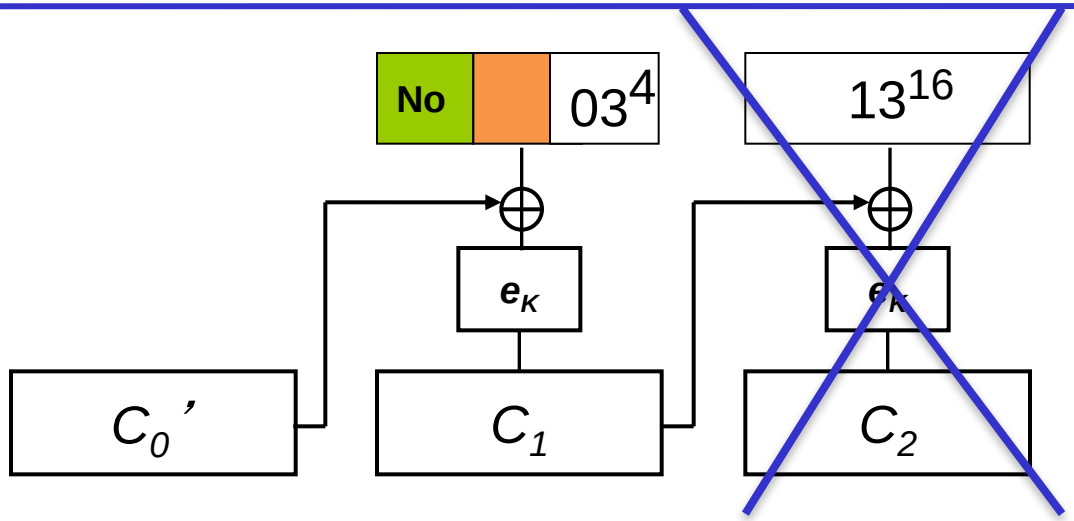
Decrypts fine to “No”

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

$C' = C_0' C_1$



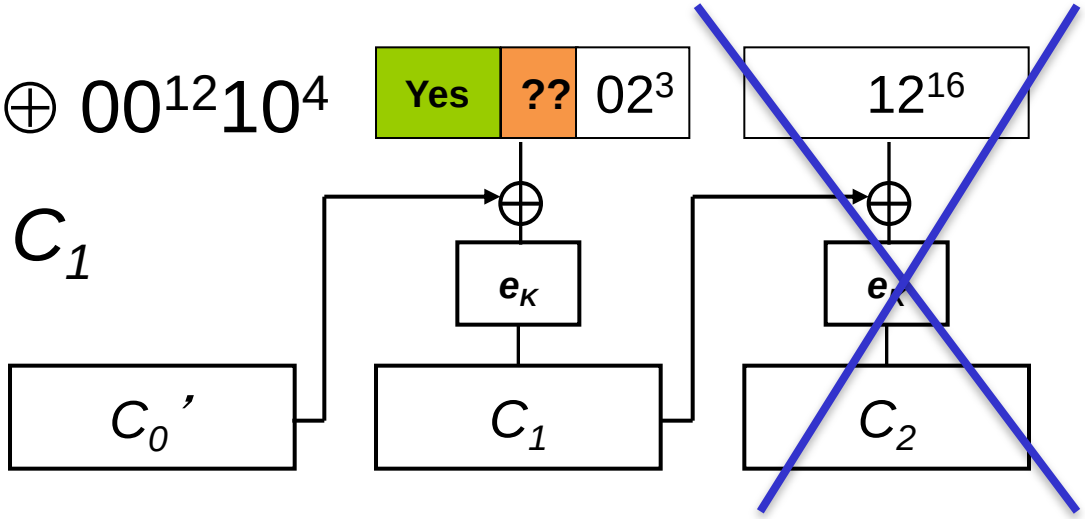
MAC length t = 80, block length n = 128



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 Attack Apply?

For TLS 1.2:

Block length

n = 64 for 3DES

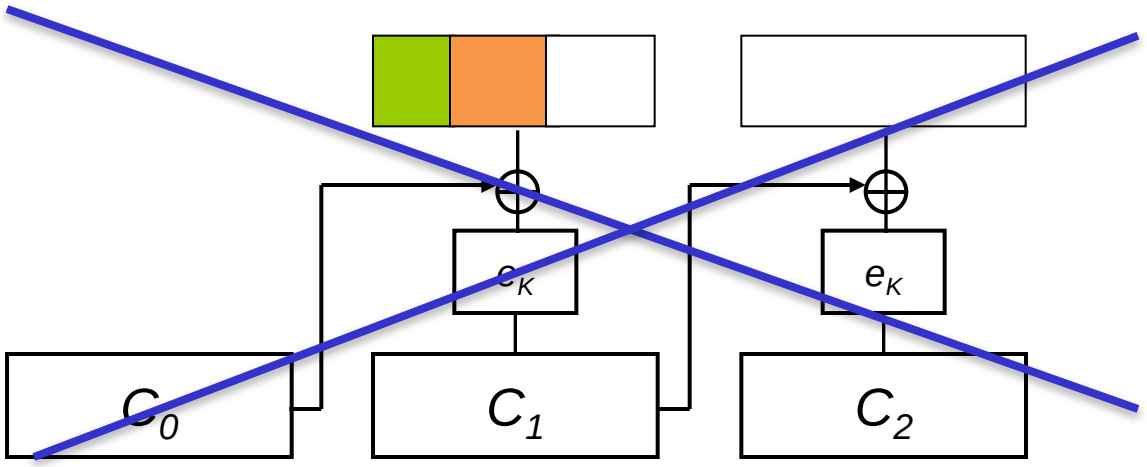
n = 128 for AES

MAC length

t = 128 for HMAC-MD5

t = 160 for HMAC-SHA1

t = 256 for HMAC-SHA256

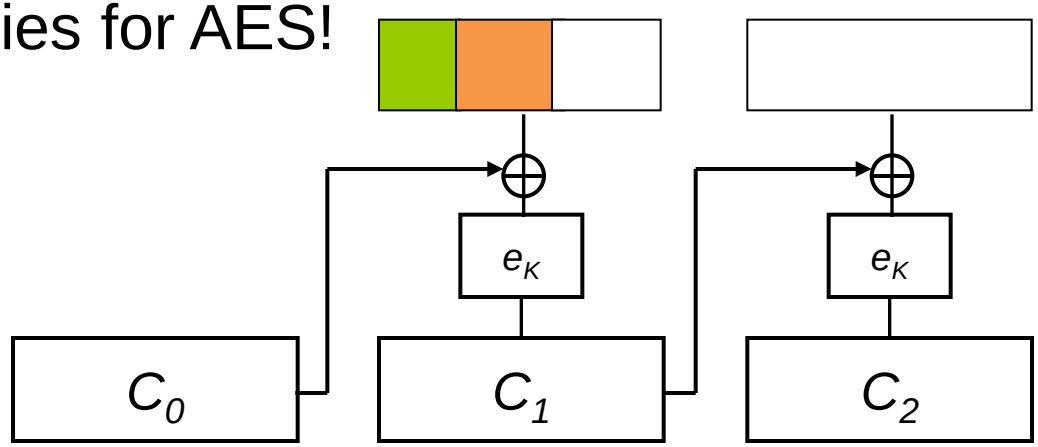


Where Does the Attack Apply?

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 for AES!



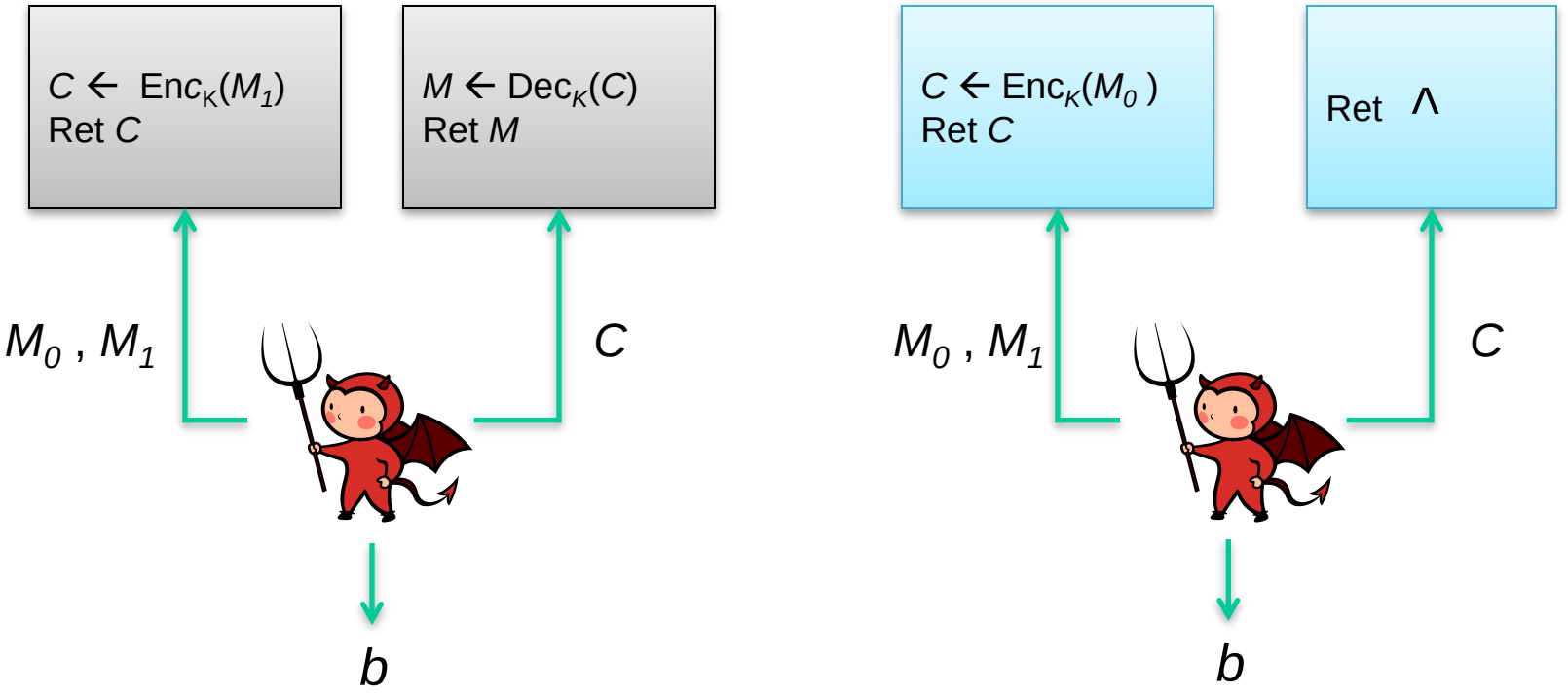
Consequences of Attack

- 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.

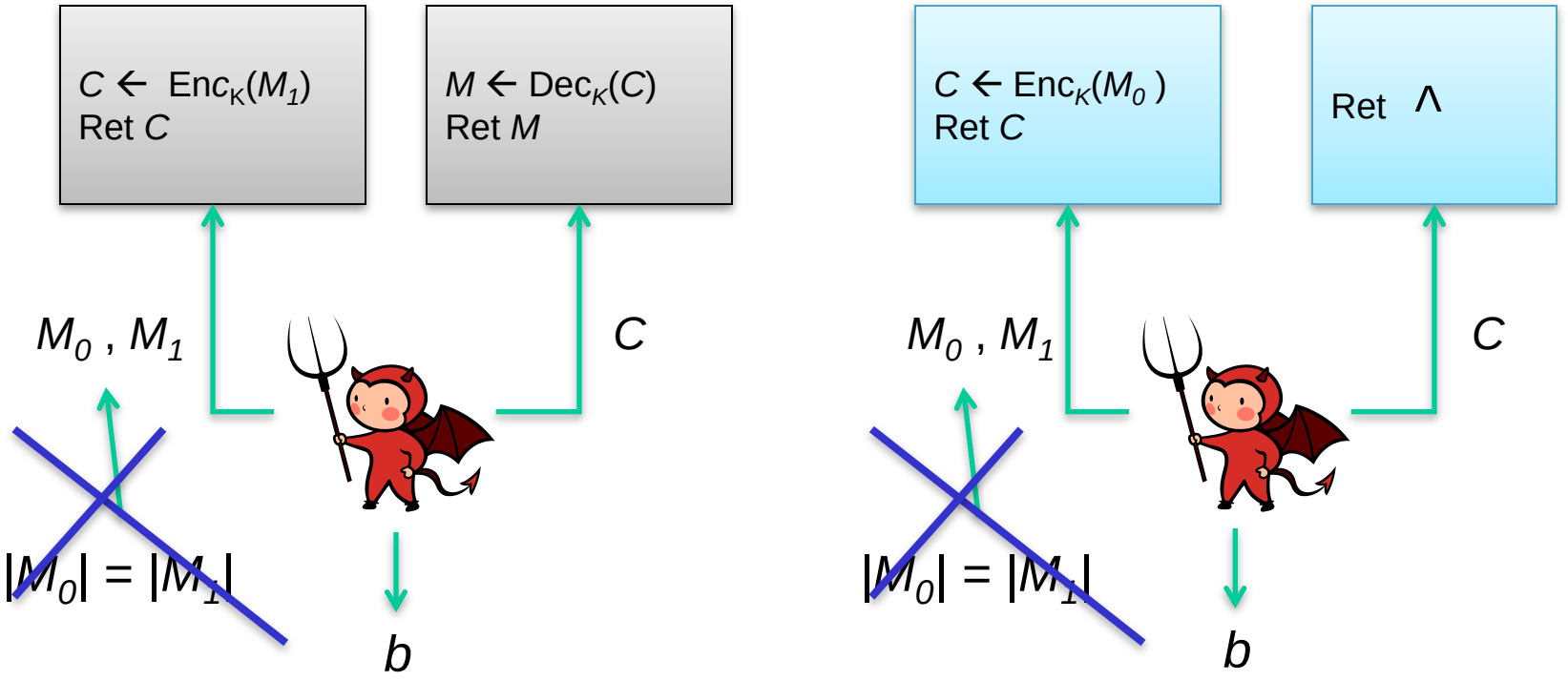
- The attack presents a barrier to obtaining proofs of security for TLS MEE construction.
  - Attack exploits variable length padding to break INT-CTXT security, leading to IND-CCA attack.

Combined AE Security Notion



AE ⇄ IND-CPA + INT-CTXT

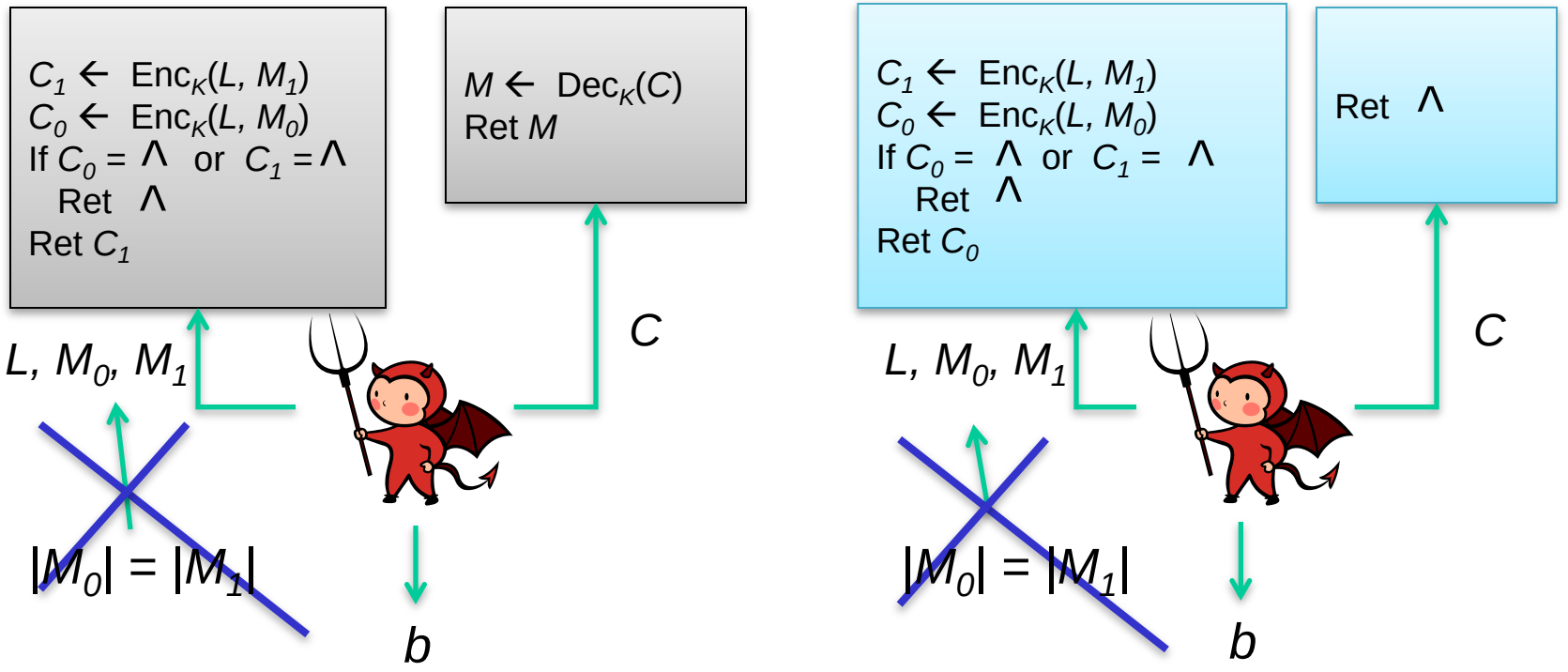
Combined AE Security Notion



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

So [PRS11] attack is outside this model.

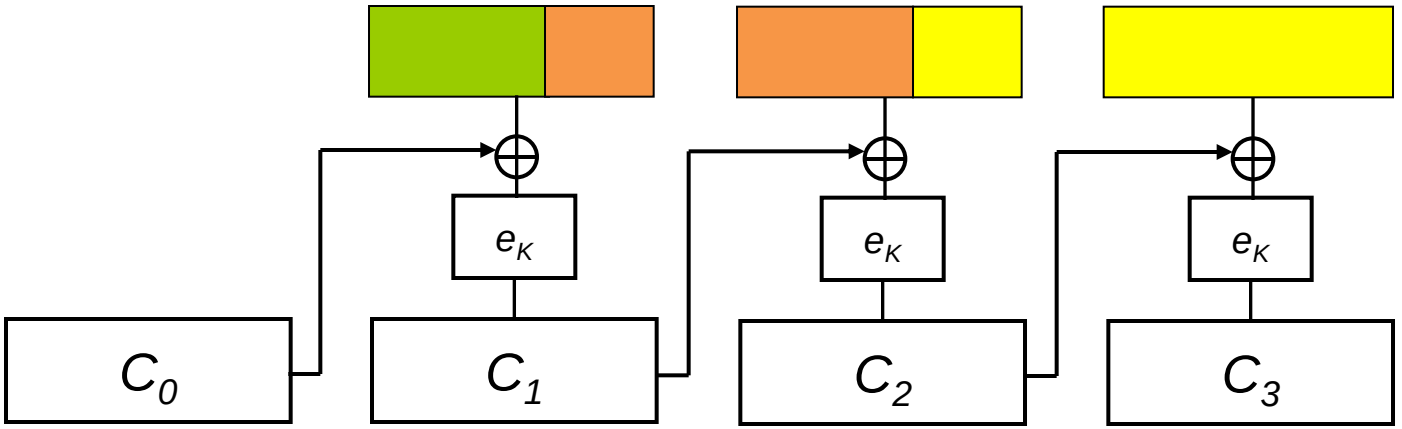
Length-hiding Authenticated Encryption (LHAE) Security



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].

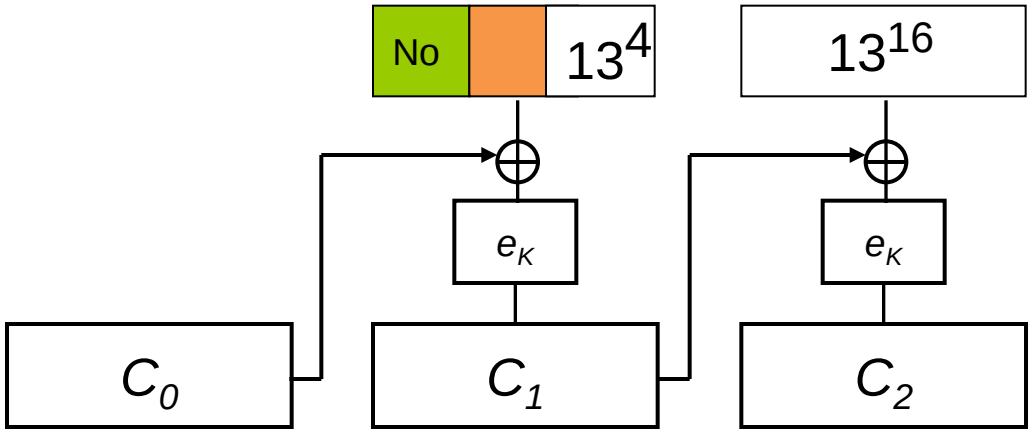
But we need INT-CTXT, and INT-PTXT does not imply it.

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.



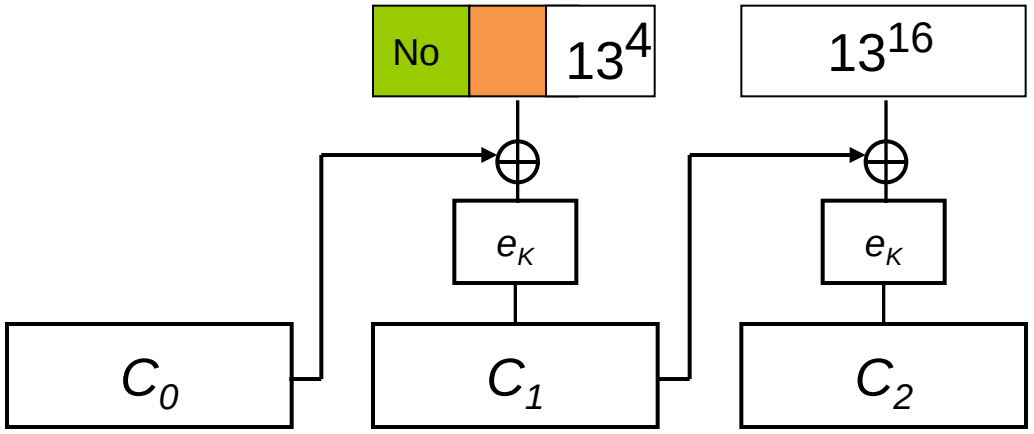
Byte 13 is hex for 19

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

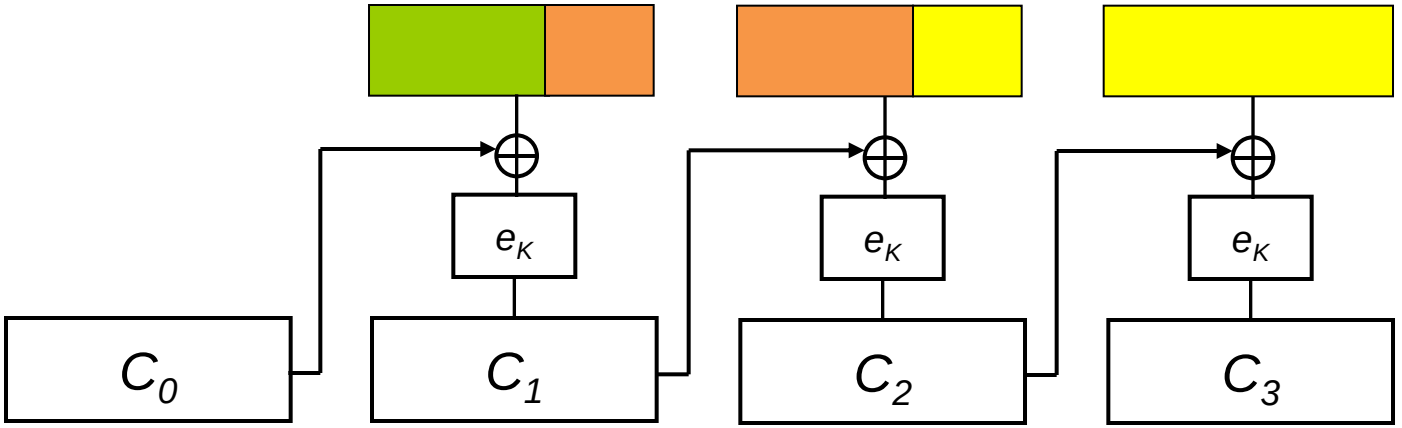
Achieving CRD security shows that no such attacks exist.

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:

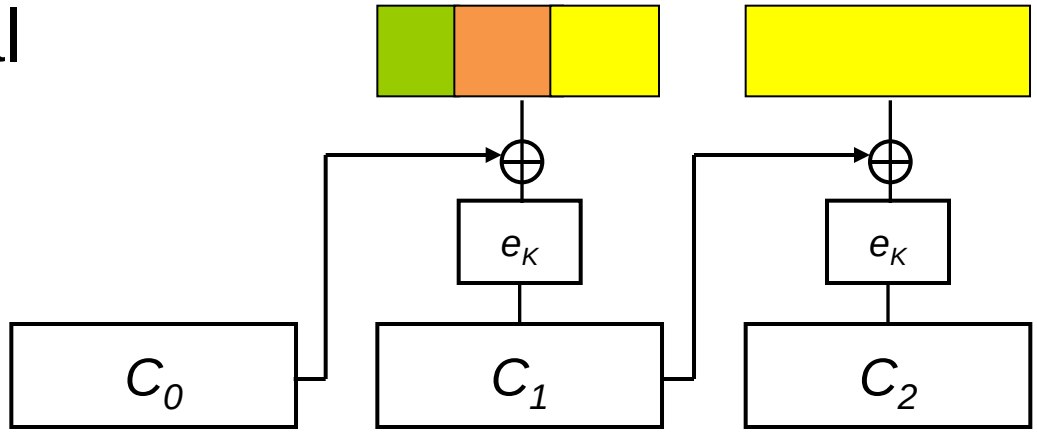
$$|M| + t \geq n.$$

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

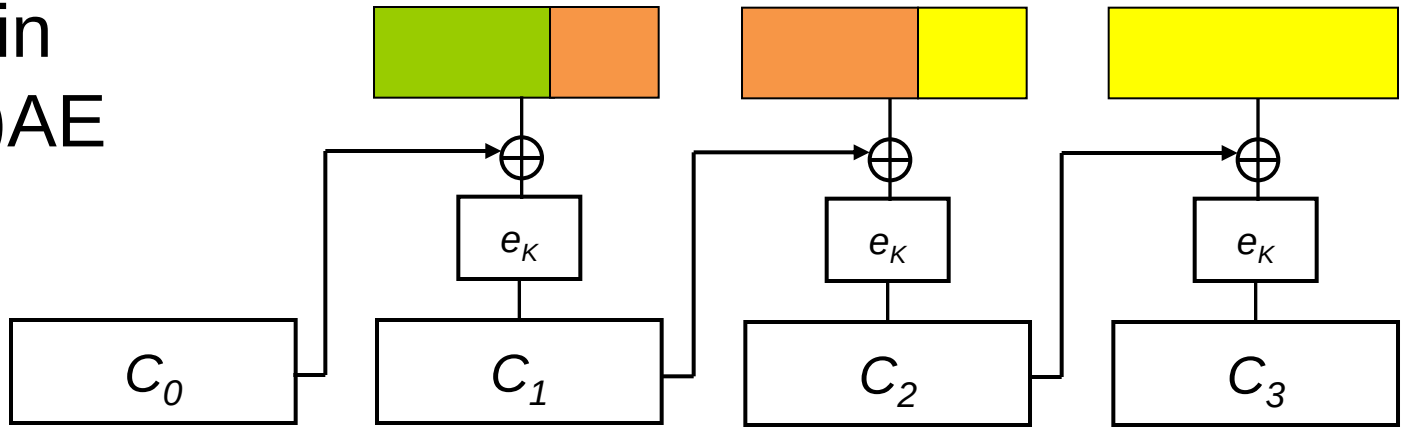


[PRS11]: Tag size matters!

Practical attacks exist



Secure in the (LH)AE model



Other Lucky 13 Countermeasures?

- Introduce random delays during decryption.
  - Surprisingly *in*effective, analysis in [AP13].
- Redesign TLS:
  - Pad-MAC-Encrypt or Pad-Encrypt-MAC?
  - Pad-Encrypt-MAC only now being adopted as a TLS extension for TLS 1.1 and higher.
  - Takes months/years to deploy.
- Switch to TLS 1.2
  - Has support for AES-GCM and AES-CCM.
  - But was not supported by browsers at time Lucky 13 was announced.
- Switch to RC4
  - As recommended by many commentators (again!).

Lessons

- TLS's 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.
  - Compare with [K01] security proof.
  - The full details of the CBC construction used in TLS were only analysed in 2011 ([PRS11]).

Lecture 4
Kenny Paterson

Bar-Ilan Winter School on Symmetric Cryptography

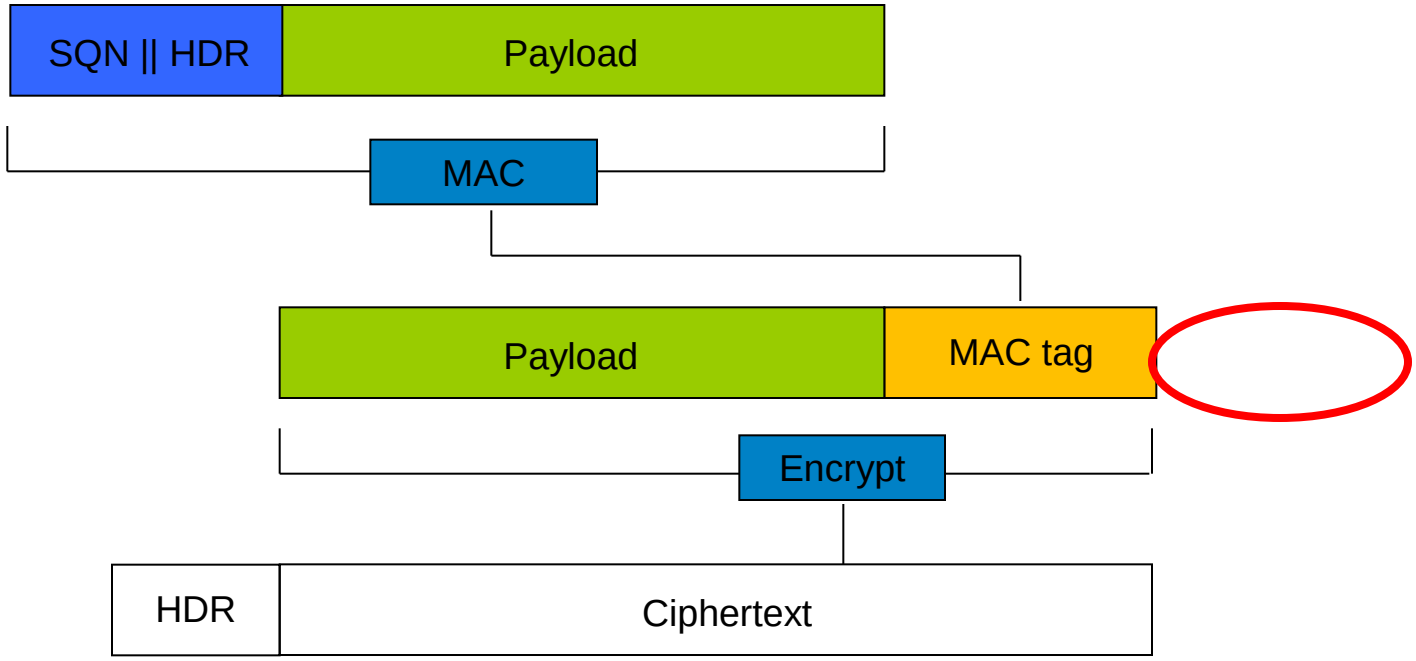
Royal Holloway
University of London

Information Security Group

Outline

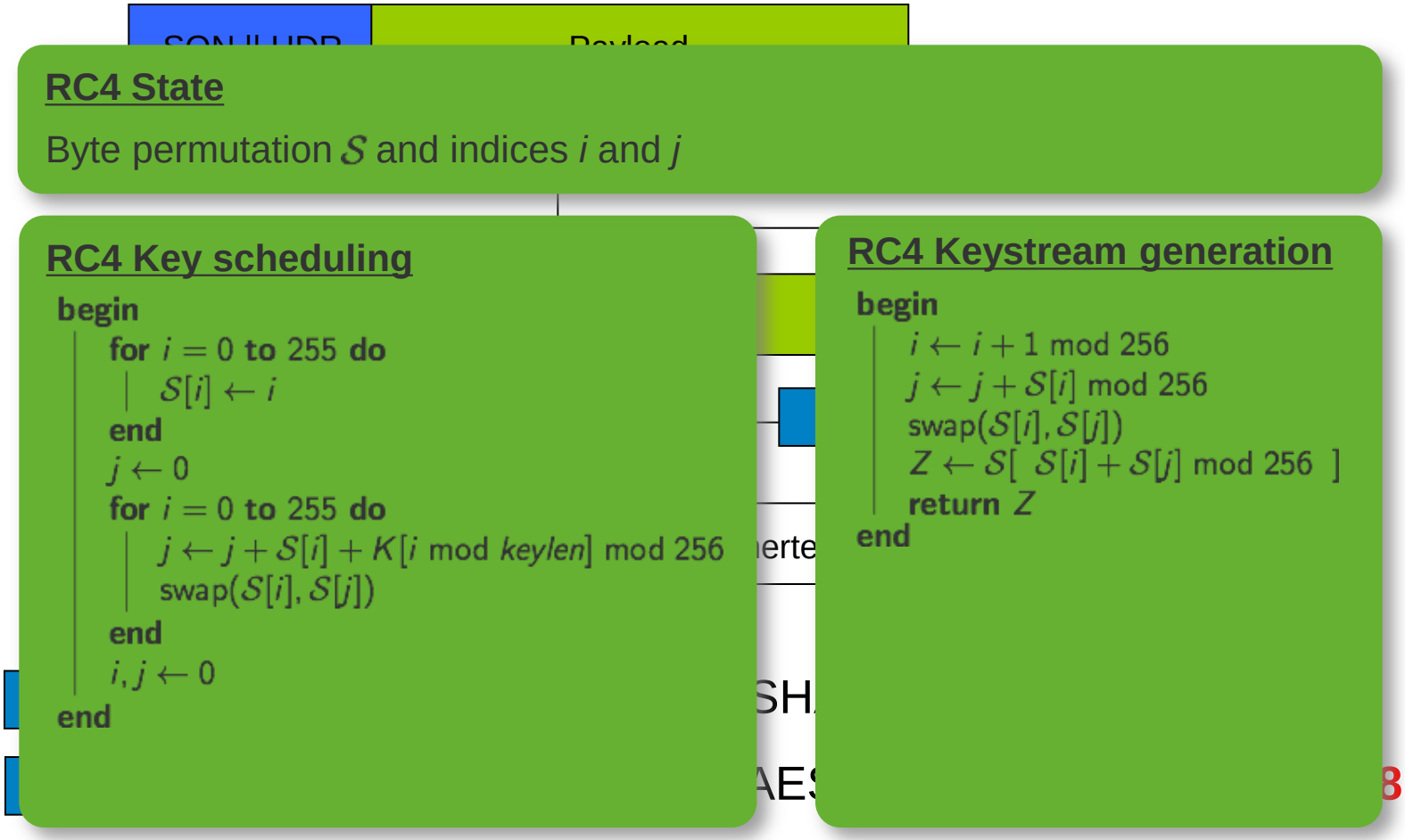
- Yet more TLS attacks
  - RC4 in TLS
  - CRIME/BREACH
- Introduction to SSH
- Security proof for SSH-CBC
- Breaking SSH-CBC
- Analysis of SSH-CTR

TLS Record Protocol: RC4-128



MAC	HMAC-MD5, HMAC-SHA1, HMAC-SHA256
Encrypt	CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

TLS Record Protocol: RC4-128



Use of RC4 in TLS

- In the face of the BEAST and Lucky 13 attacks on CBC-based ciphersuites in TLS, switching to RC4 was a recommended mitigation.

- RC4 is also fast when AES hardware not available

- Use of RC4 in the wild:

ICSI Certificate Notary

Jan. 2013 survey of 16 billion TLS connections: Approx. 50% protected via RC4 ciphersuites

- Problem: RC4 is known to have statistical weaknesses.

Single-byte Biases in the RC4 Keystream

Z_i = value of *i*-th keystream byte

- [Mantin-Shamir 2001]:

$$\Pr[Z_2 = 0] \approx \frac{1}{128}$$

- [Mironov 2002]:
  - Described distribution of Z_1 (bias away from 0, sine-like distribution)

- [Maitra-Paul-Sen Gupta 2011]: for $3 \leq r \leq 255$

$$\Pr[Z_r = 0] = \frac{1}{256} + \frac{c_r}{256^2} \qquad 0.242811 \leq c_r \leq 1.337057$$

- [Sen Gupta-Maitra-Paul-Sarkar 2011]:

$$\Pr[Z_l = 256 - l] \geq \frac{1}{256} + \frac{1}{256^2}$$

l = keylength

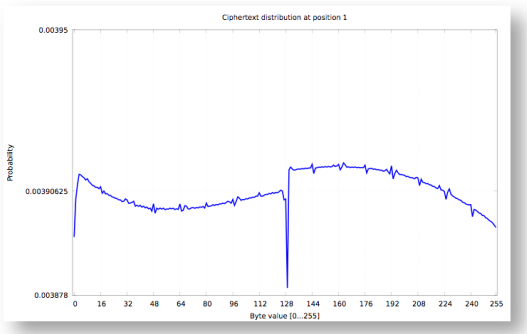
What's going on?

- Why were we all still using RC4 in half of all TLS connections when we knew it was broken?

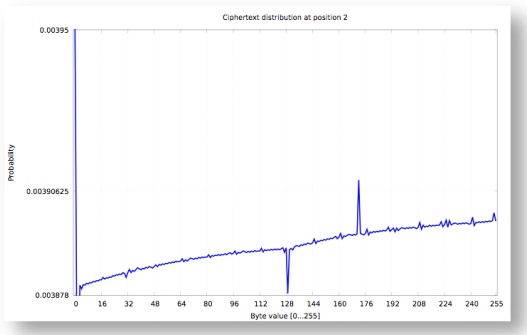
- "Google uses it, so it must be OK for my site".
- "The biases are only in the first handful of bytes and they don't encrypt anything interesting in TLS".
- "The biases are not exploitable in any meaningful scenario".
- "RC4 is fast."
- "I'm worried about BEAST on CBC mode."

Complete Keystream Byte Distributions

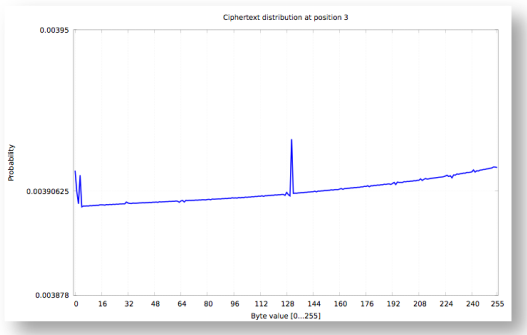
- Approach in [ABPPS13]:
  - Based on the output from 2^{45} random independent 128-bit RC4 keys, estimate the keystream byte distribution of the first 256 bytes ..



Z_1



Z_2

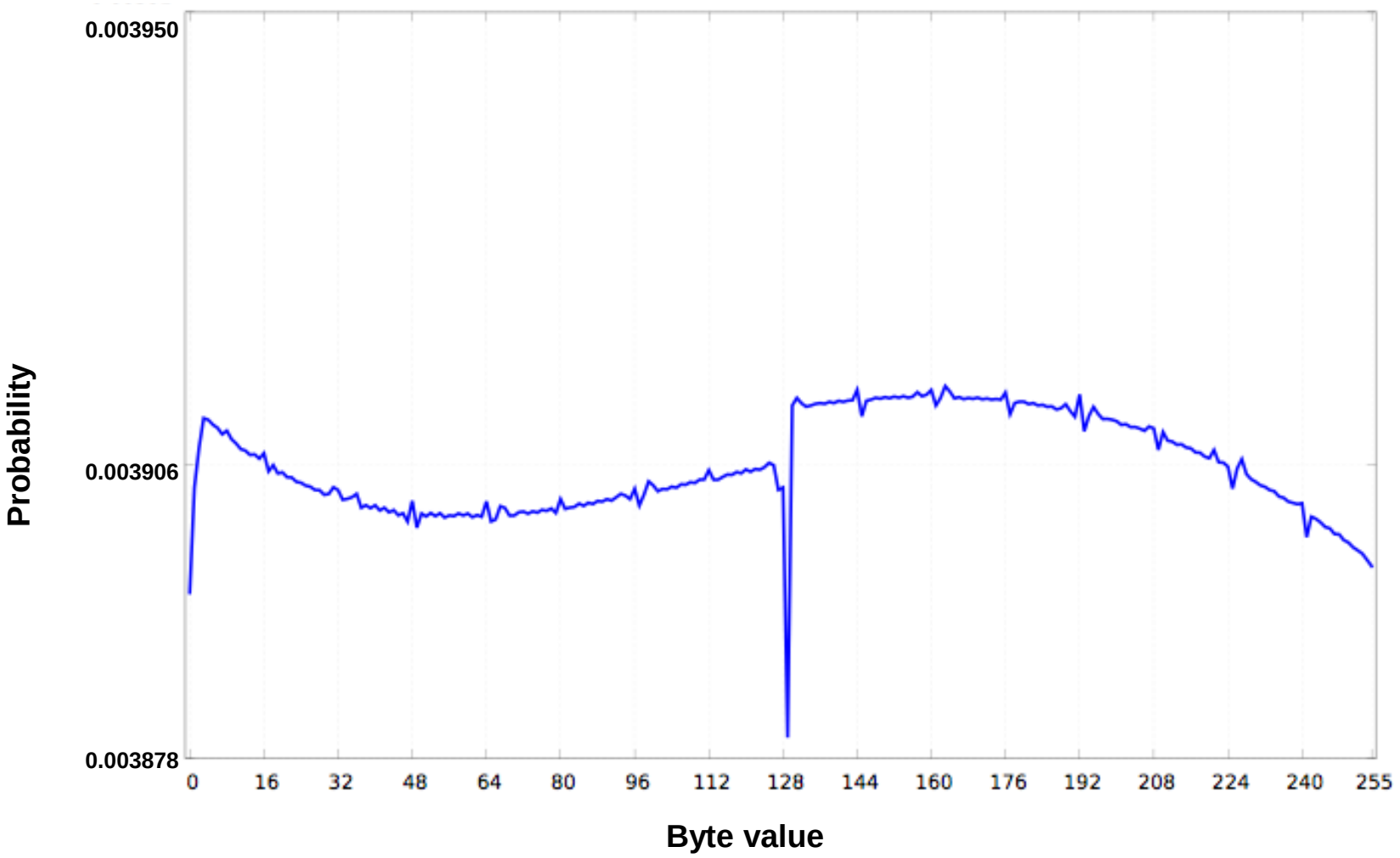


Z_3

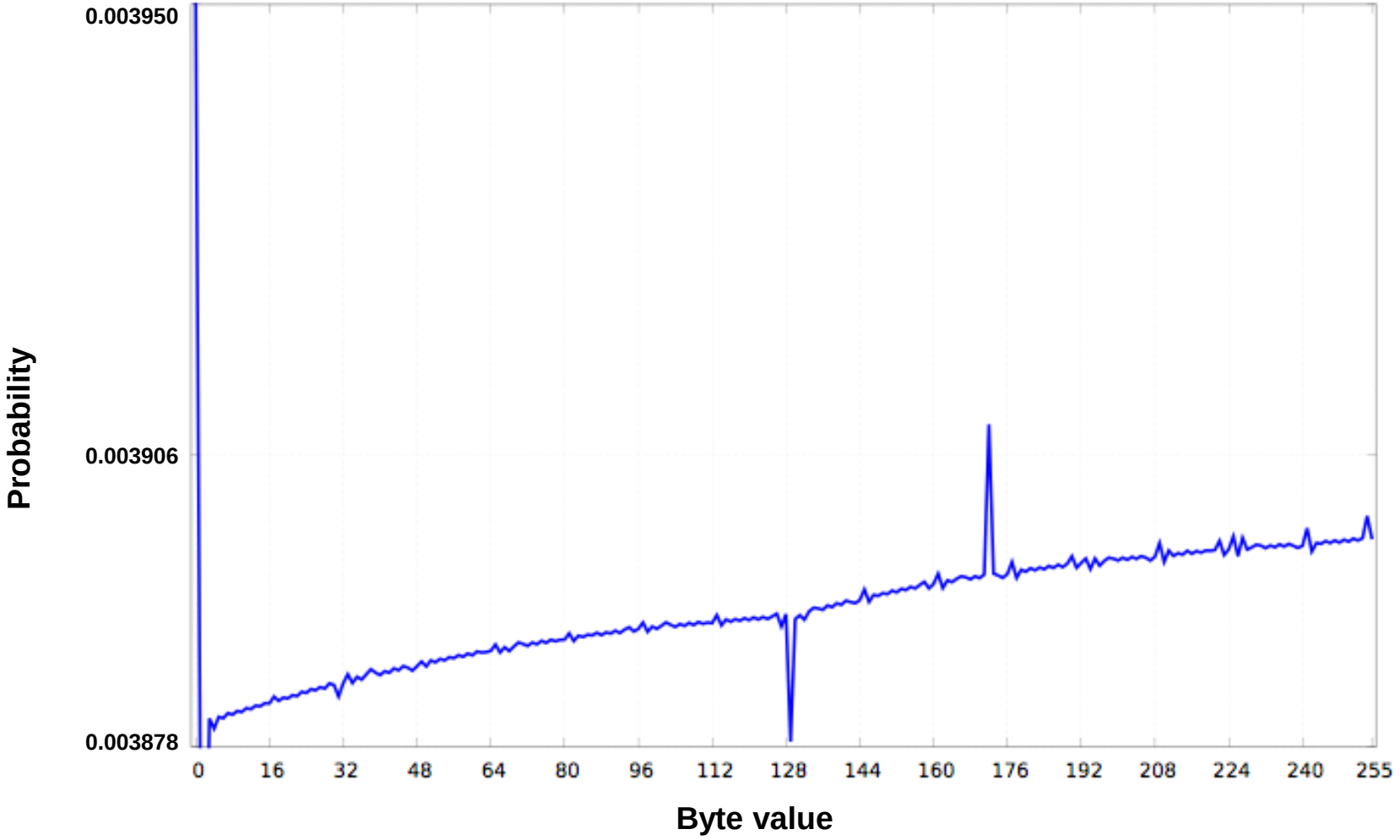
... ...

- Revealed many new biases in the RC4 keystream.
  - (Some of these were independently discovered by Isobe *et al*.)

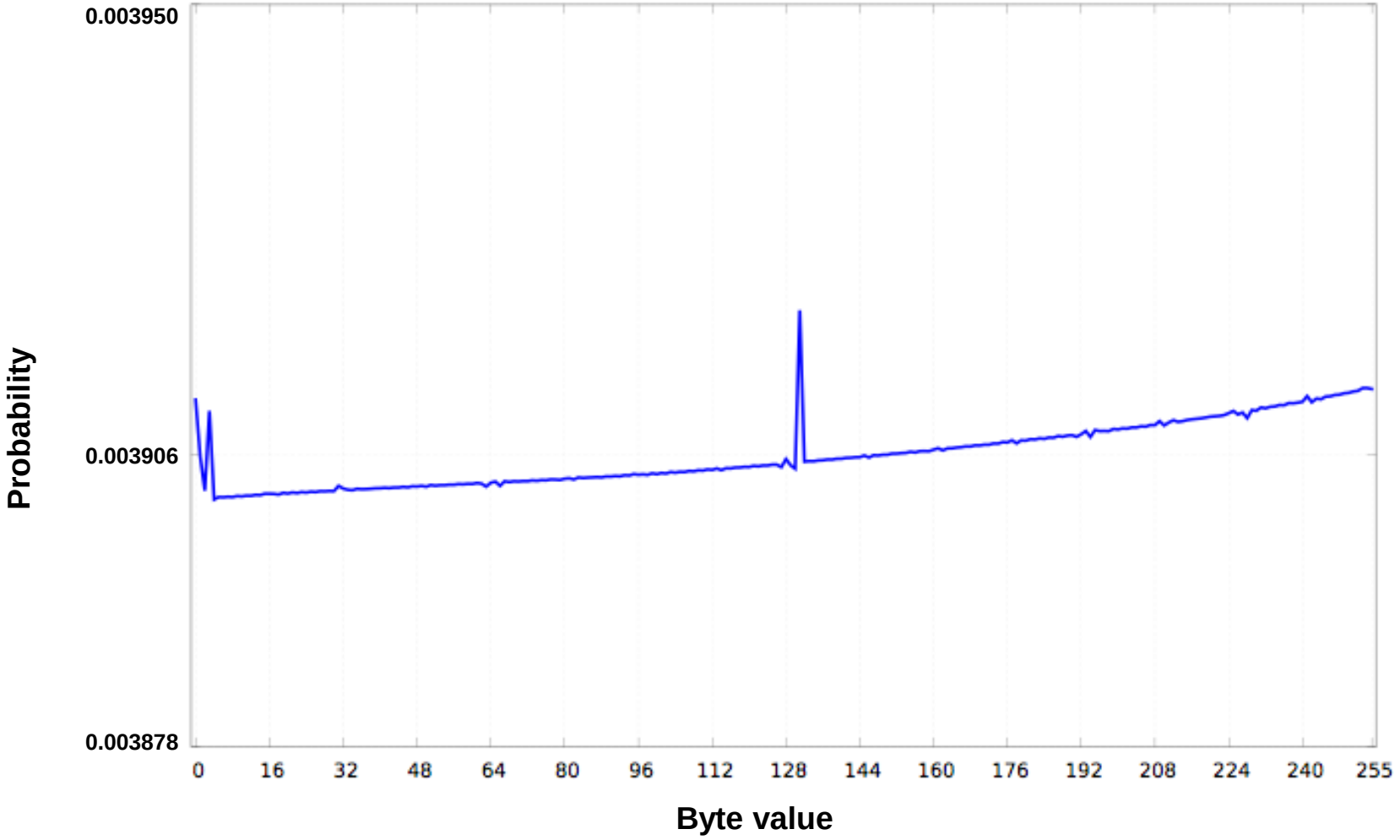
Keystream Distribution at Position 1



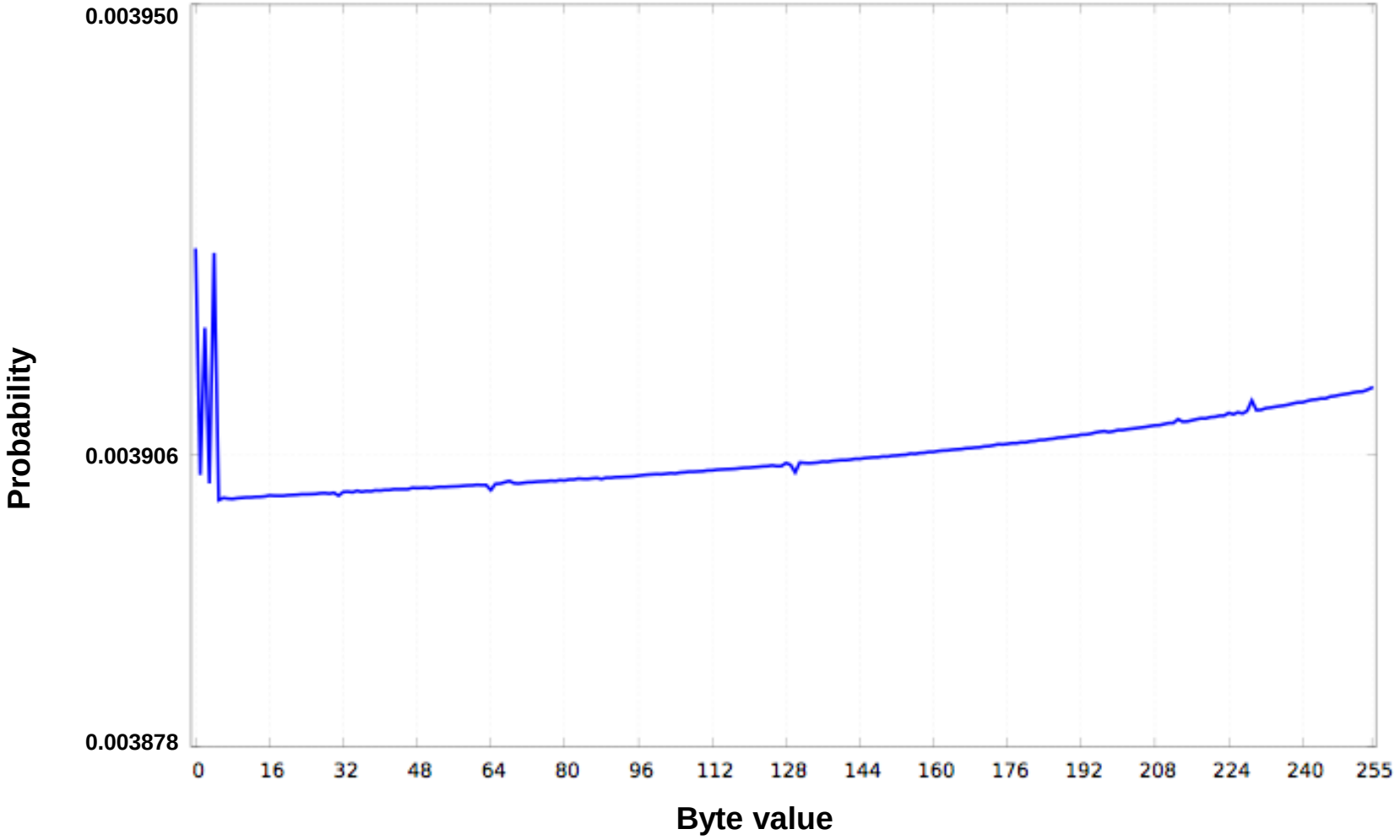
Keystream Distribution at Position 2



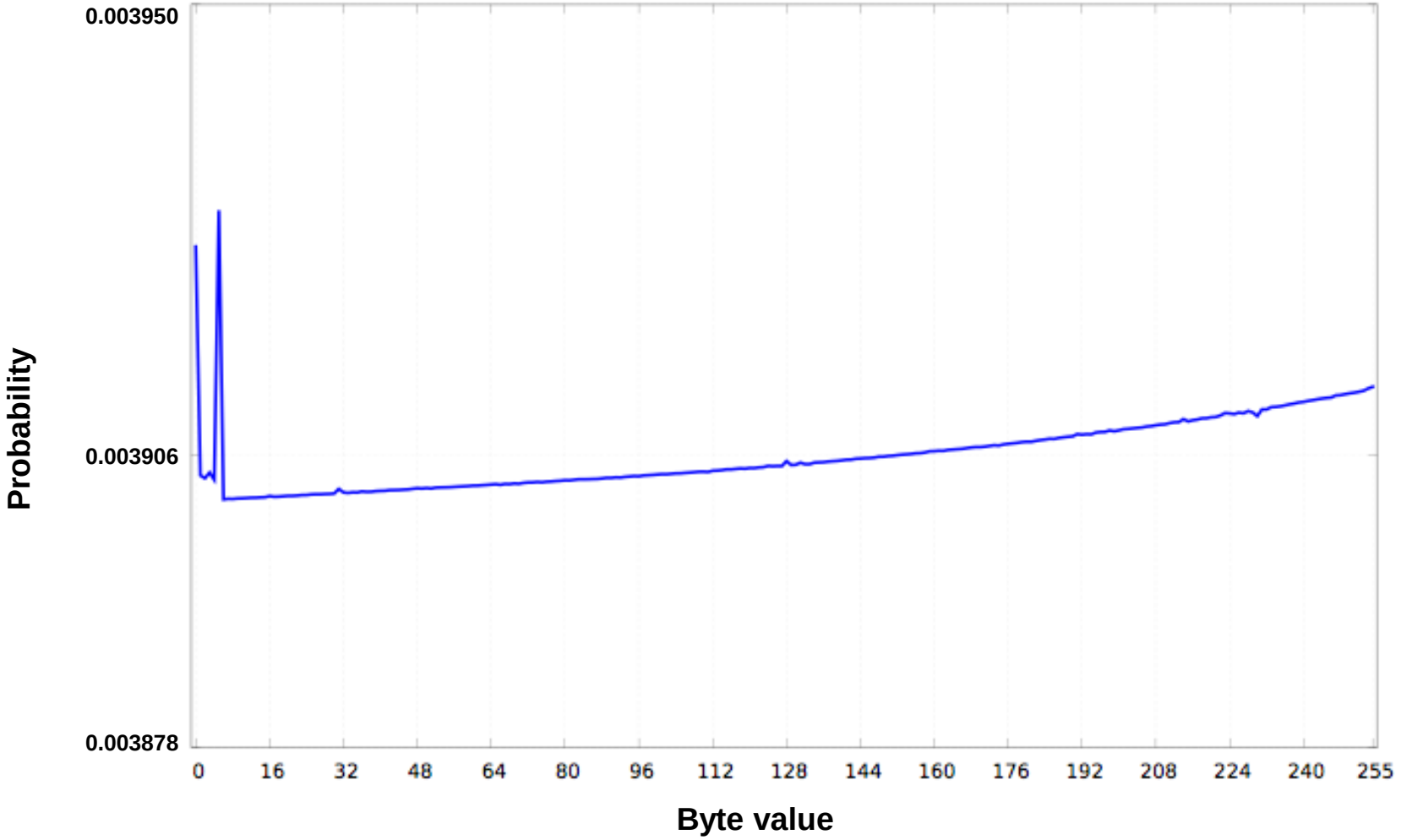
Keystream Distribution at Position 3



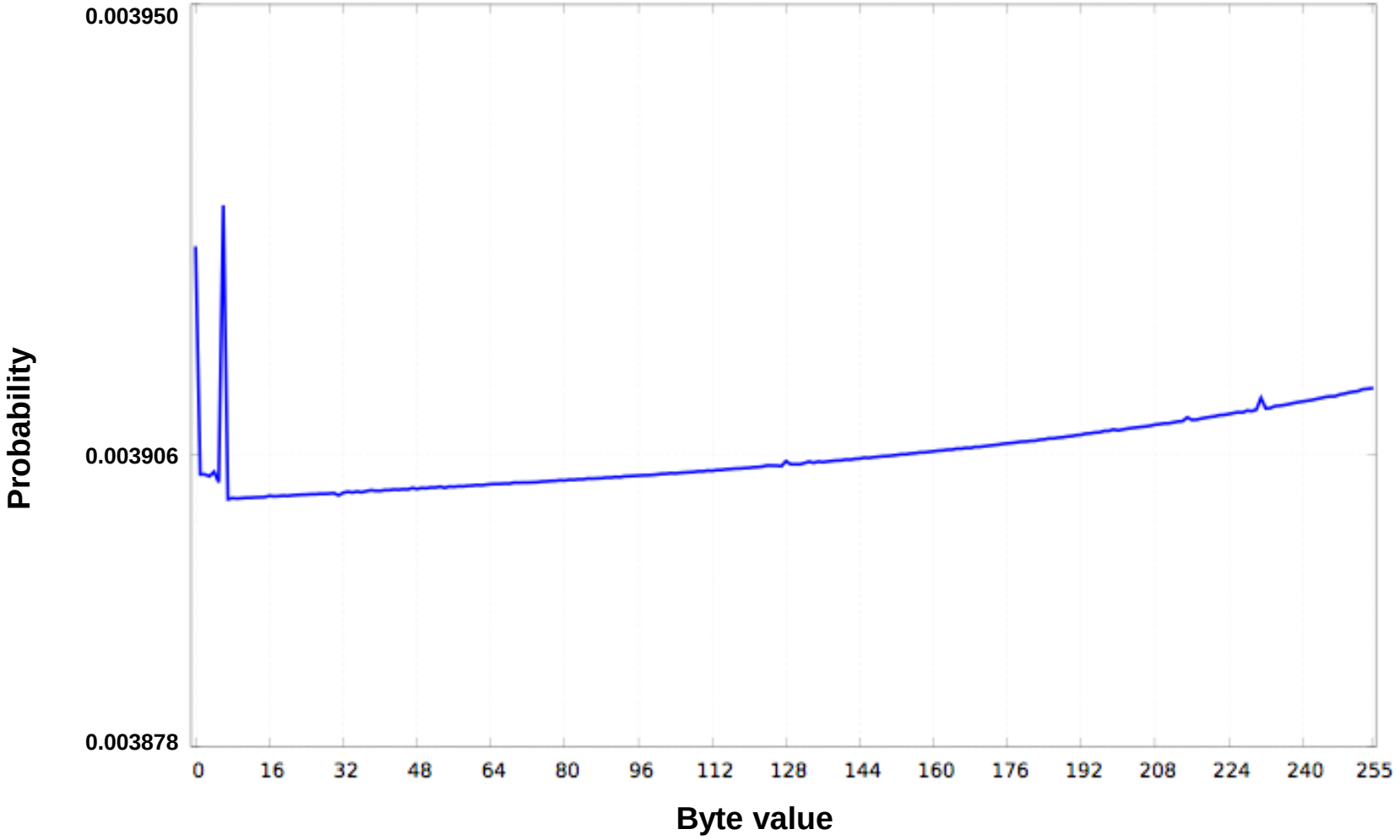
Keystream Distribution at Position 4



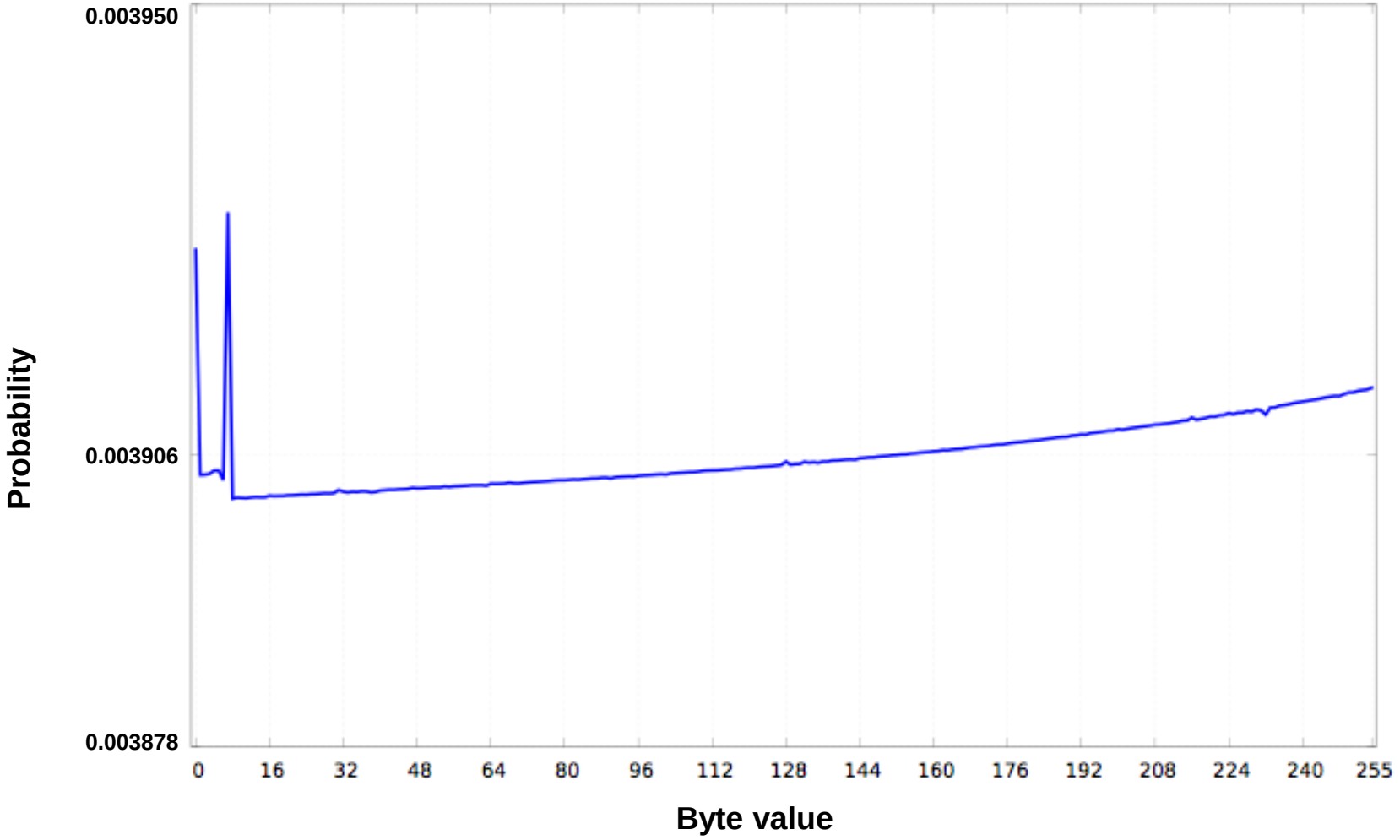
Keystream Distribution at Position 5



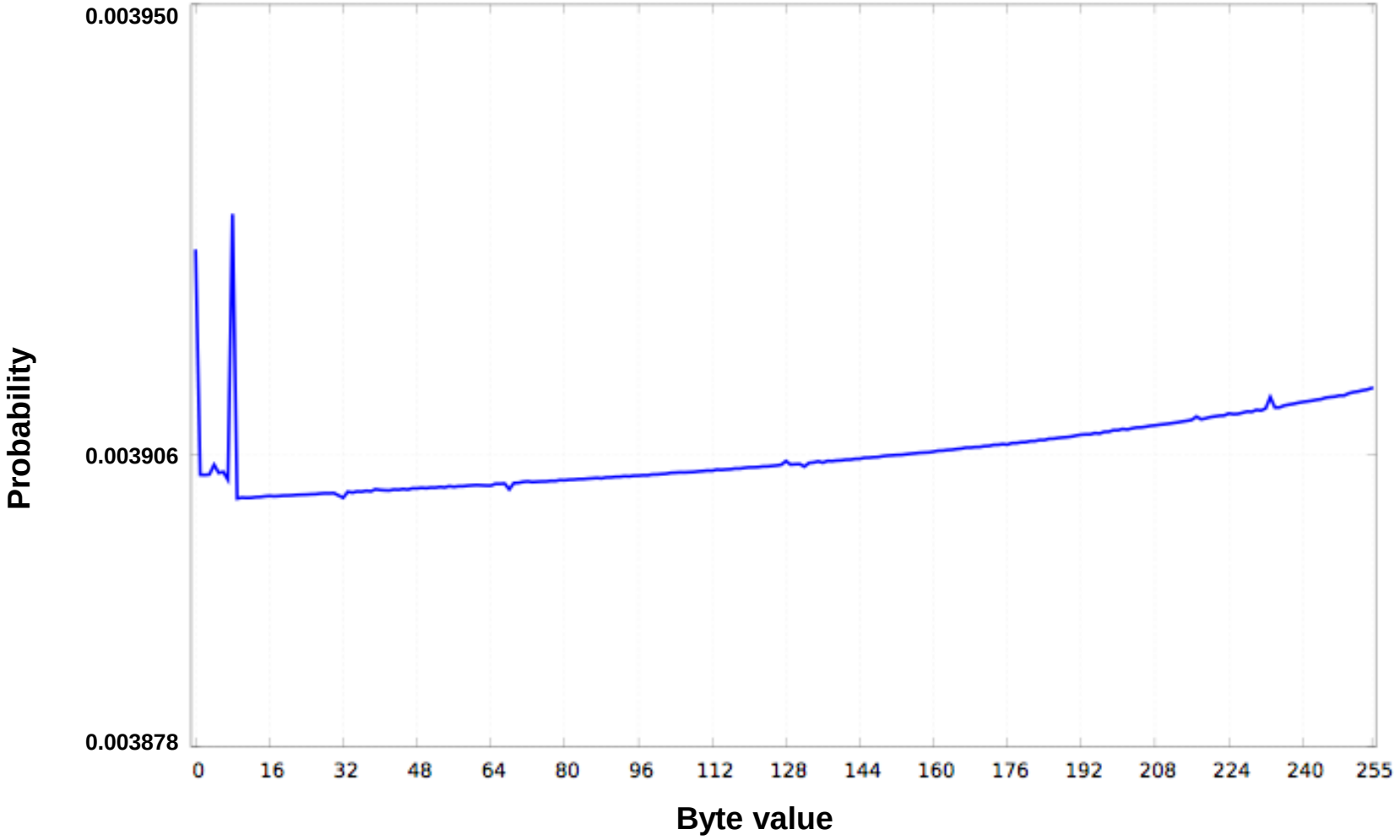
Keystream Distribution at Position 6



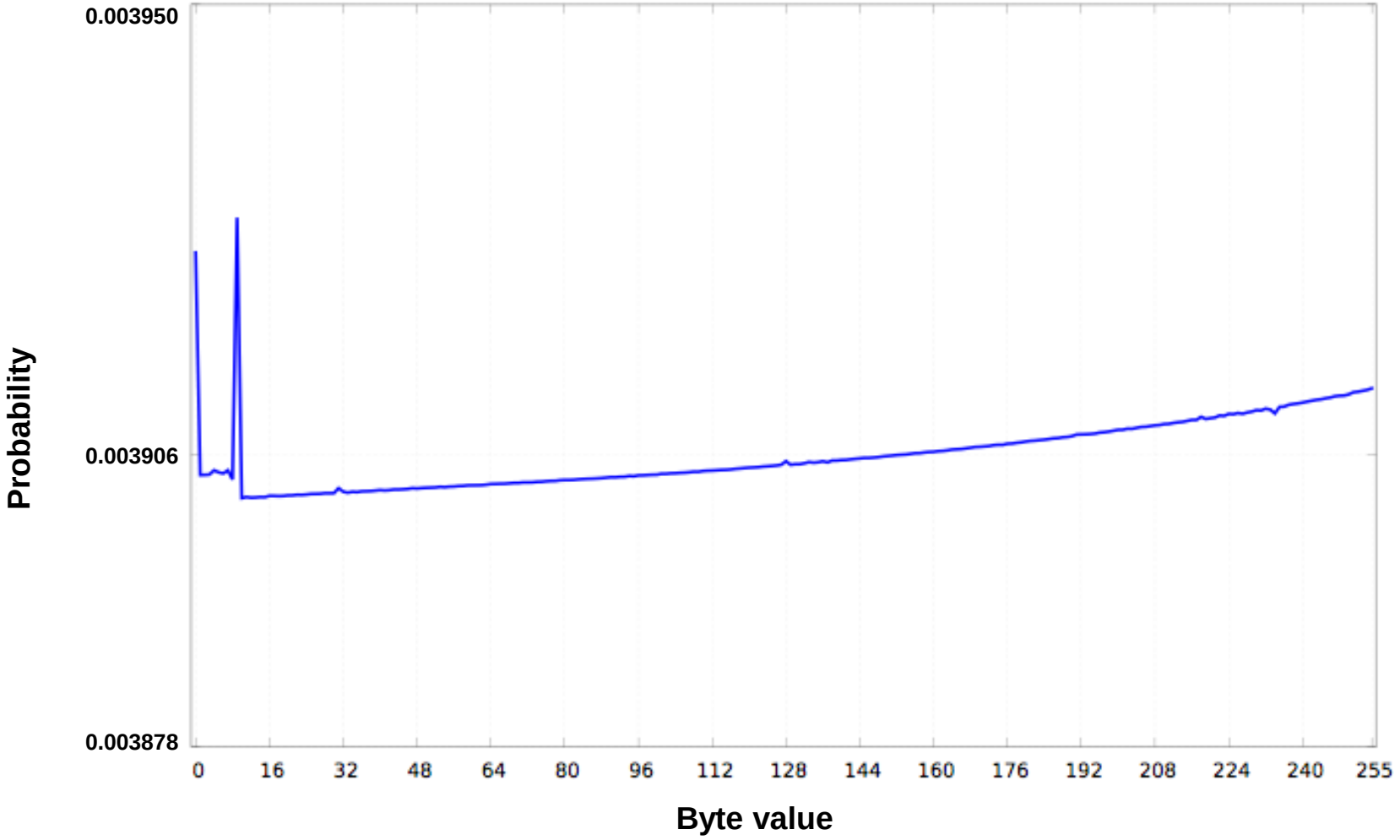
Keystream Distribution at Position 7



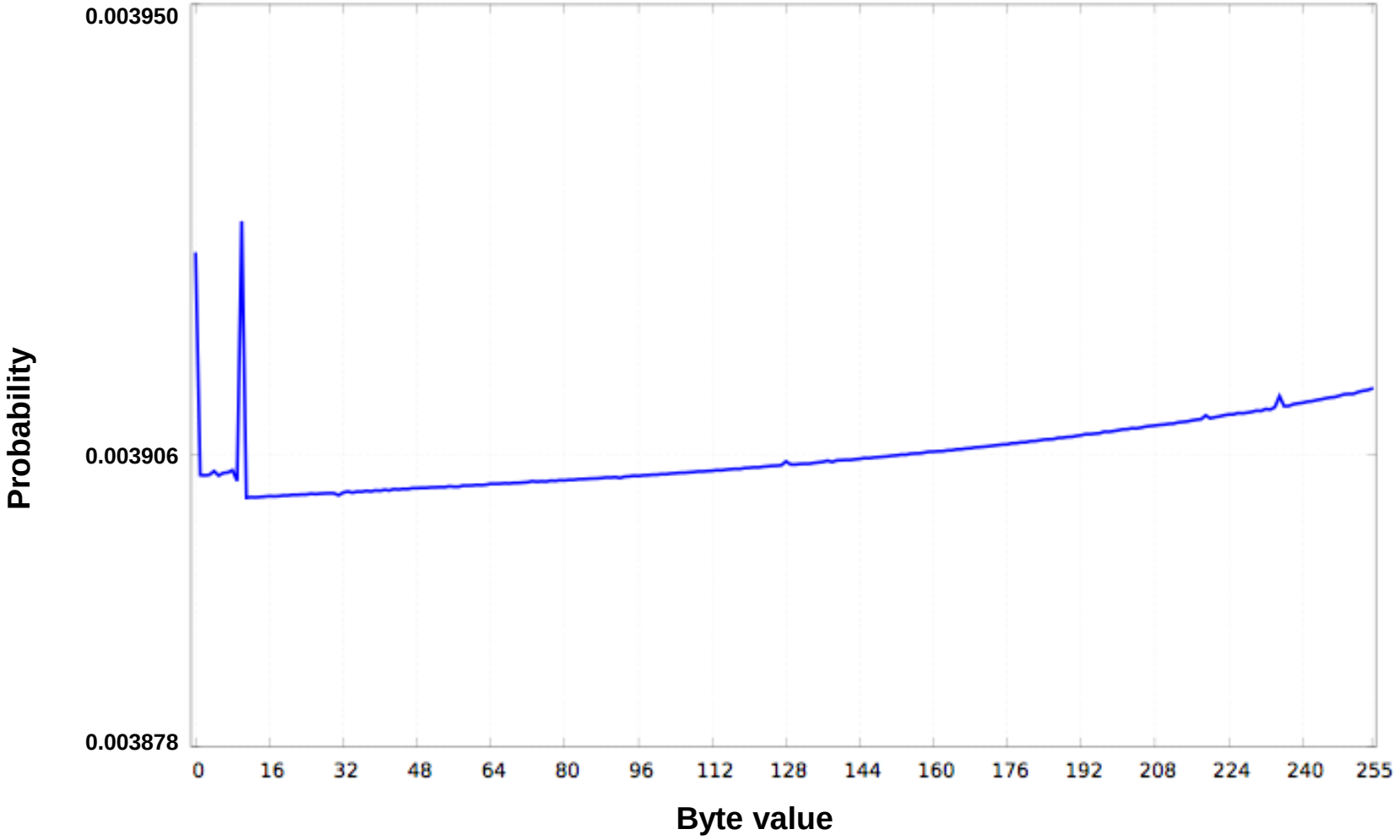
Keystream Distribution at Position 8



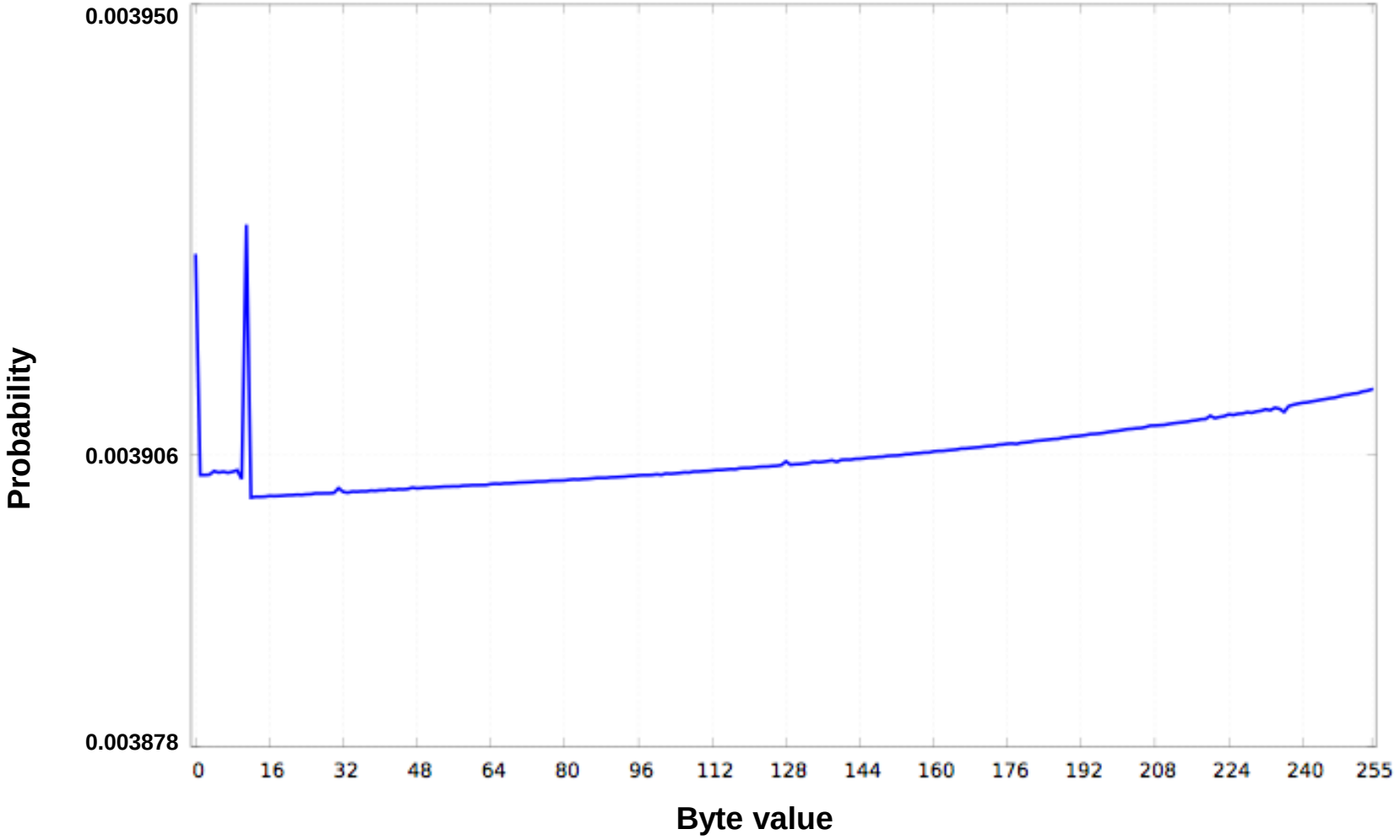
Keystream Distribution at Position 9



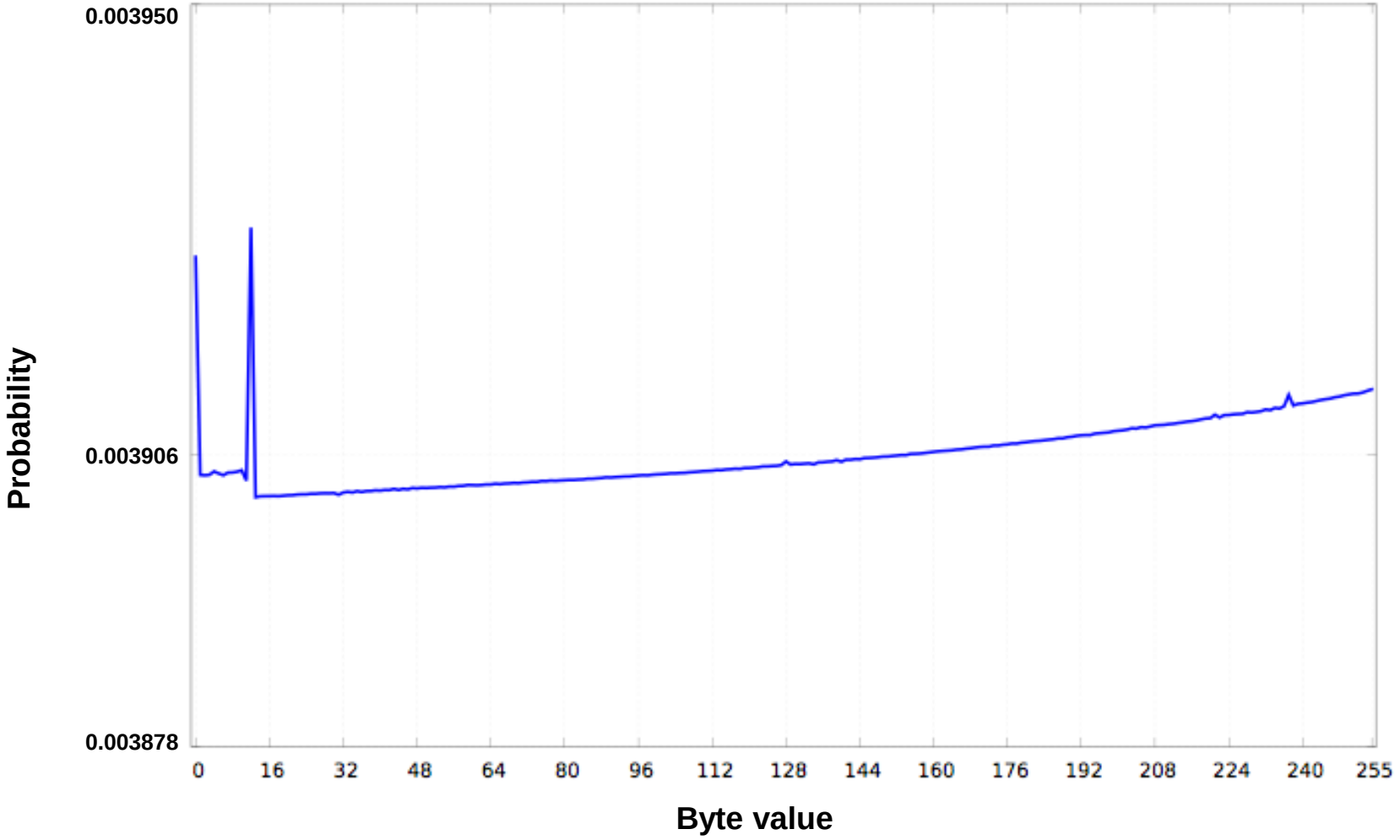
Keystream Distribution at Position 10



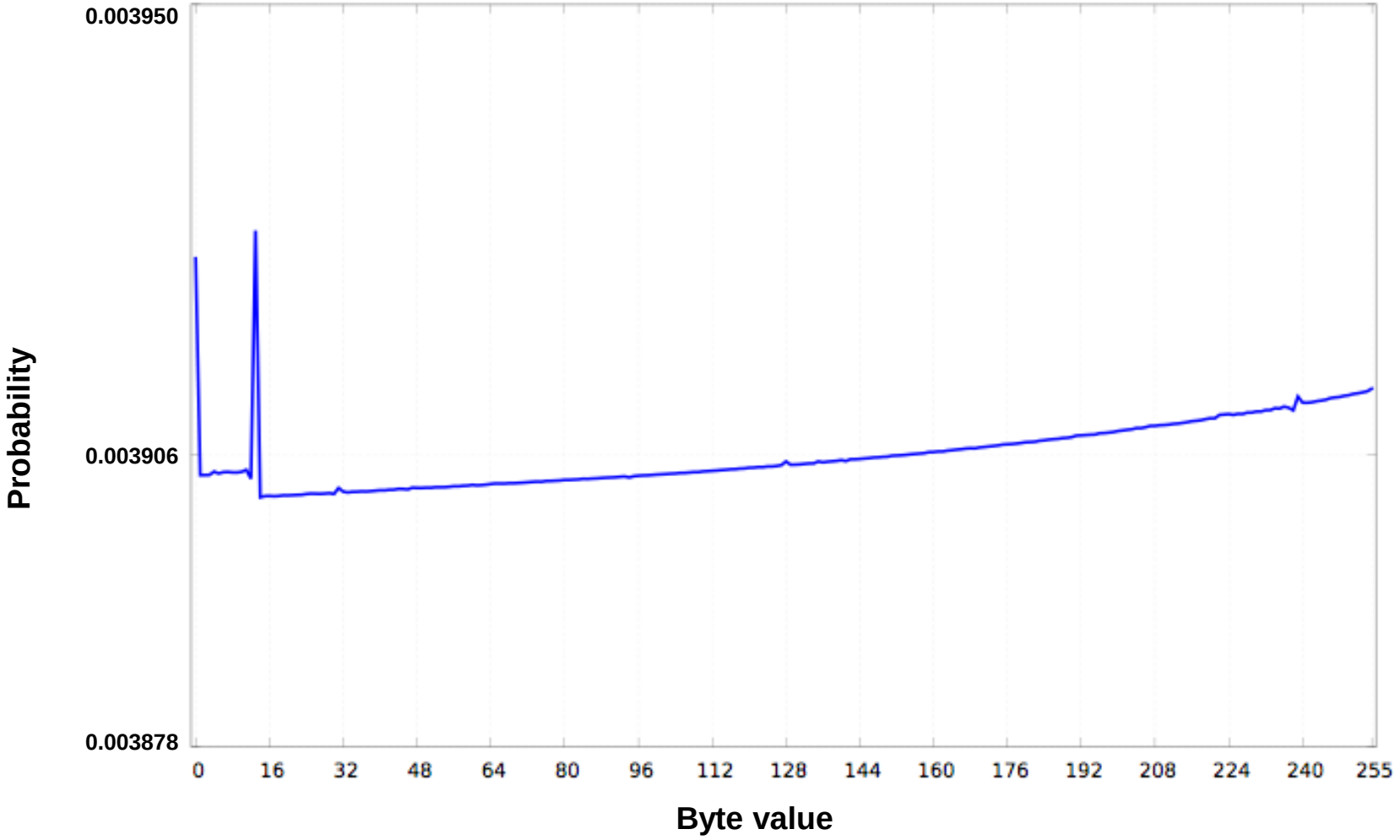
Keystream Distribution at Position 11



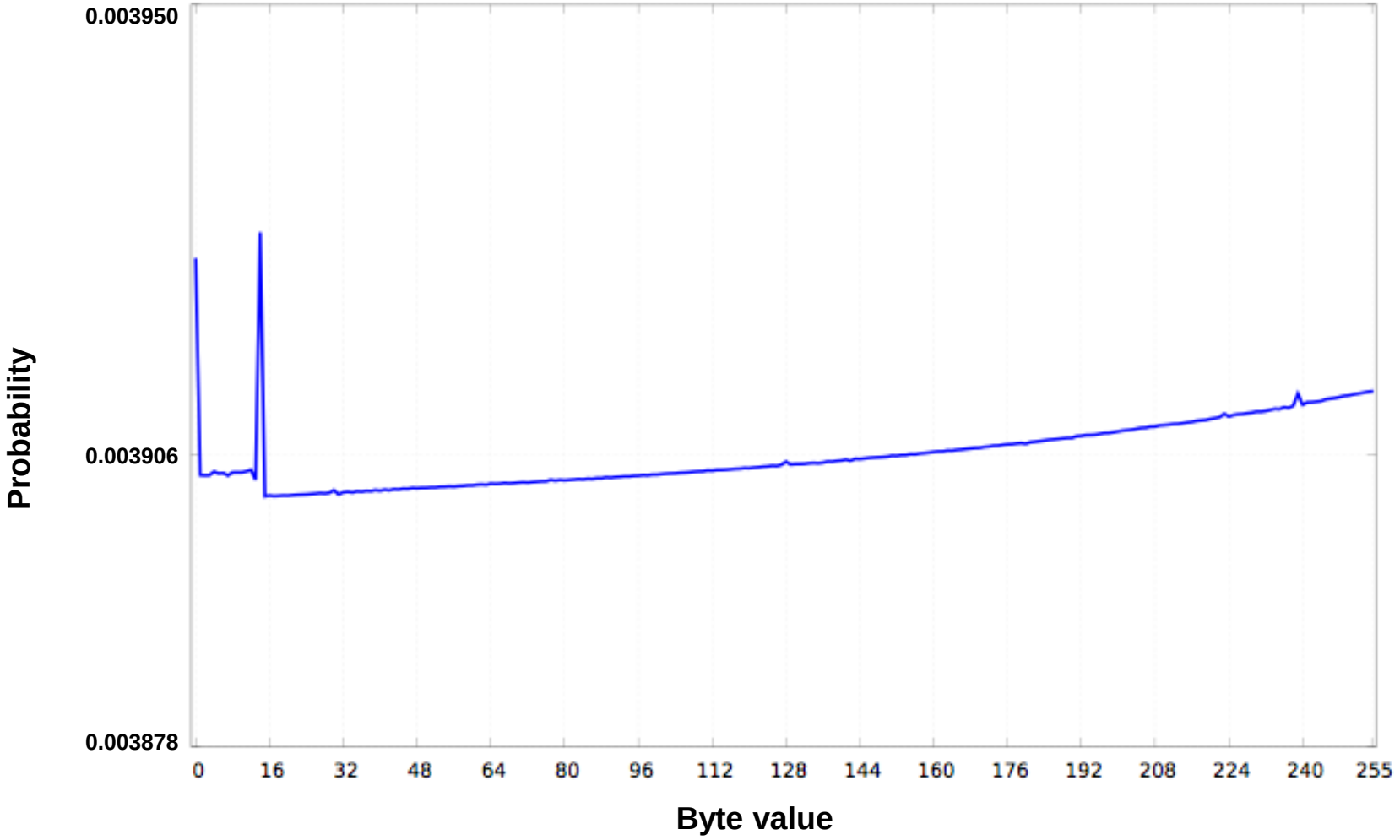
Keystream Distribution at Position 12



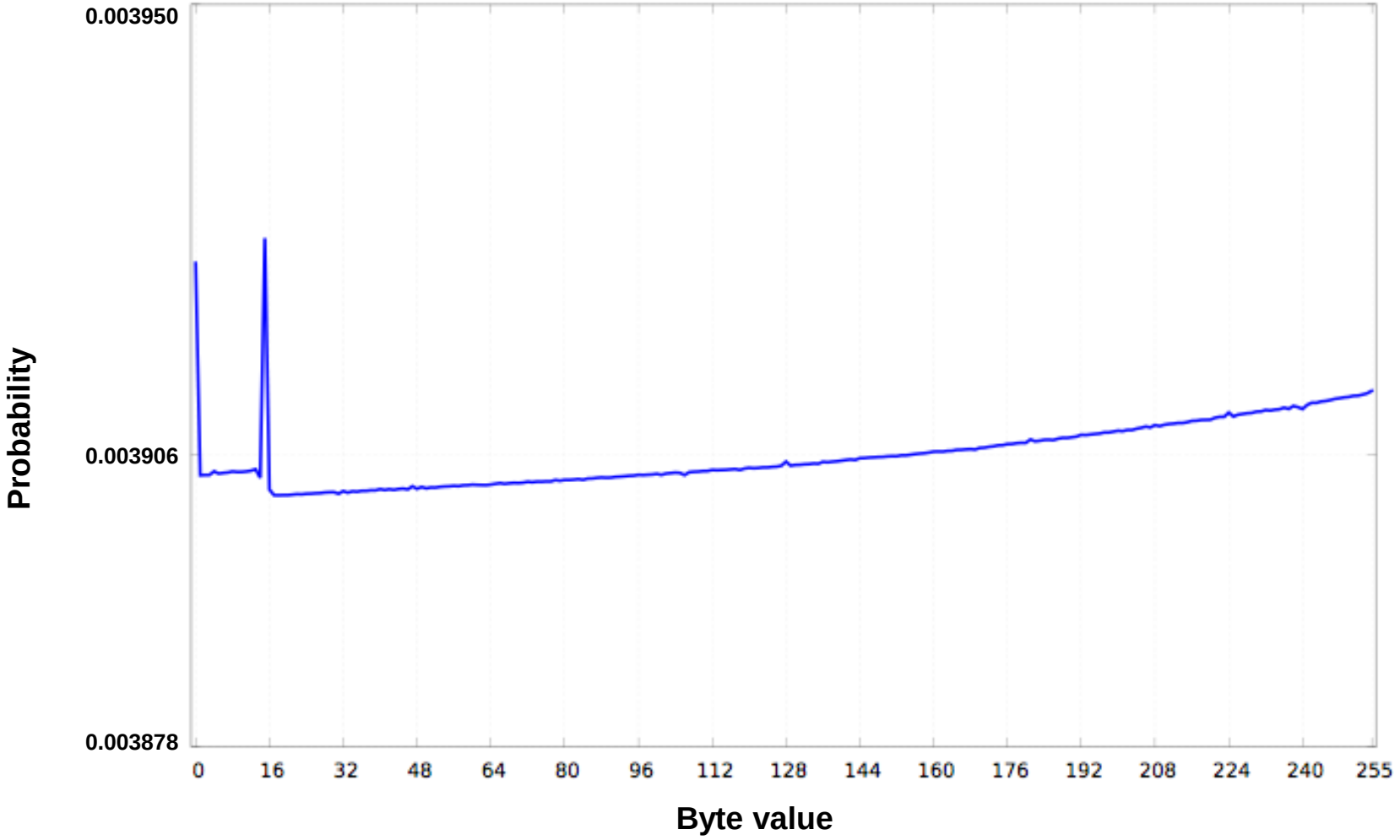
Keystream Distribution at Position 13



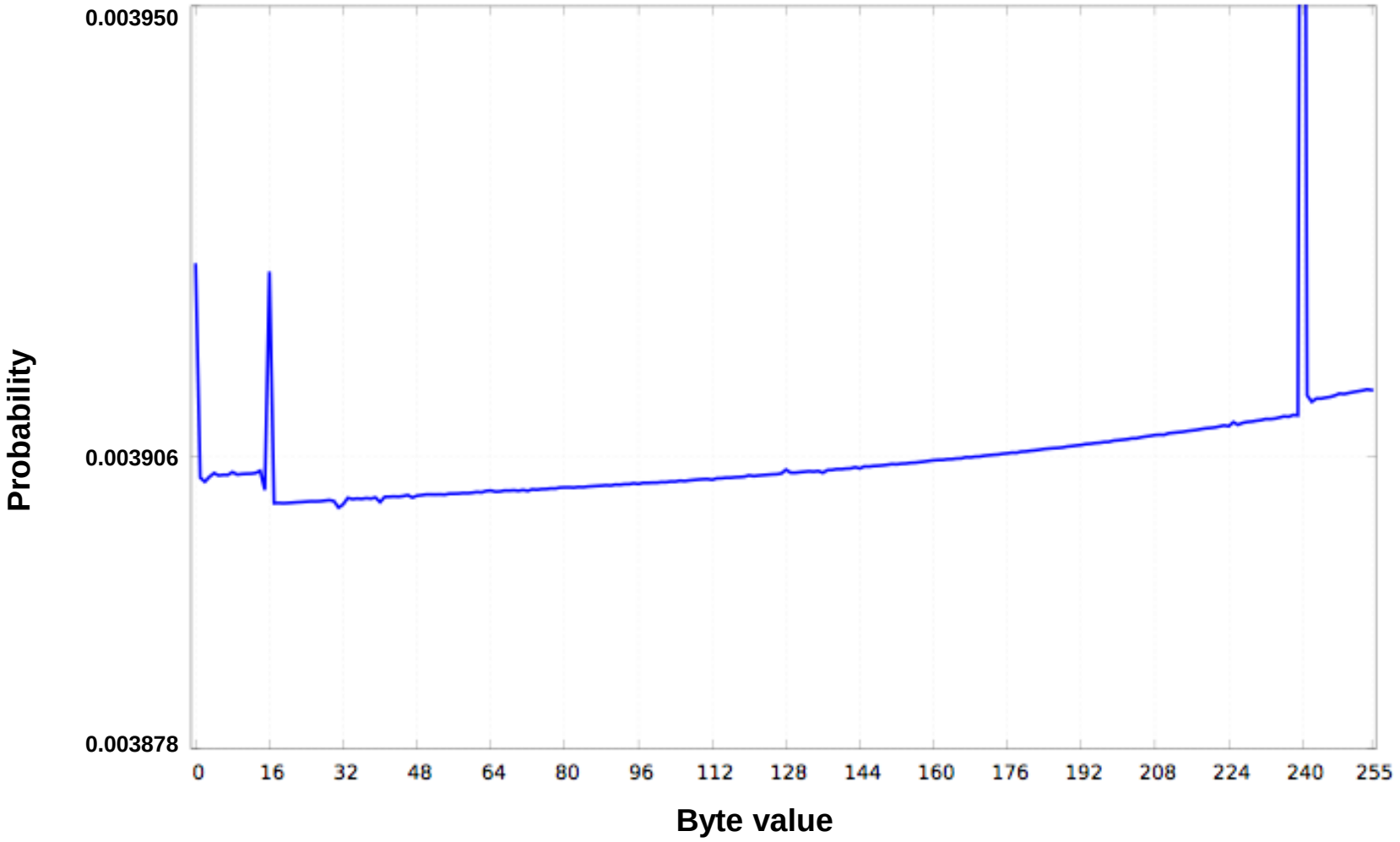
Keystream Distribution at Position 14



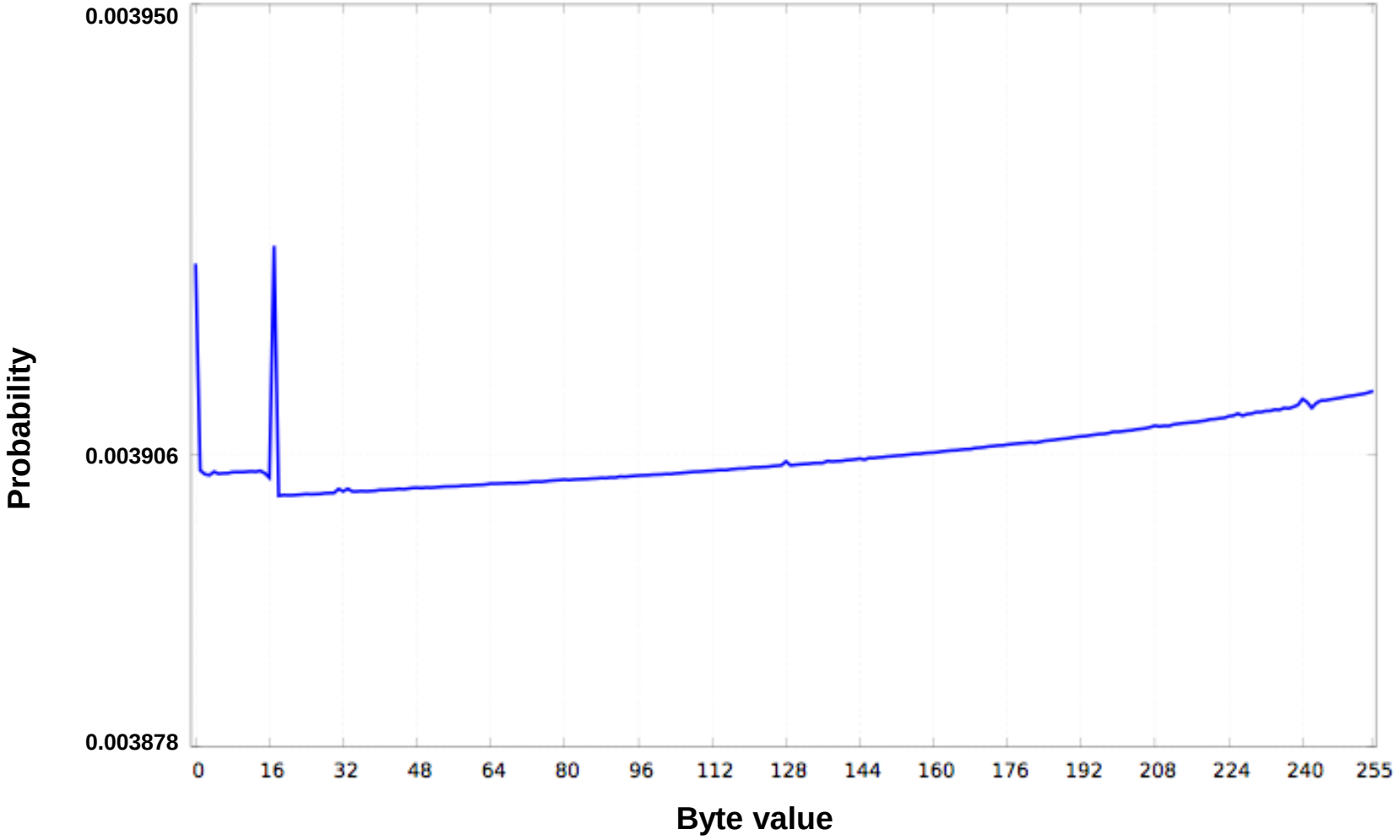
Keystream Distribution at Position 15



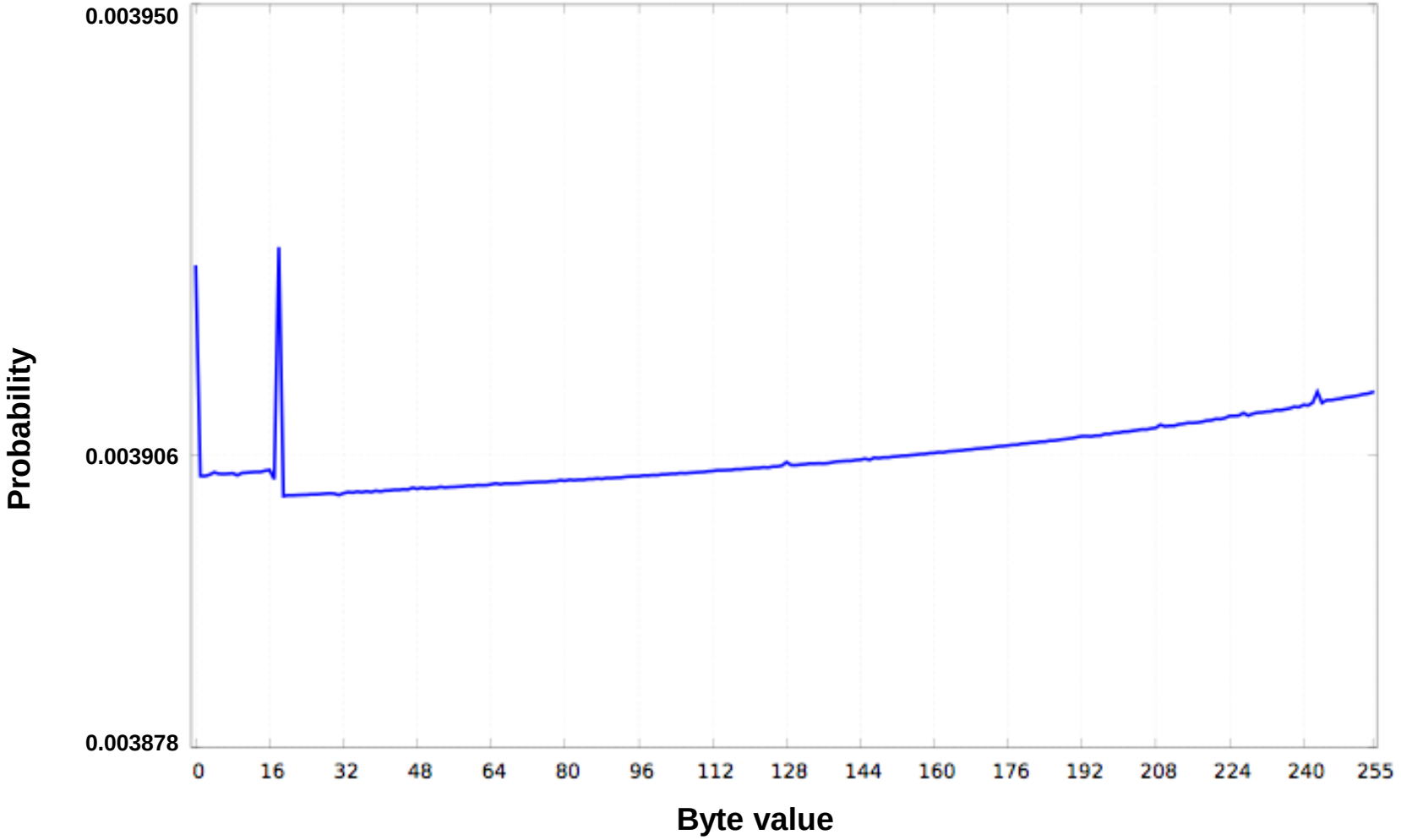
Keystream Distribution at Position 16



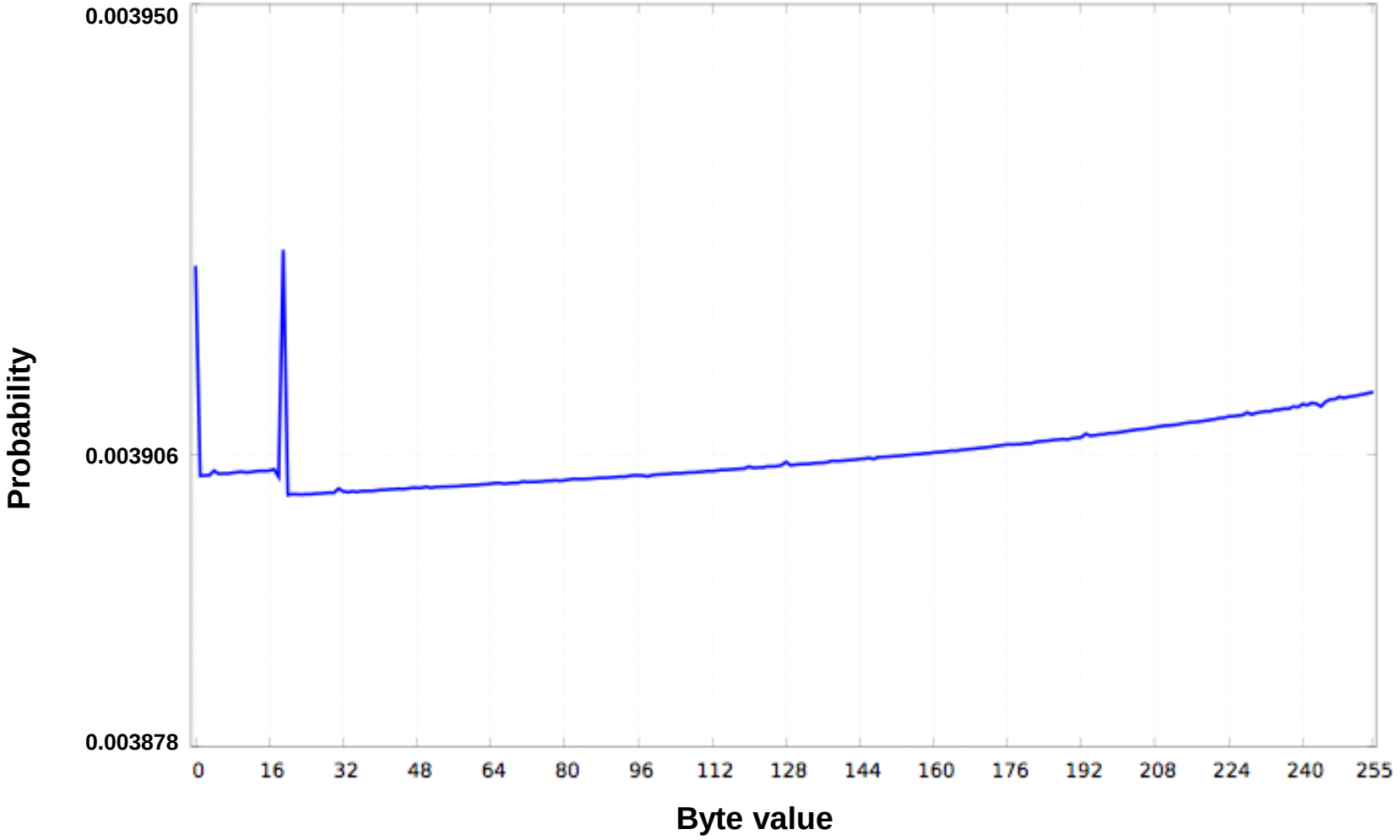
Keystream Distribution at Position 17



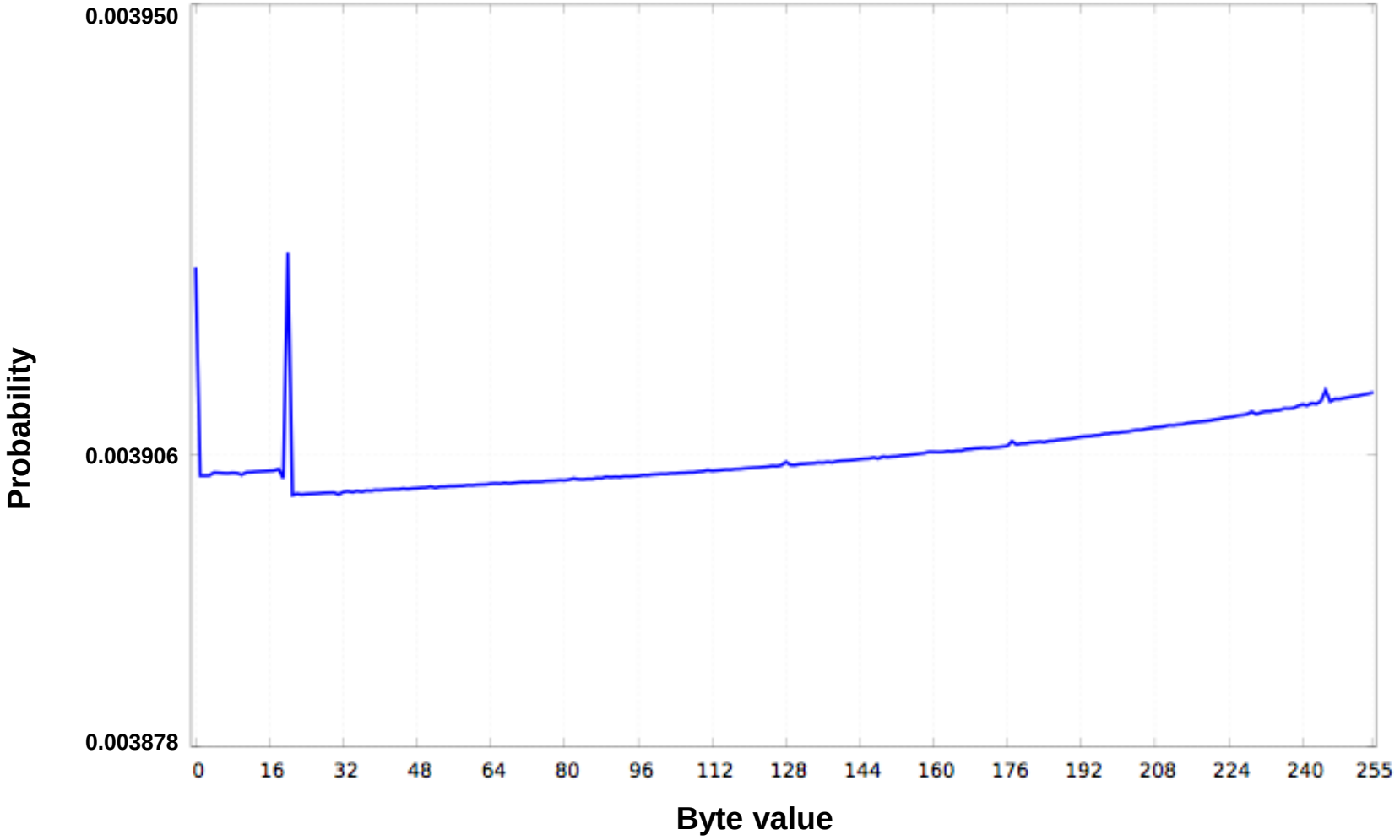
Keystream Distribution at Position 18



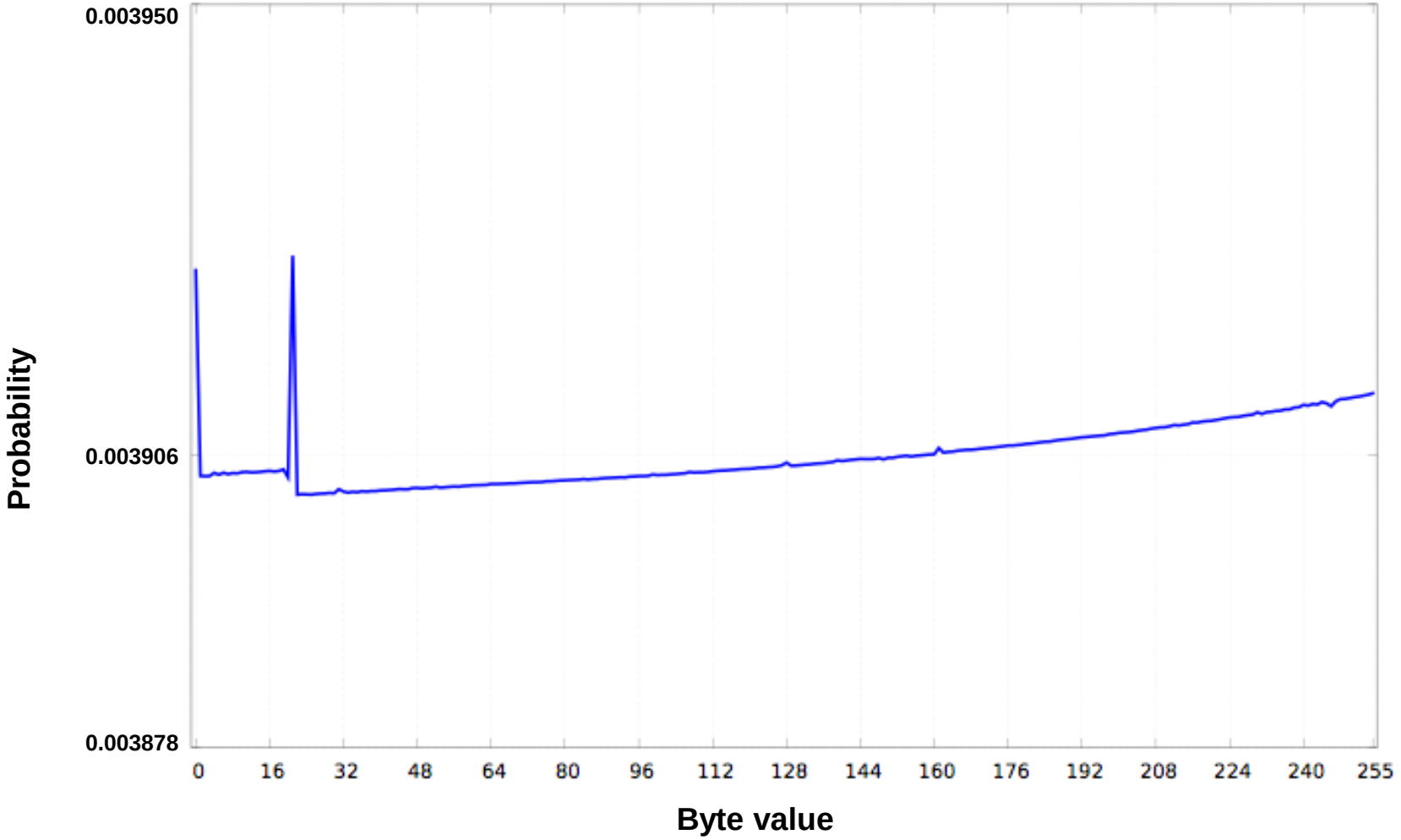
Keystream Distribution at Position 19



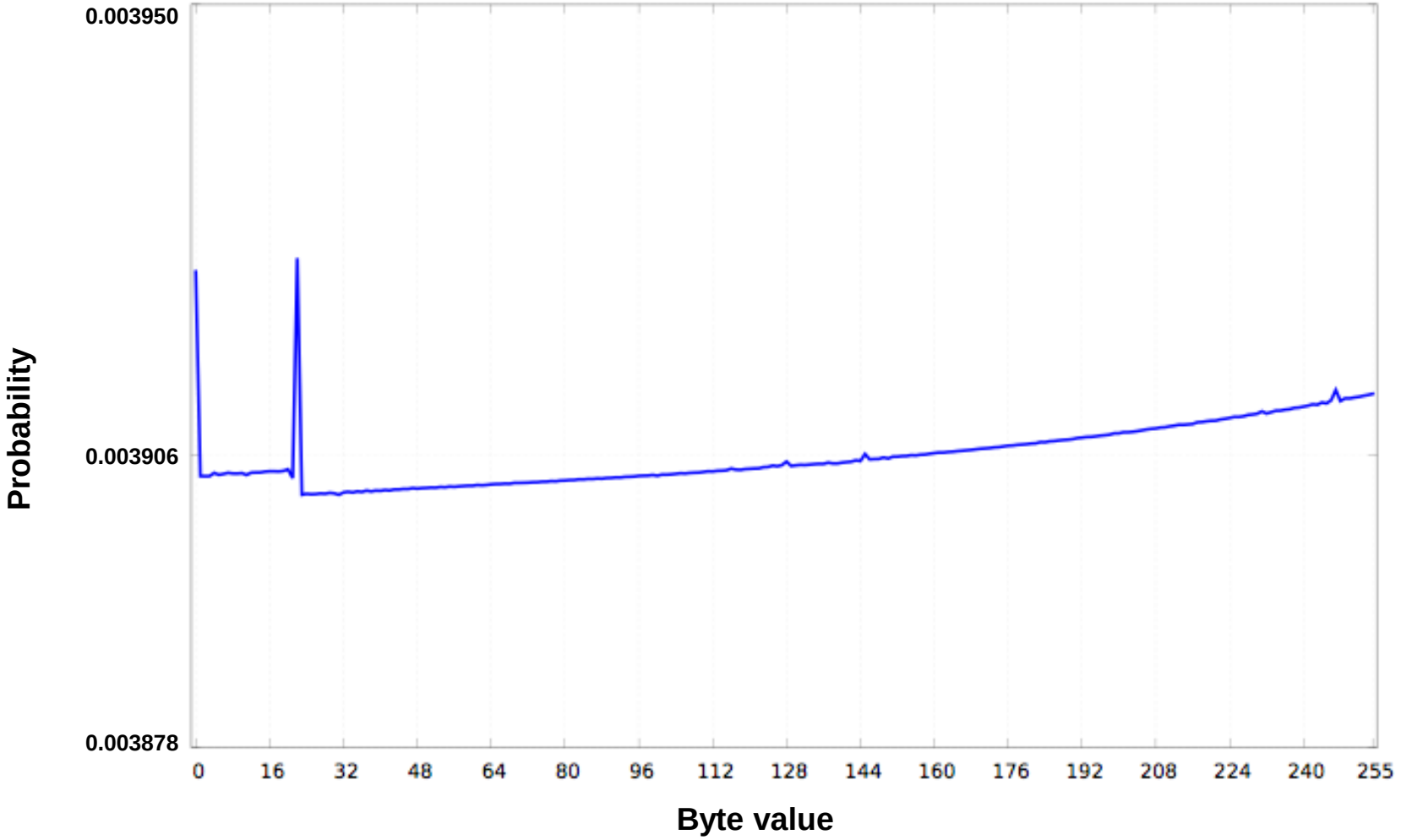
Keystream Distribution at Position 20



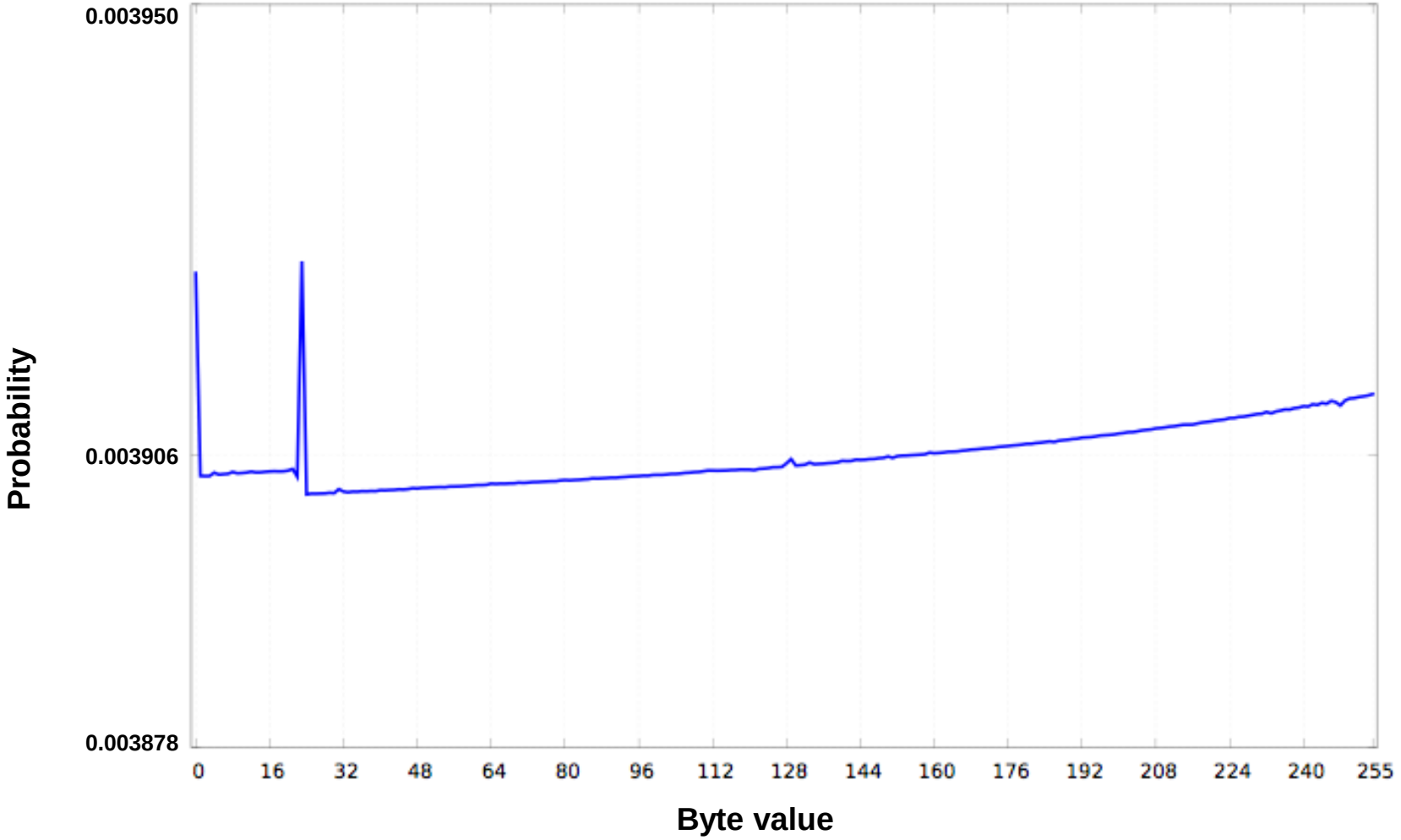
Keystream Distribution at Position 21



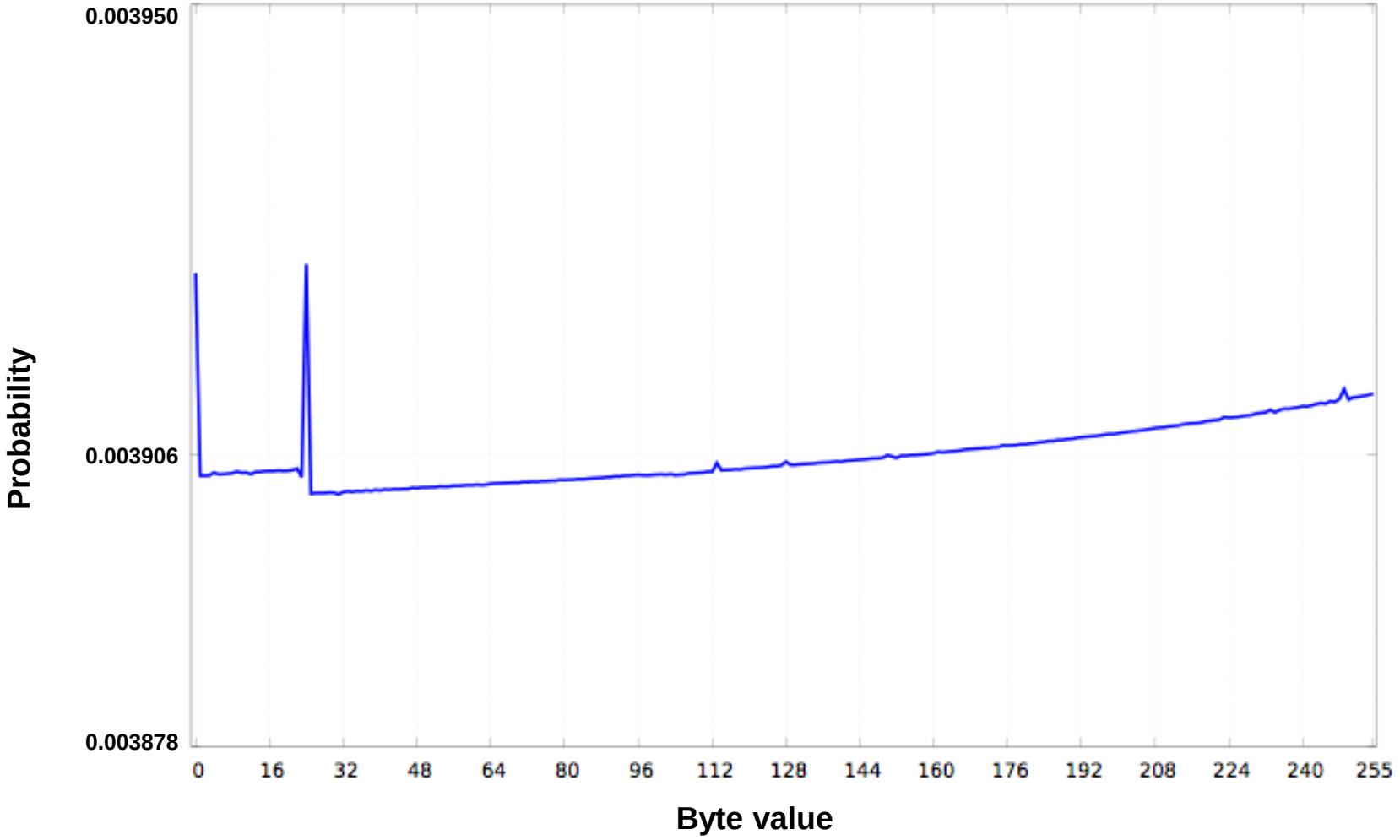
Keystream Distribution at Position 22



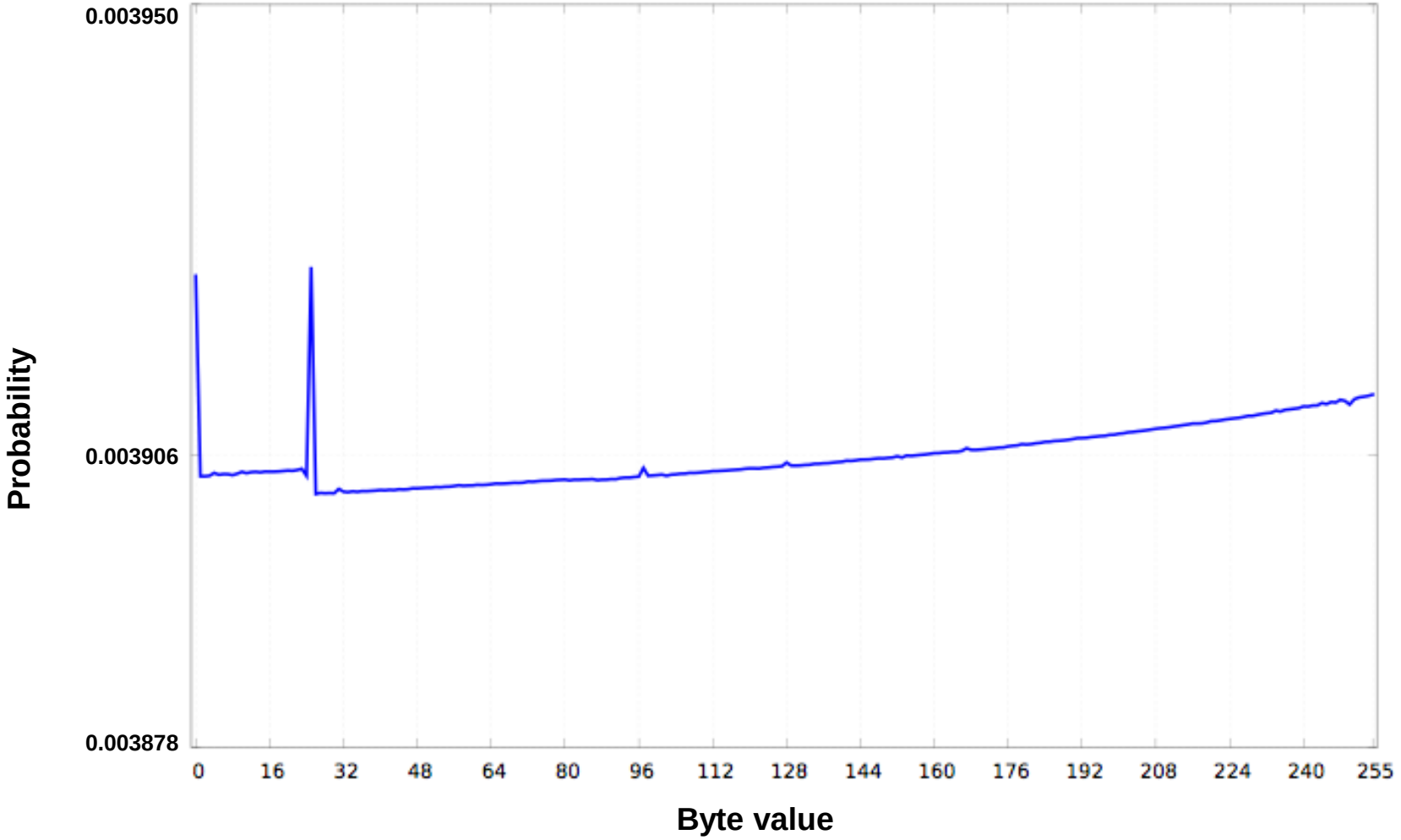
Keystream Distribution at Position 23



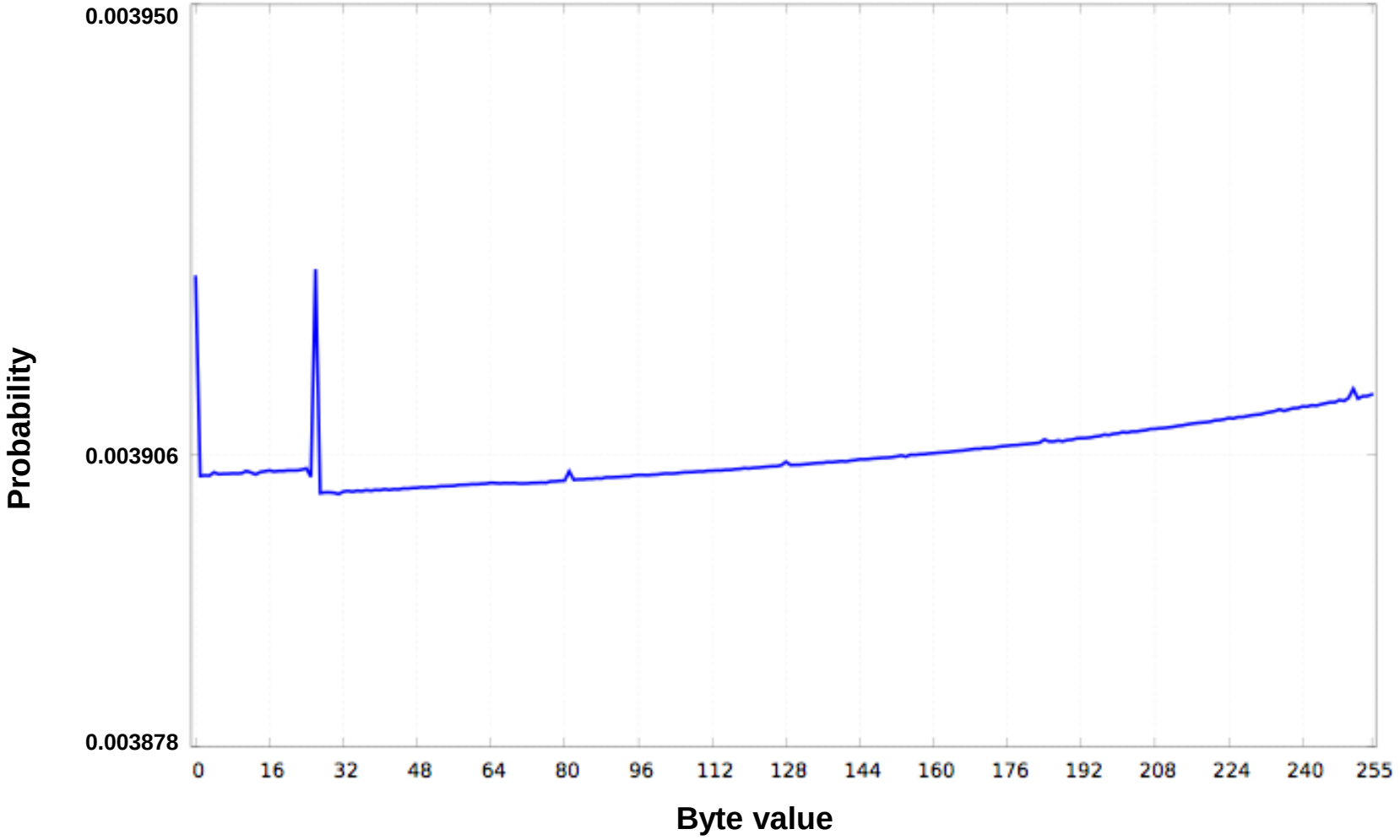
Keystream Distribution at Position 24



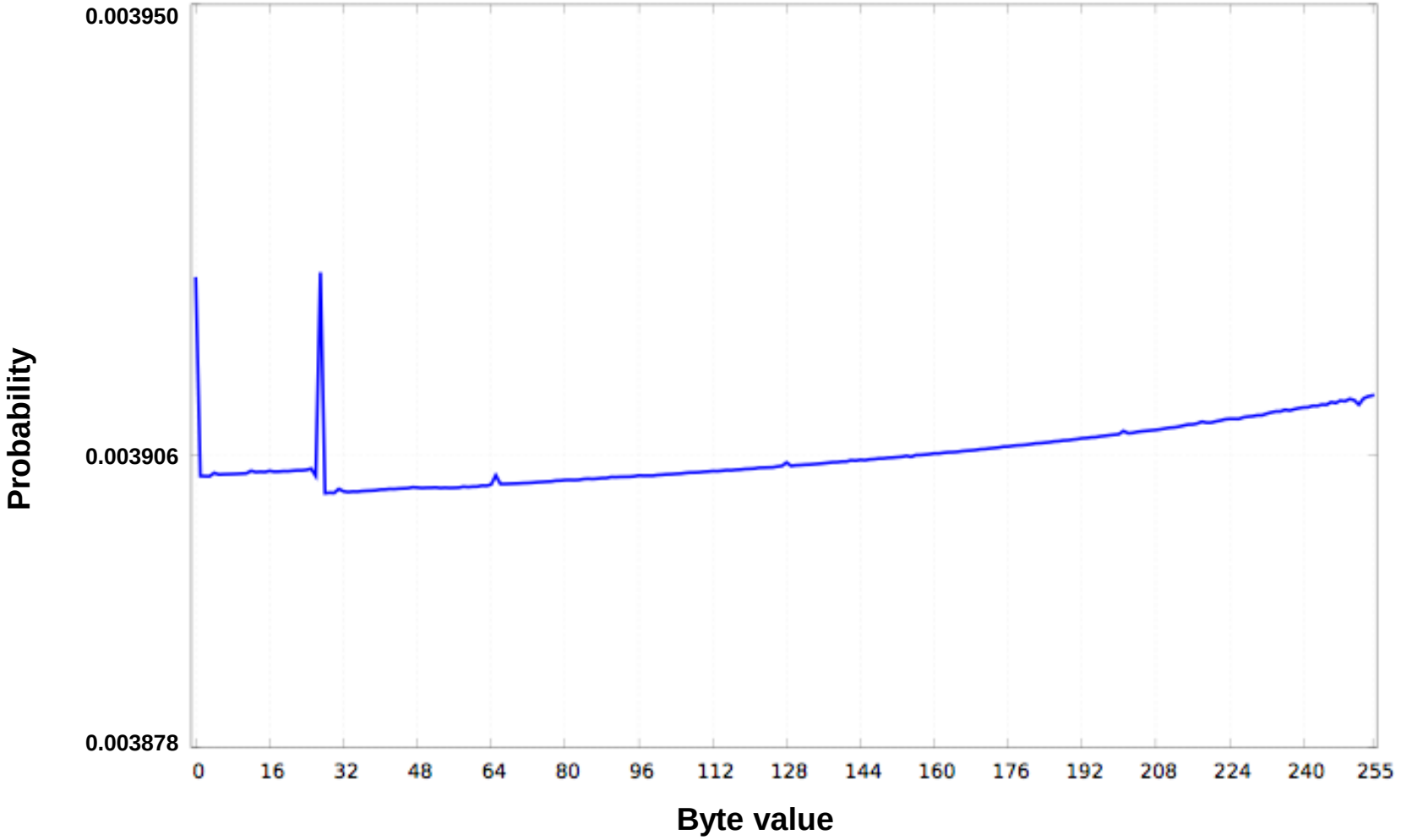
Keystream Distribution at Position 25



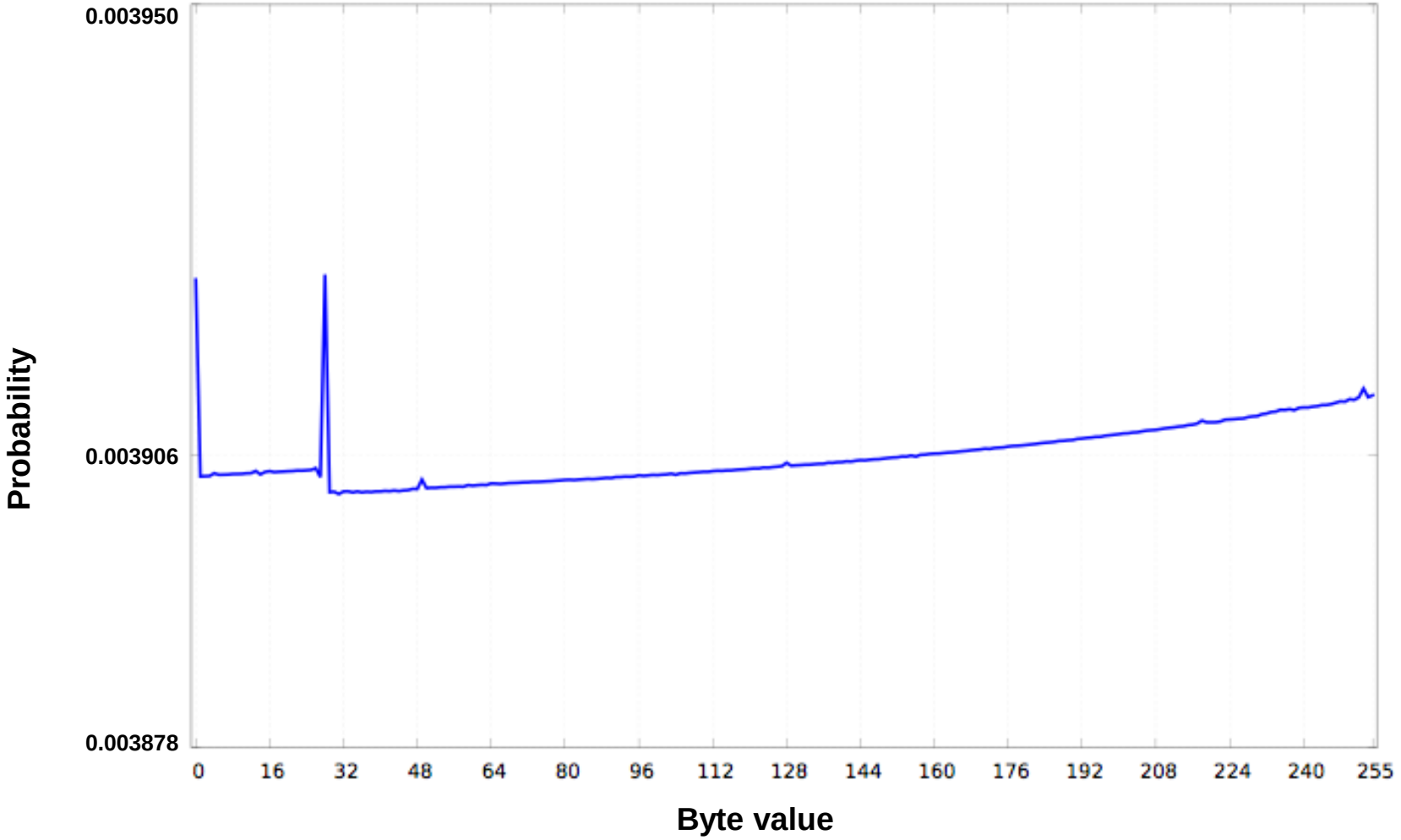
Keystream Distribution at Position 26



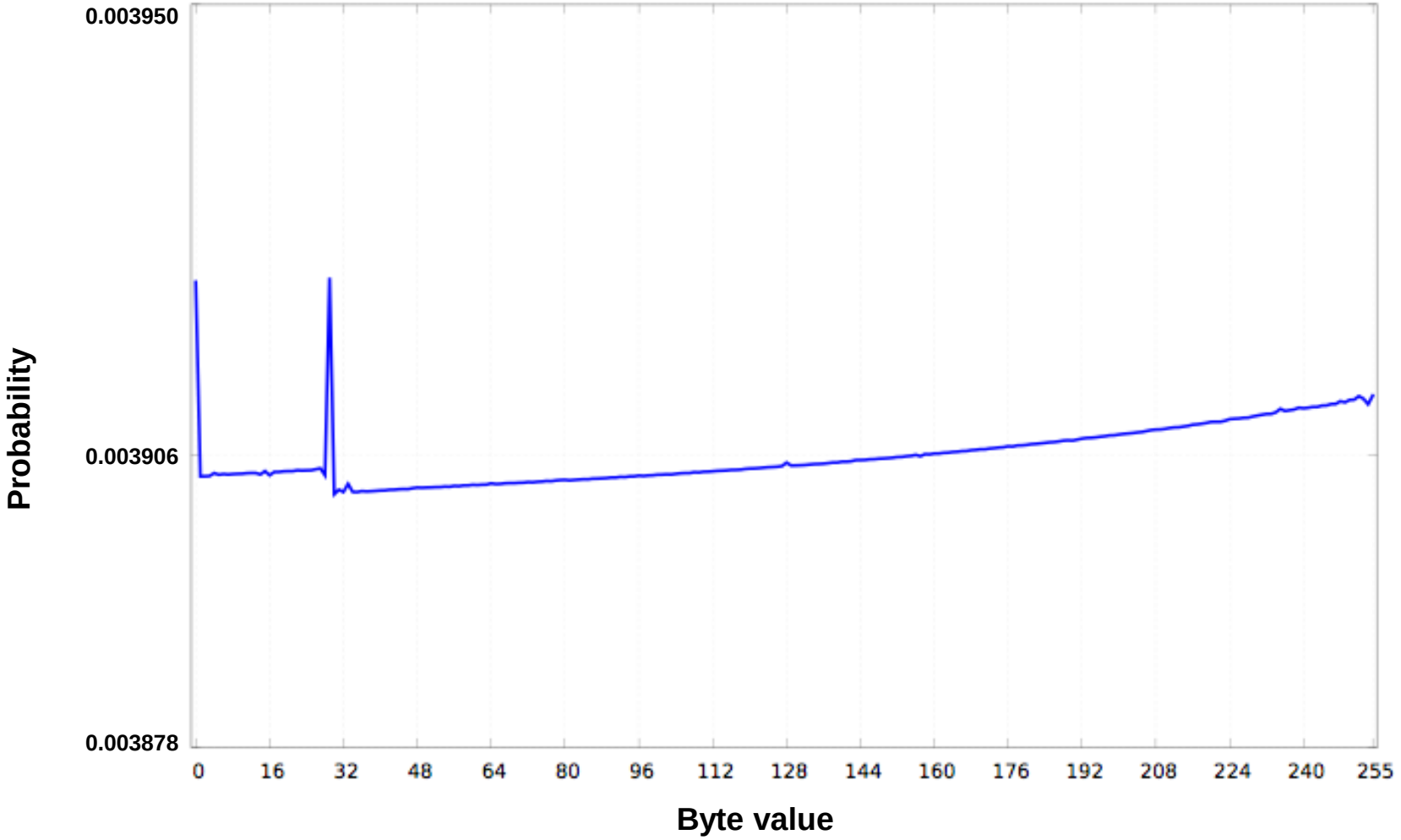
Keystream Distribution at Position 27



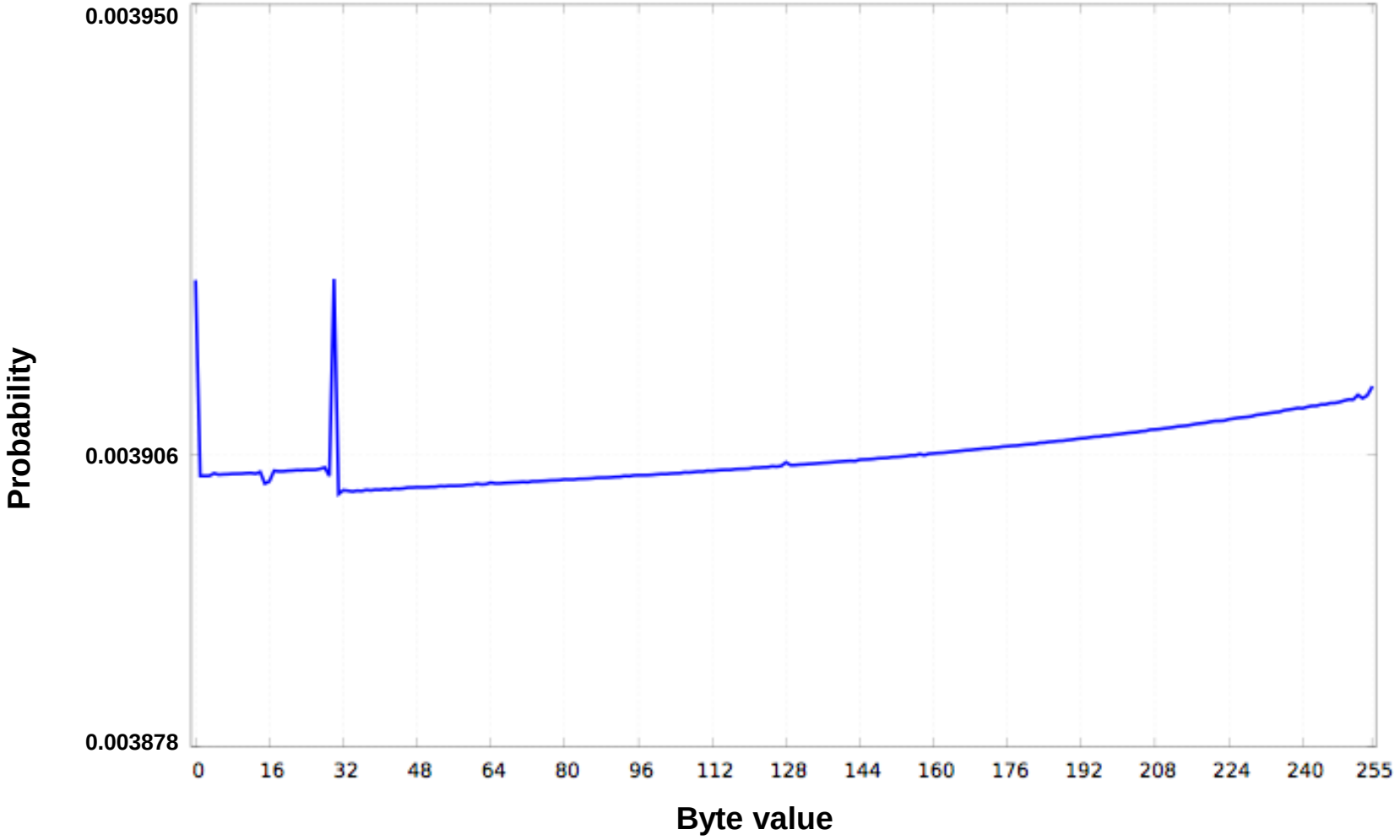
Keystream Distribution at Position 28



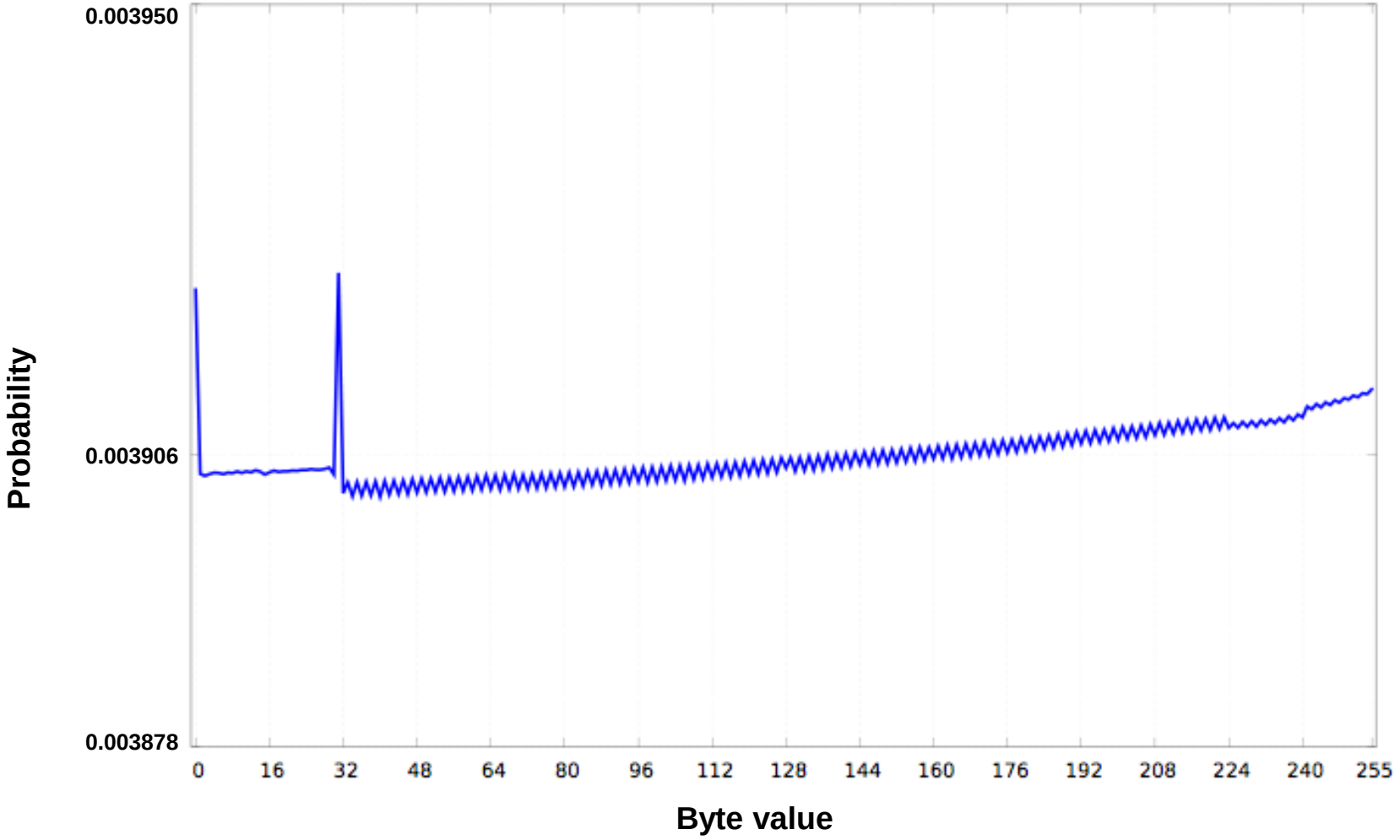
Keystream Distribution at Position 29



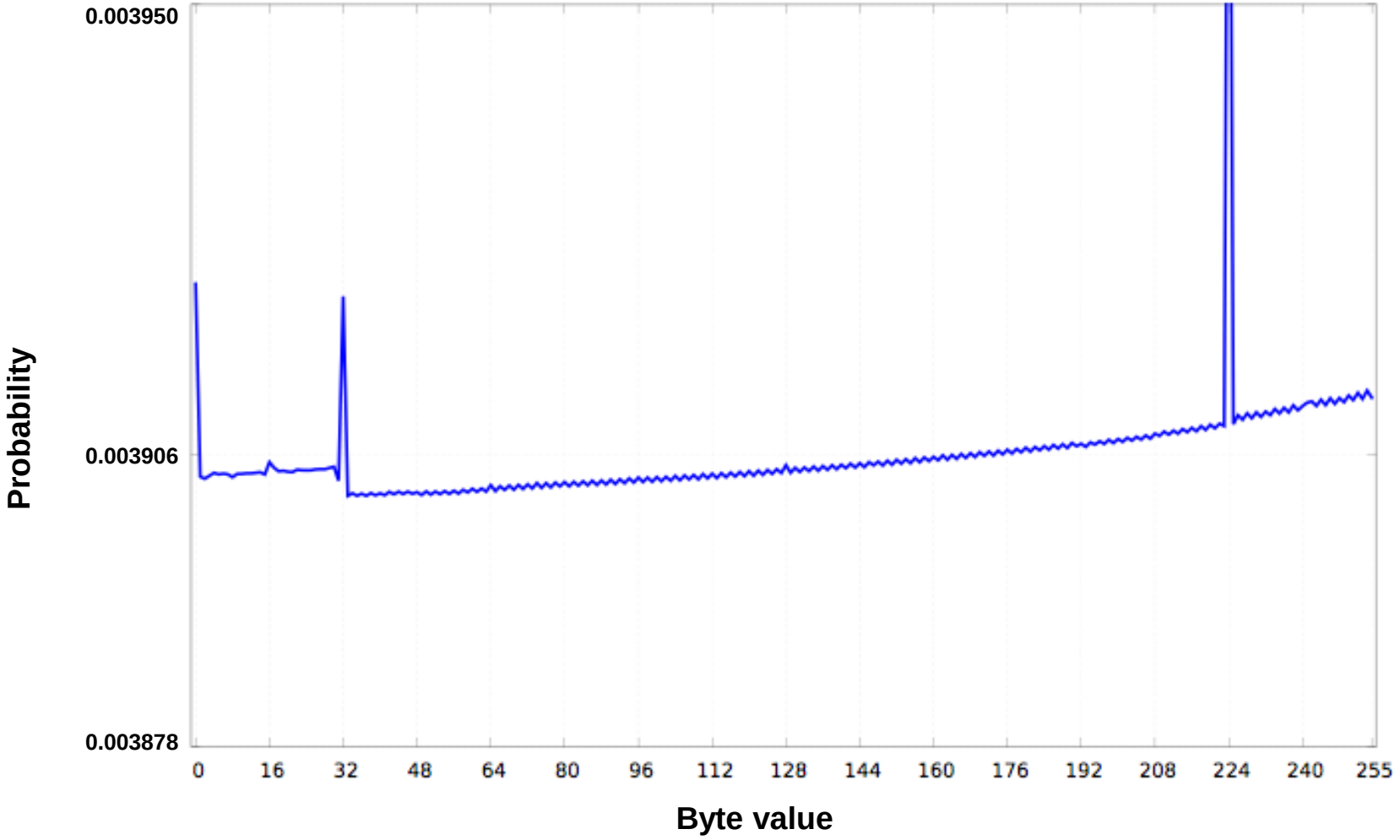
Keystream Distribution at Position 30



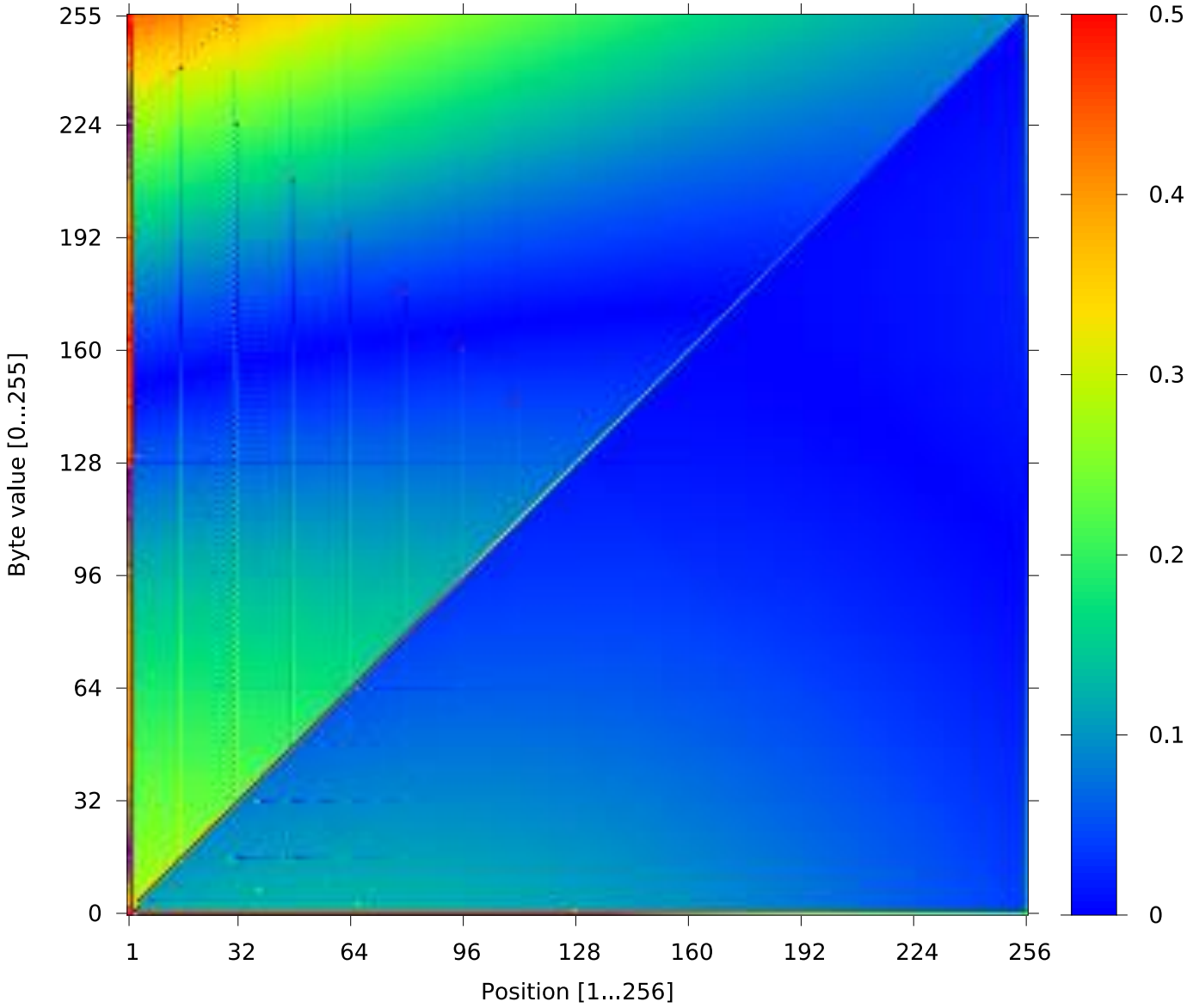
Keystream Distribution at Position 31



Keystream Distribution at Position 32



All the Biases



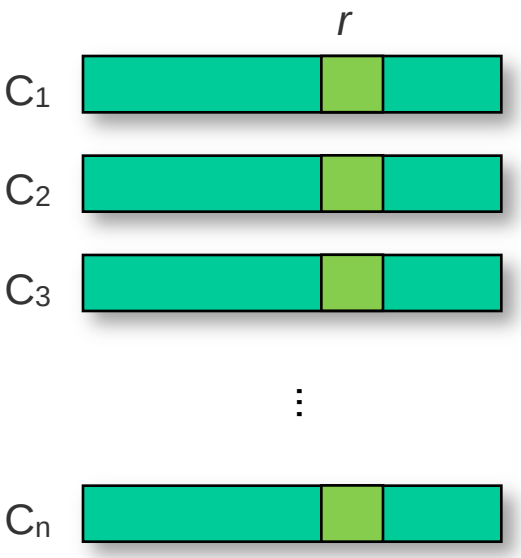
Plaintext Recovery for TLS-RC4

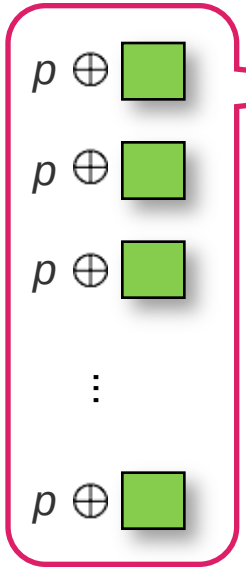
- So what?
- Using the biased keystream byte distributions, we can construct a plaintext recovery attack against TLS.
- The attack requires the same plaintext to be encrypted under *many* different keys.
  - Use Javascript in browser as mechanism, cookies as target, as in BEAST attack.
  - There is a meaningful attack scenario!

Plaintext recovery using keystream biases

Encryptions of fixed plaintext under different keys

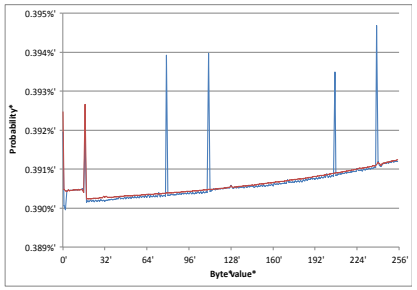
Plaintext candidate byte *p*





yields induced distribution on keystream byte *Z*

combine with known distribution



Likelihood of *p* being correct plaintext byte

Recovery algorithm:
Compute most likely plaintext byte

Details of Statistical Analysis

Let $\boldsymbol{c}$ be the n-vector of ciphertext bytes in position r.

Let $\boldsymbol{q} = (q_{00}, q_{01},\ldots, q_{ff})$ be the vector of keystream byte probabilities in position r.

Bayes theorem:

$$\Pr[P=p \mid \boldsymbol{C}=\boldsymbol{c}] = \Pr[\boldsymbol{C}=\boldsymbol{c} \mid P=p].\ \Pr[P=p]/\Pr[\boldsymbol{C}=\boldsymbol{c}]$$
$$= \Pr[\boldsymbol{Z}=\boldsymbol{c} \oplus p \mid P=p].\Pr[P=p]/\Pr[\boldsymbol{C}=\boldsymbol{c}].$$

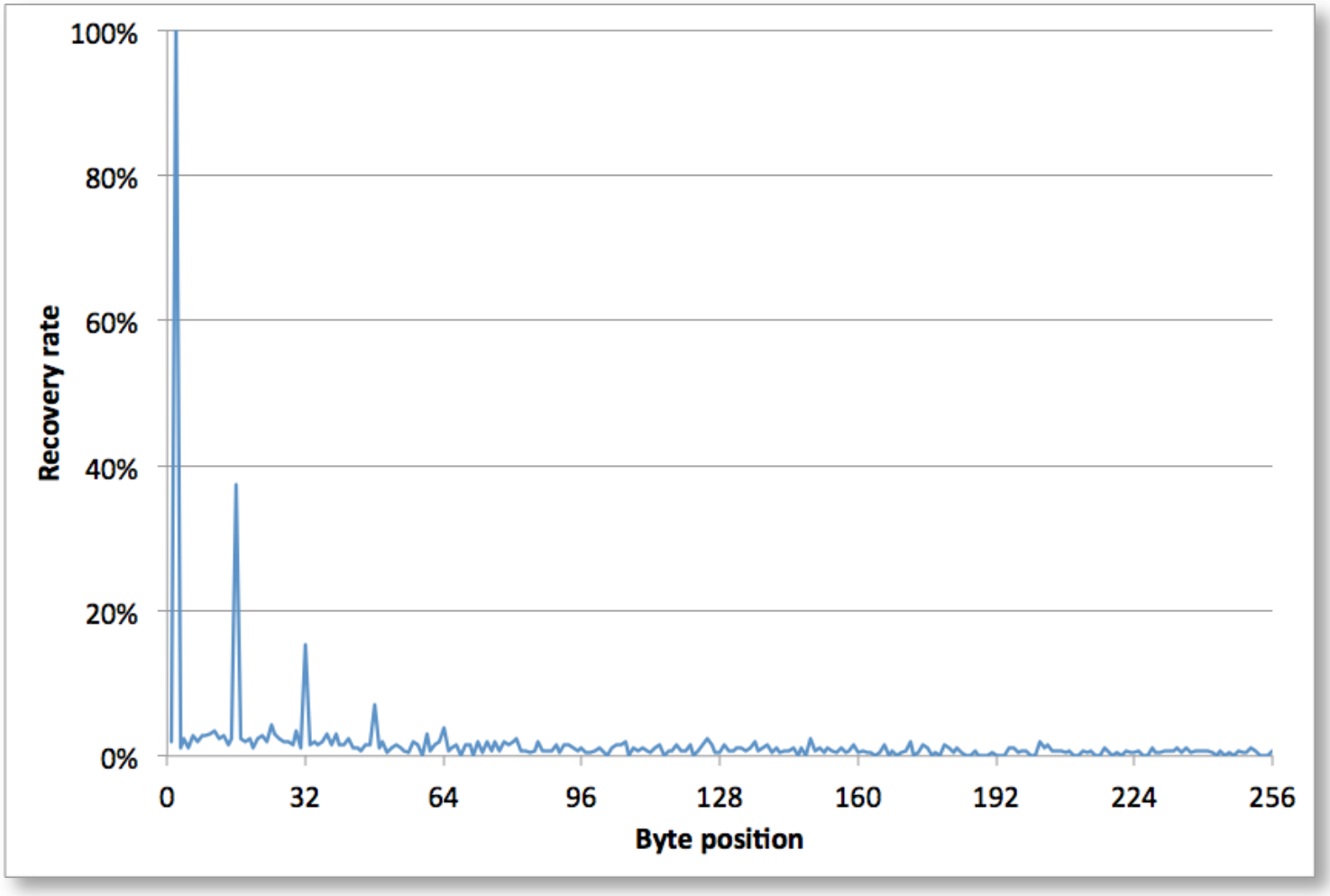
Assume $\Pr[P=p]$ is constant; $\Pr[\boldsymbol{C}=\boldsymbol{c}]$ is independent of the choice of p.

Then to maximise $\Pr[P=p \mid \boldsymbol{C}=\boldsymbol{c}]$ over all choices of p, we simply need to maximise:

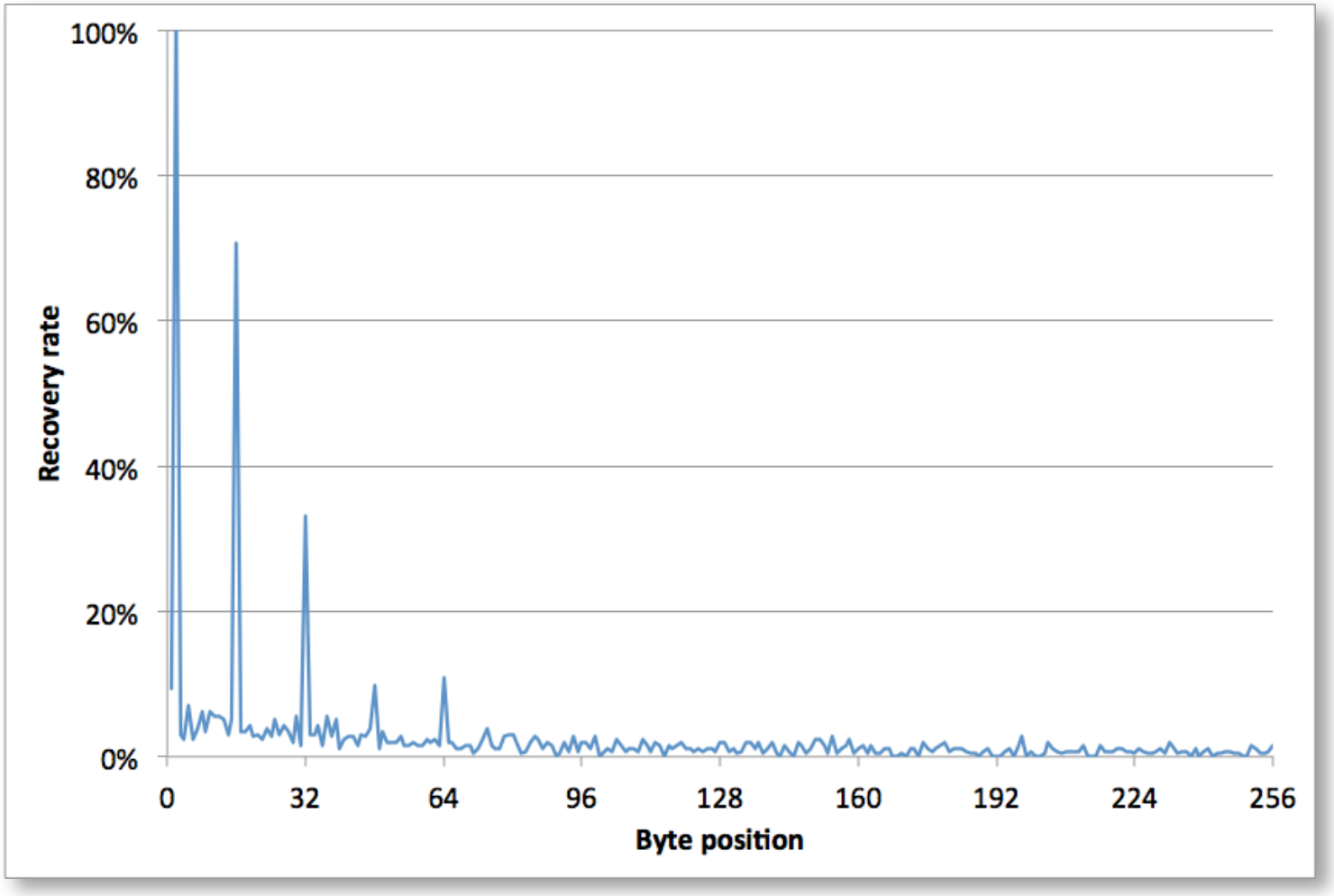
$$\Pr[\boldsymbol{Z}=\boldsymbol{c} \oplus p \mid P=p] = \frac{n!}{n_{00}!n_{01}!\ldots n_{ff}!} q_{00}^{n_{00}} q_{01}^{n_{01}} \ldots q_{ff}^{n_{ff}}$$

where n_x is the number of occurrences of byte value x in $\boldsymbol{Z}=\boldsymbol{c} \oplus p$ (which equals the number of occurrences of $x \oplus p$ in $\boldsymbol{c}$).

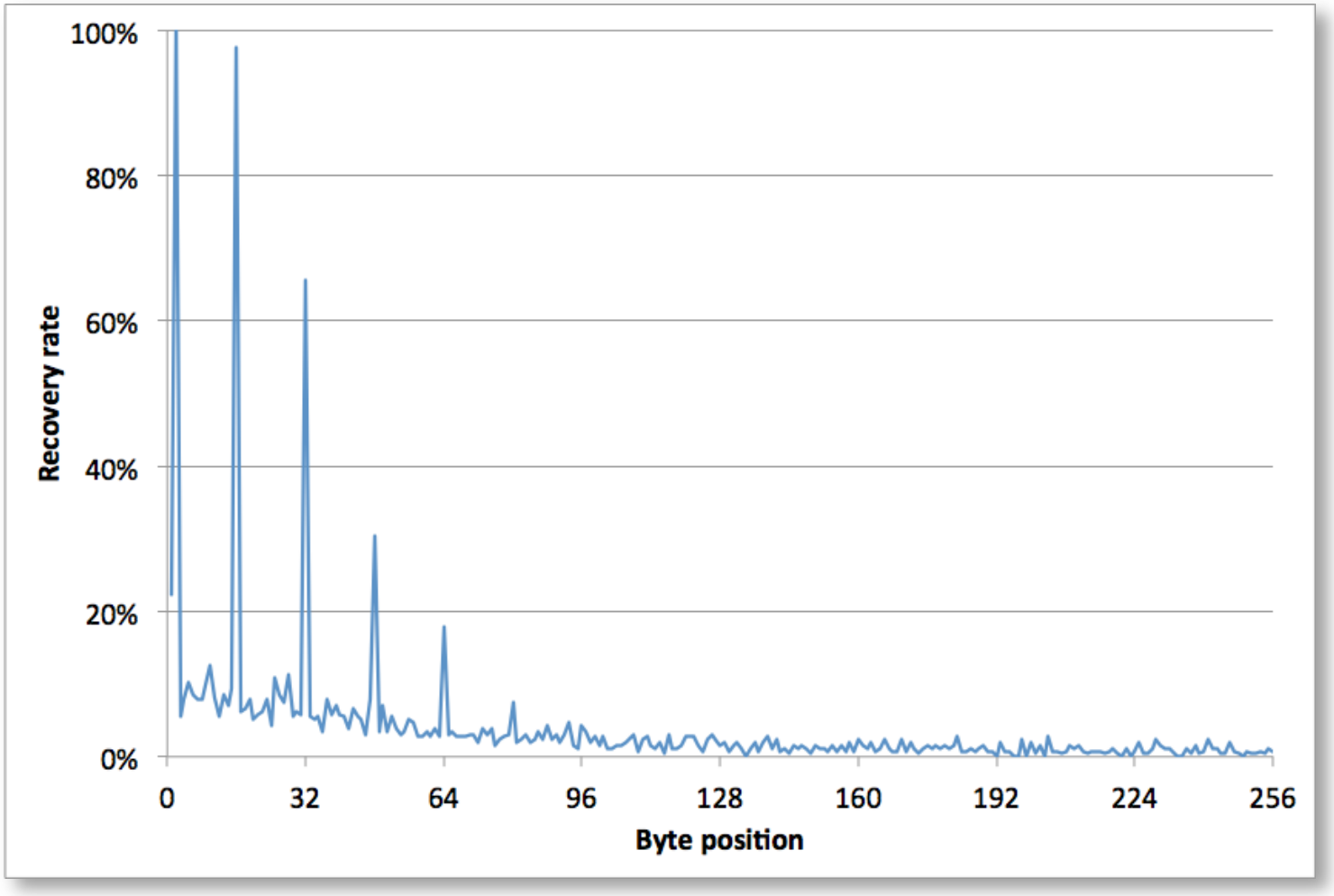
Success Probability
2^{20} Sessions



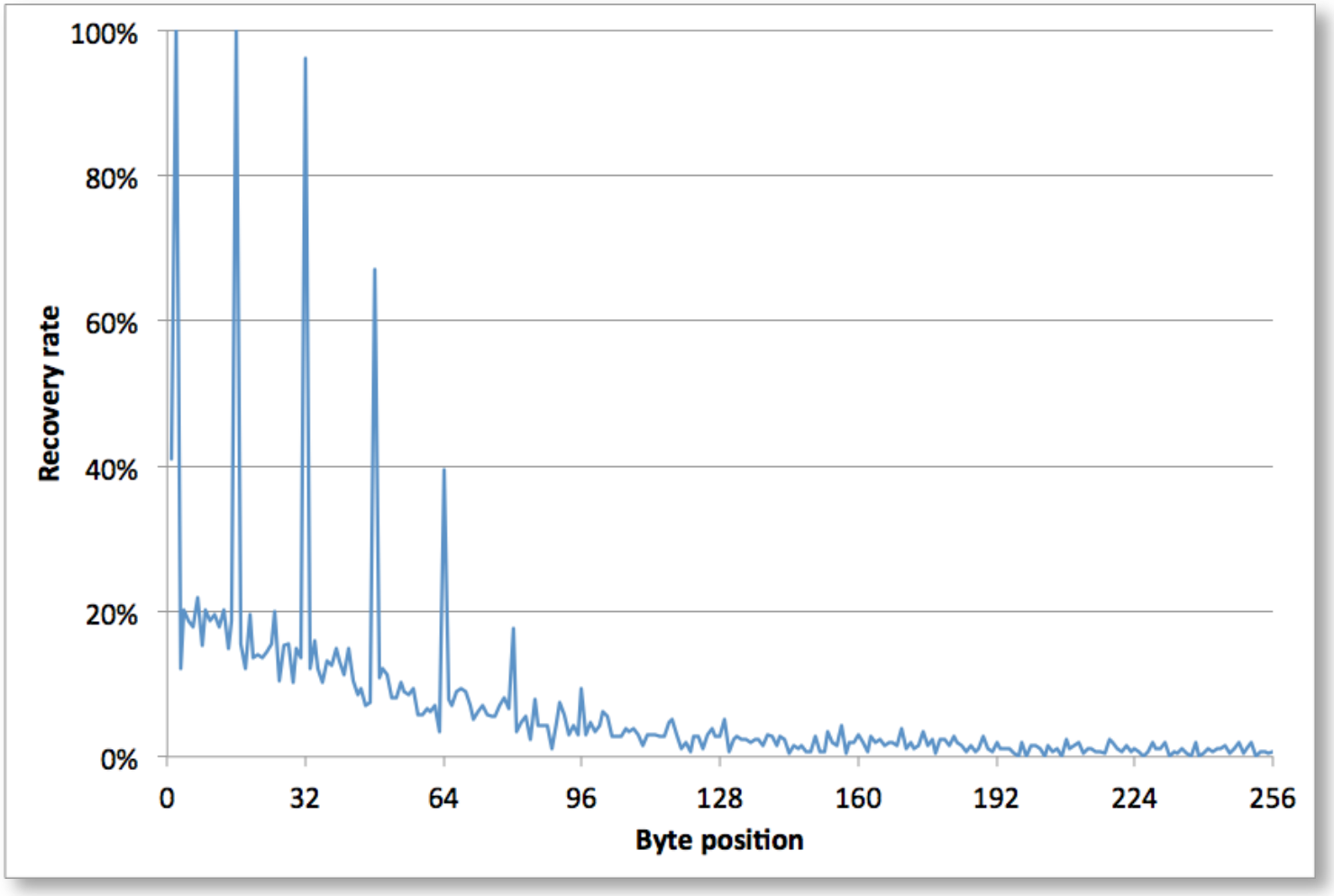
Success Probability
2^{21} Sessions



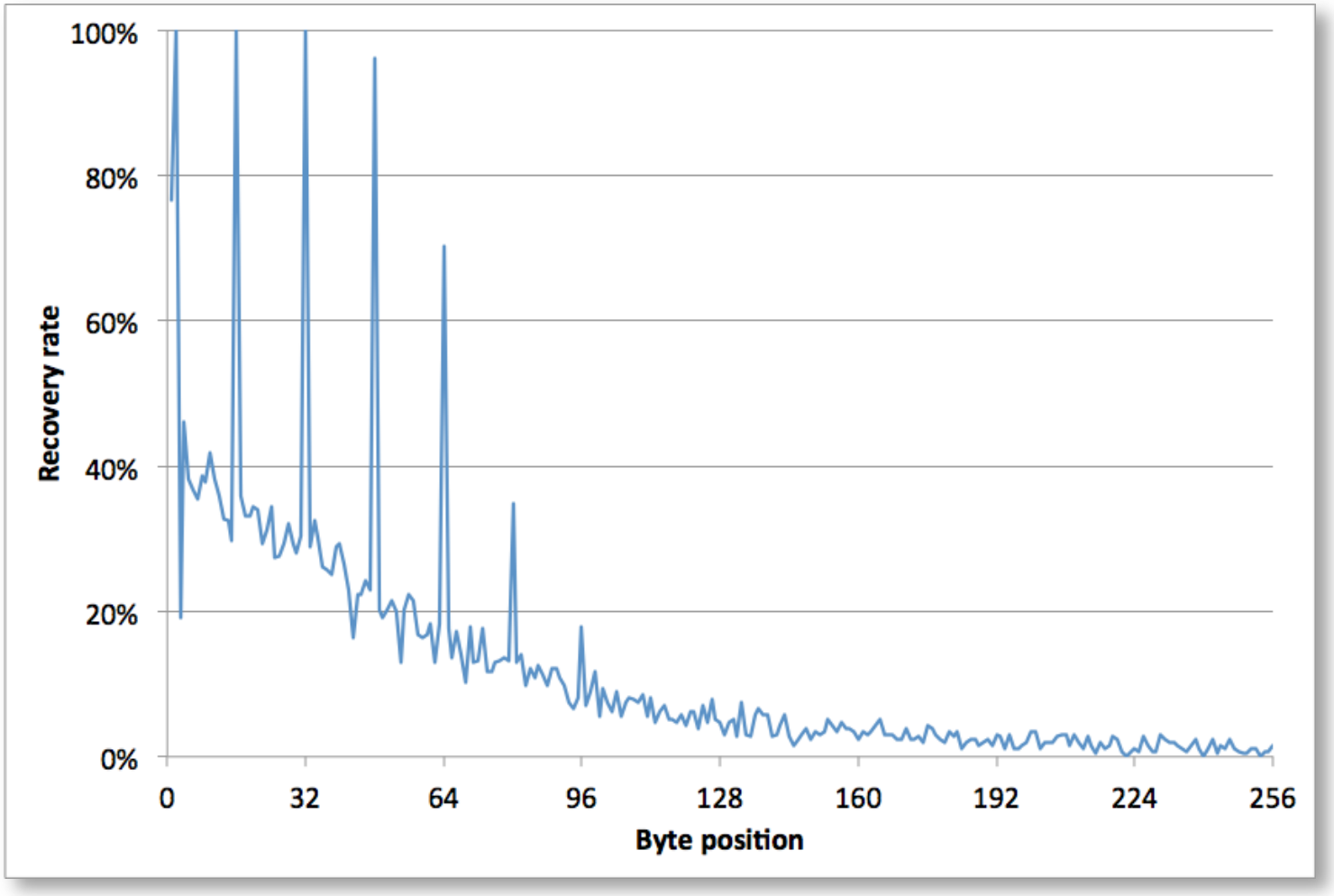
Success Probability
2^{22} Sessions



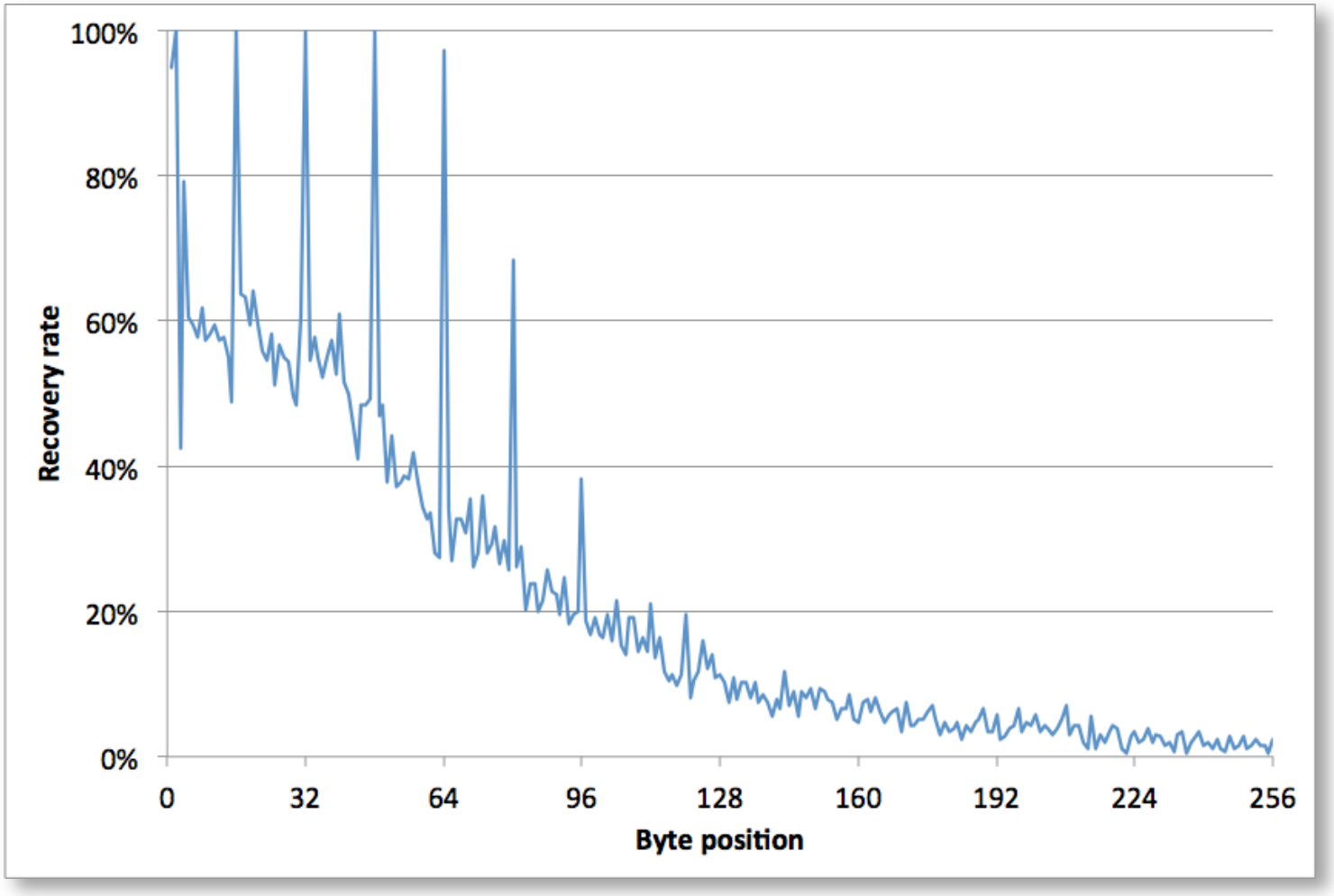
Success Probability
2^{23} Sessions



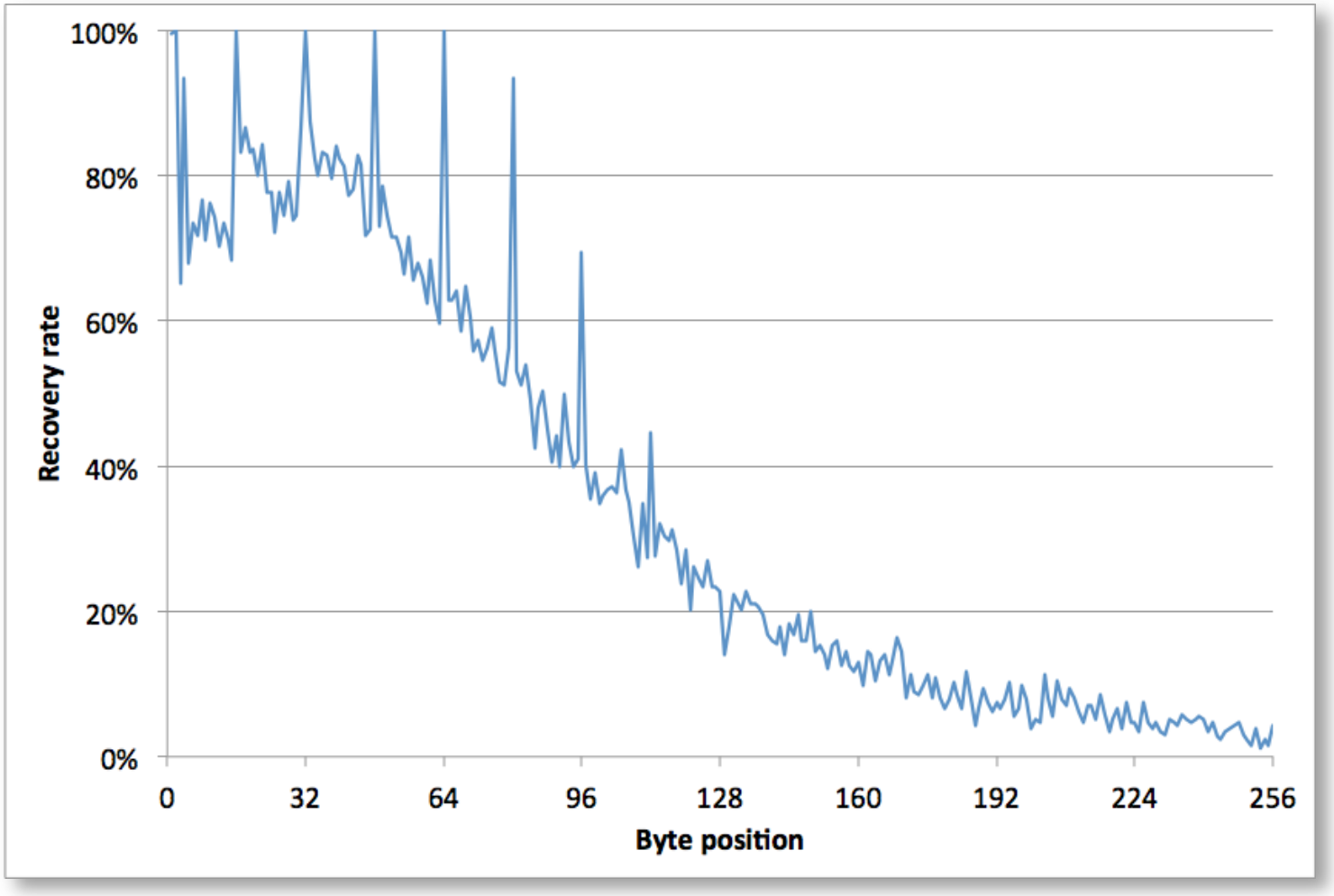
Success Probability
2^{24} Sessions



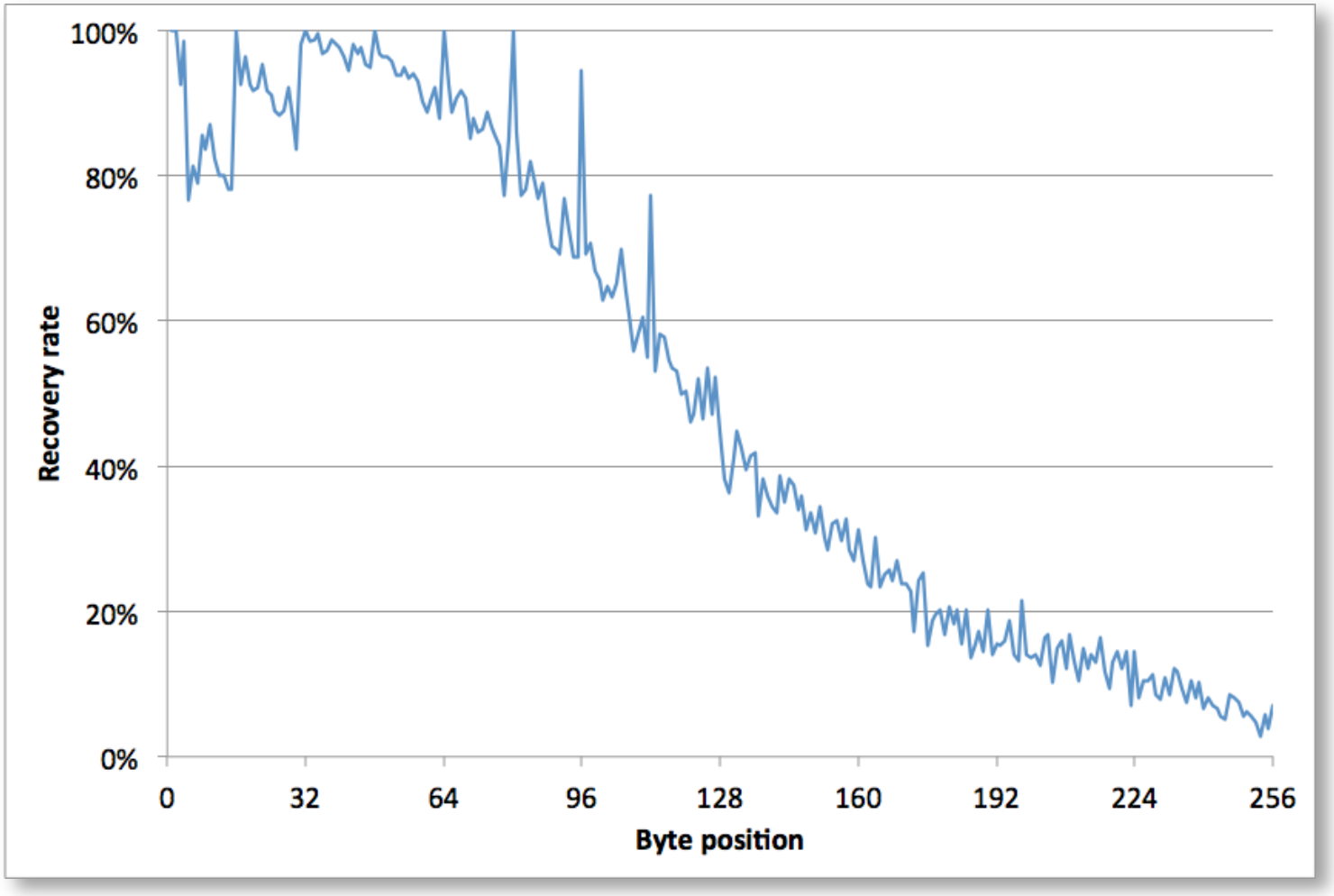
Success Probability
2^{25} Sessions



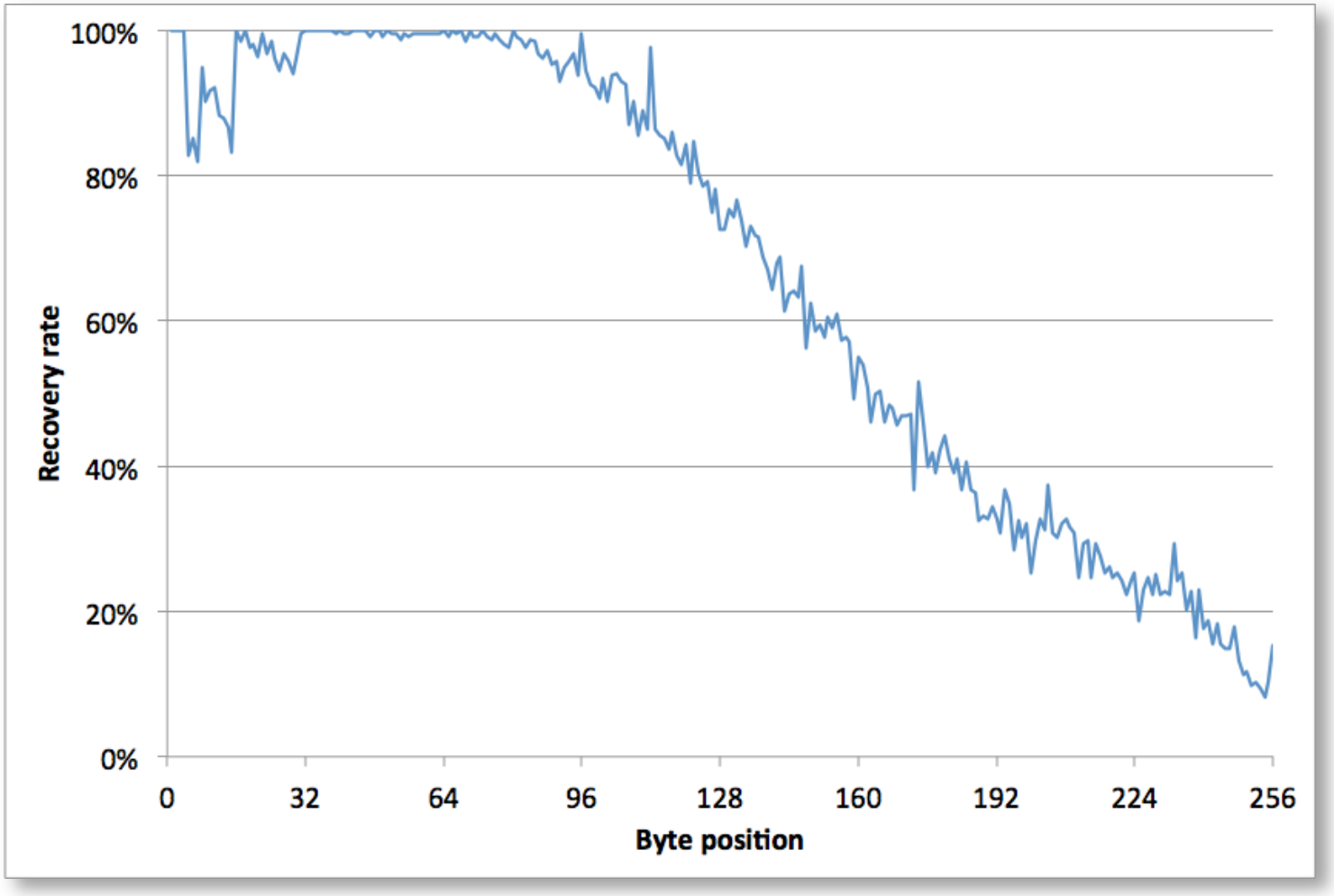
Success Probability
2^{26} Sessions



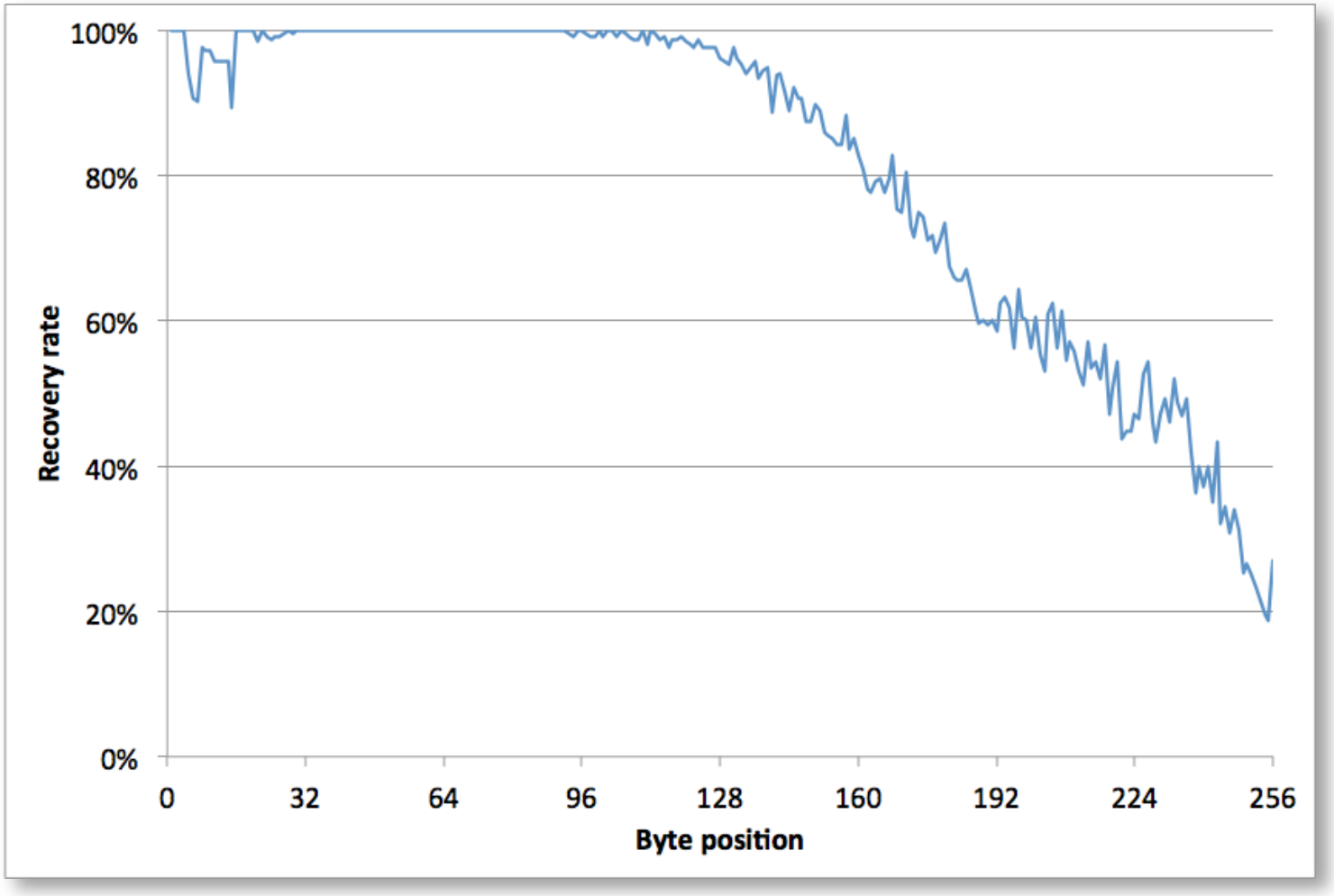
Success Probability
2^{27} Sessions



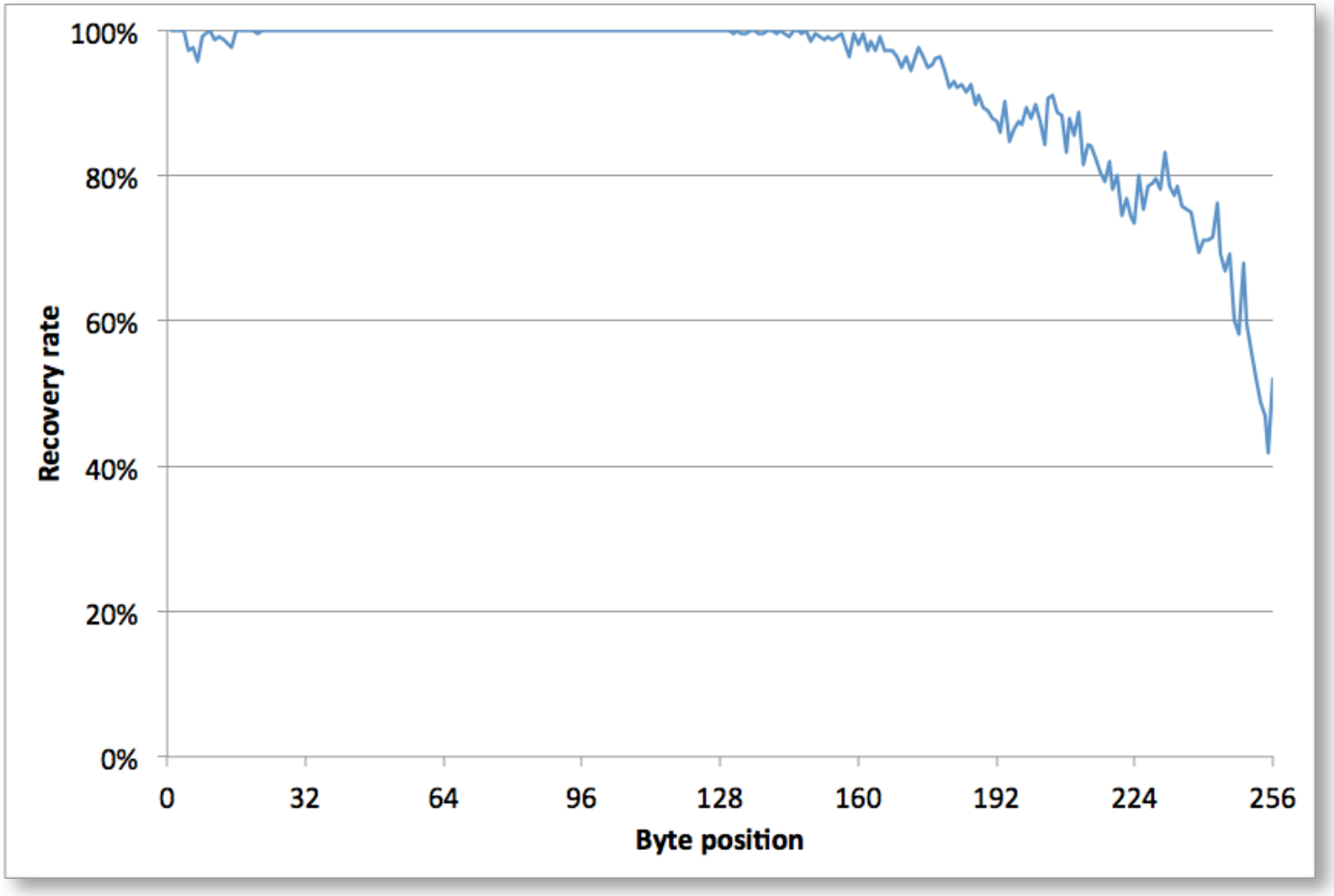
Success Probability
2^{28} Sessions



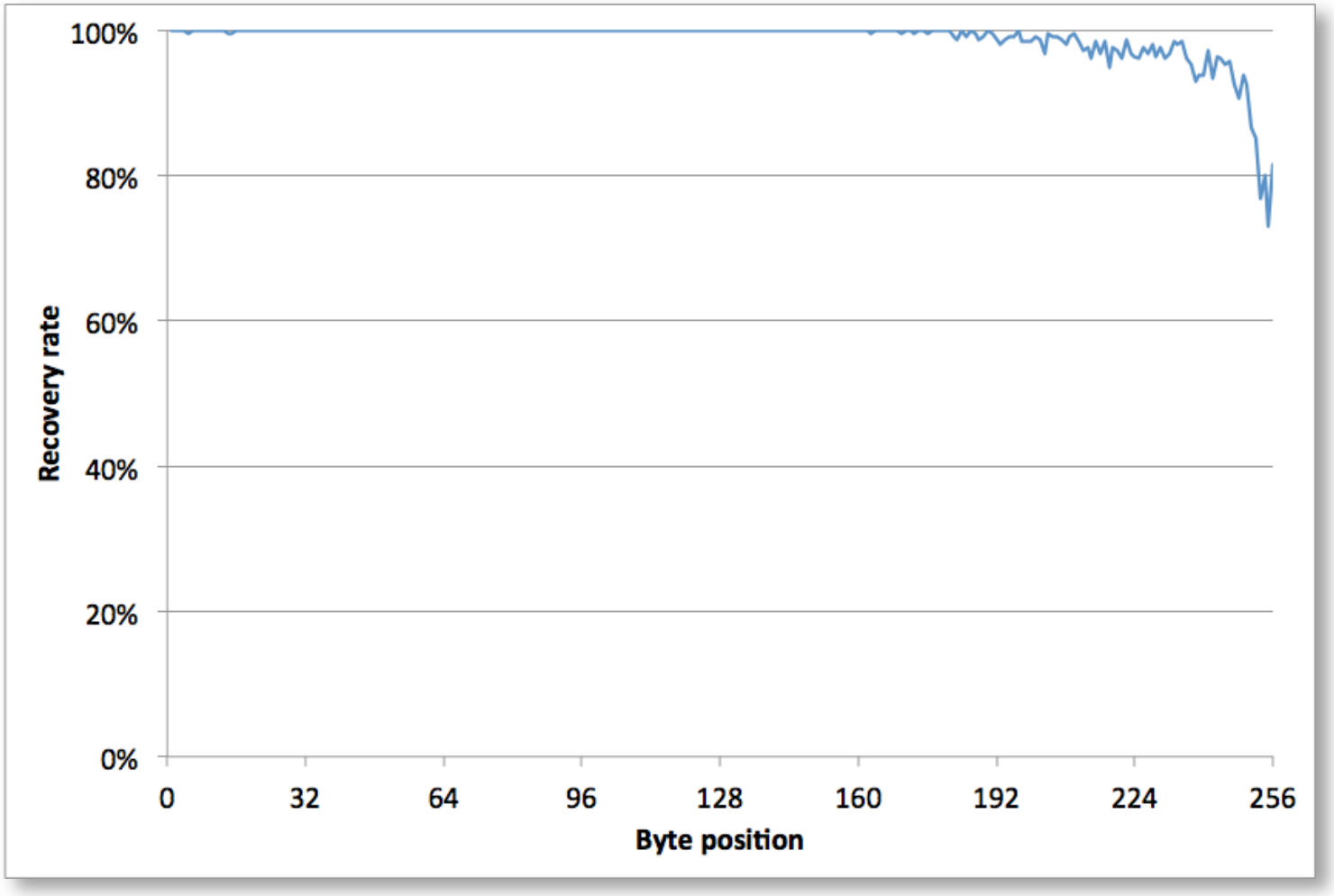
Success Probability
2^{29} Sessions



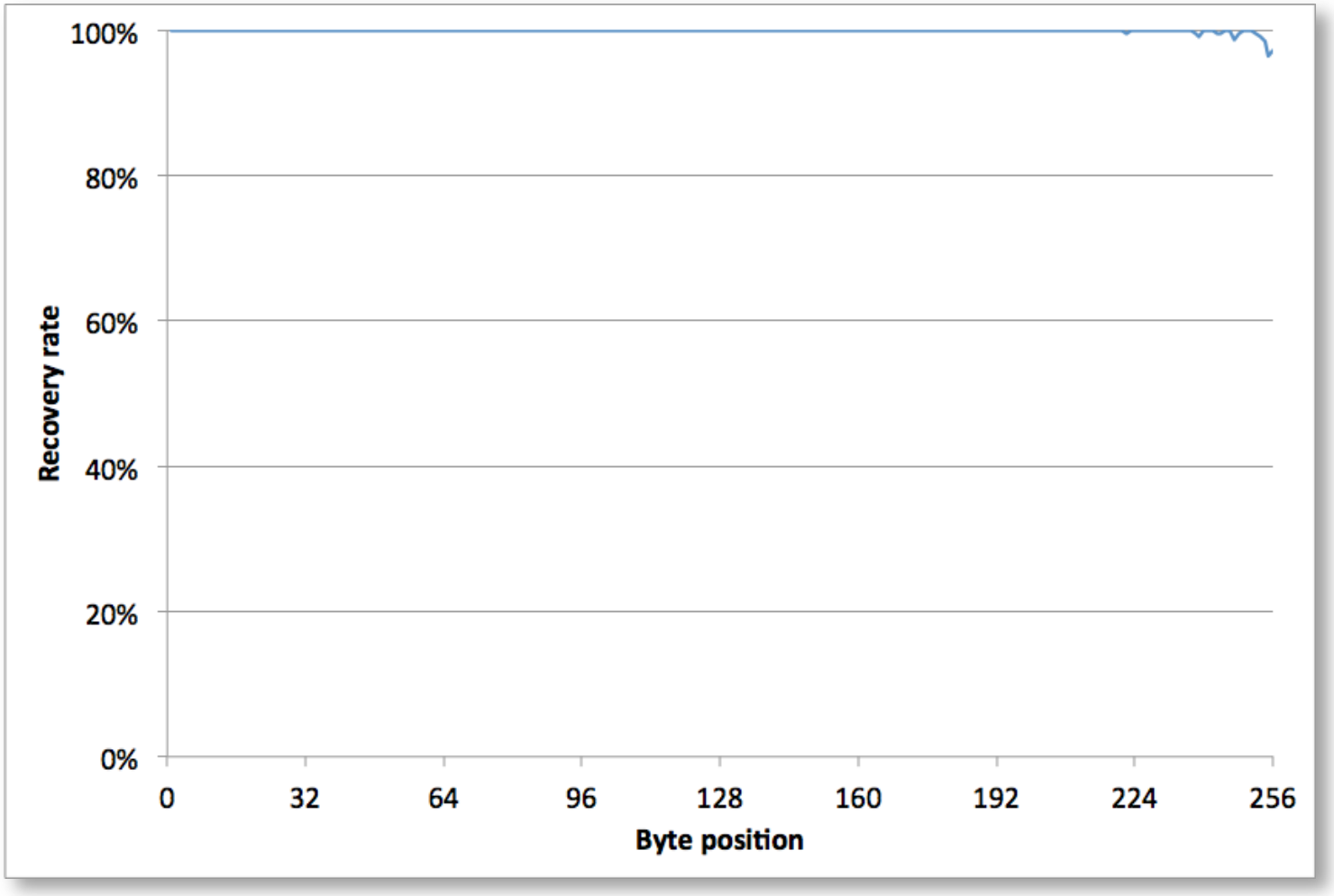
Success Probability
2^{30} Sessions



Success Probability
2^{31} Sessions



Success Probability
2^{32} Sessions



Limitations of Attack

- Requires 2^{28} ~ 2^{32} TLS sessions/connections for reliable recovery.
- Attacker has to force TLS session renegotiation/resumption.
  - No known mechanism from within Javascript.
- Only the first 220 bytes of application data can be targeted.
  - Initial 36 bytes of keystream are used to encrypt last message of Handshake protocol.
- In reality, first 220 bytes of application data usually contain uninteresting HTTP headers.

A Second Attack

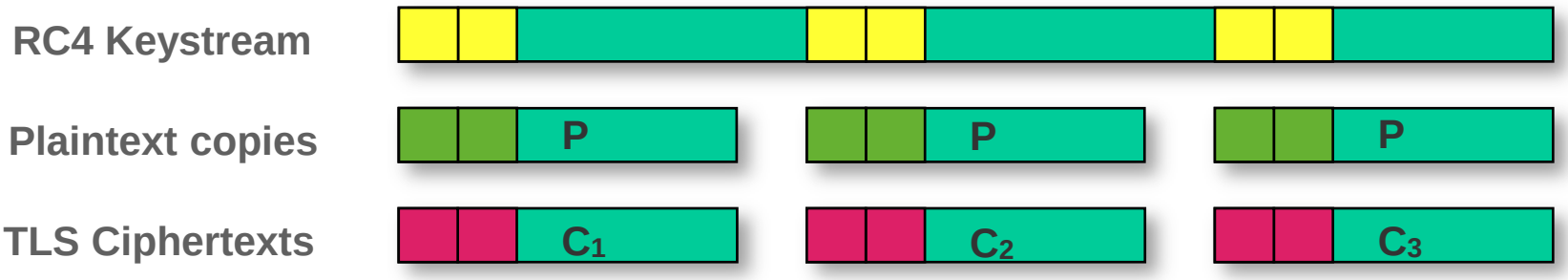
- Fluhrer and McGrew identified biases for consecutive keystream bytes.
  - Persistent throughout keystream.
- Based on these, [ABPPS13] constructed an attack which:
  - Can target *any* plaintext byte positions;
  - Does not require session renegotiation / resumption.

i : keystream byte position mod 256

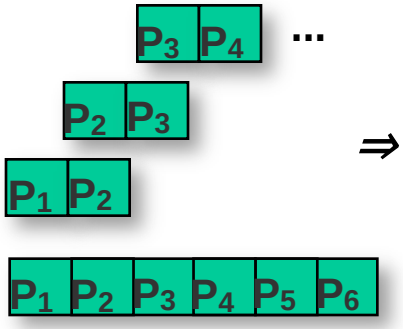
Byte pair	Condition on i	Probability
$(0, 0)$	$i = 1$	$2^{-16}(1 + 2^{-9})$
$(0, 0)$	$i \neq 1, 255$	$2^{-16}(1 + 2^{-8})$
$(0, 1)$	$i \neq 0, 1$	$2^{-16}(1 + 2^{-8})$
$(i + 1, 255)$	$i \neq 254$	$2^{-16}(1 + 2^{-8})$
$(255, i + 1)$	$i \neq 1, 254$	$2^{-16}(1 + 2^{-8})$
$(255, i + 2)$	$i \neq 0, 253, 254, 255$	$2^{-16}(1 + 2^{-8})$
$(255, 0)$	$i = 254$	$2^{-16}(1 + 2^{-8})$
$(255, 1)$	$i = 255$	$2^{-16}(1 + 2^{-8})$
$(255, 2)$	$i = 0, 1$	$2^{-16}(1 + 2^{-8})$
$(129, 129)$	$i = 2$	$2^{-16}(1 + 2^{-8})$
$(255, 255)$	$i \neq 254$	$2^{-16}(1 - 2^{-8})$
$(0, i + 1)$	$i \neq 0, 255$	$2^{-16}(1 - 2^{-8})$

A Second Attack

- Align plaintext with repeating Fluhrer-McGrew biases



- Exploit overlapping nature of plaintext byte pairs to obtain approximate likelihood for plaintext candidates.

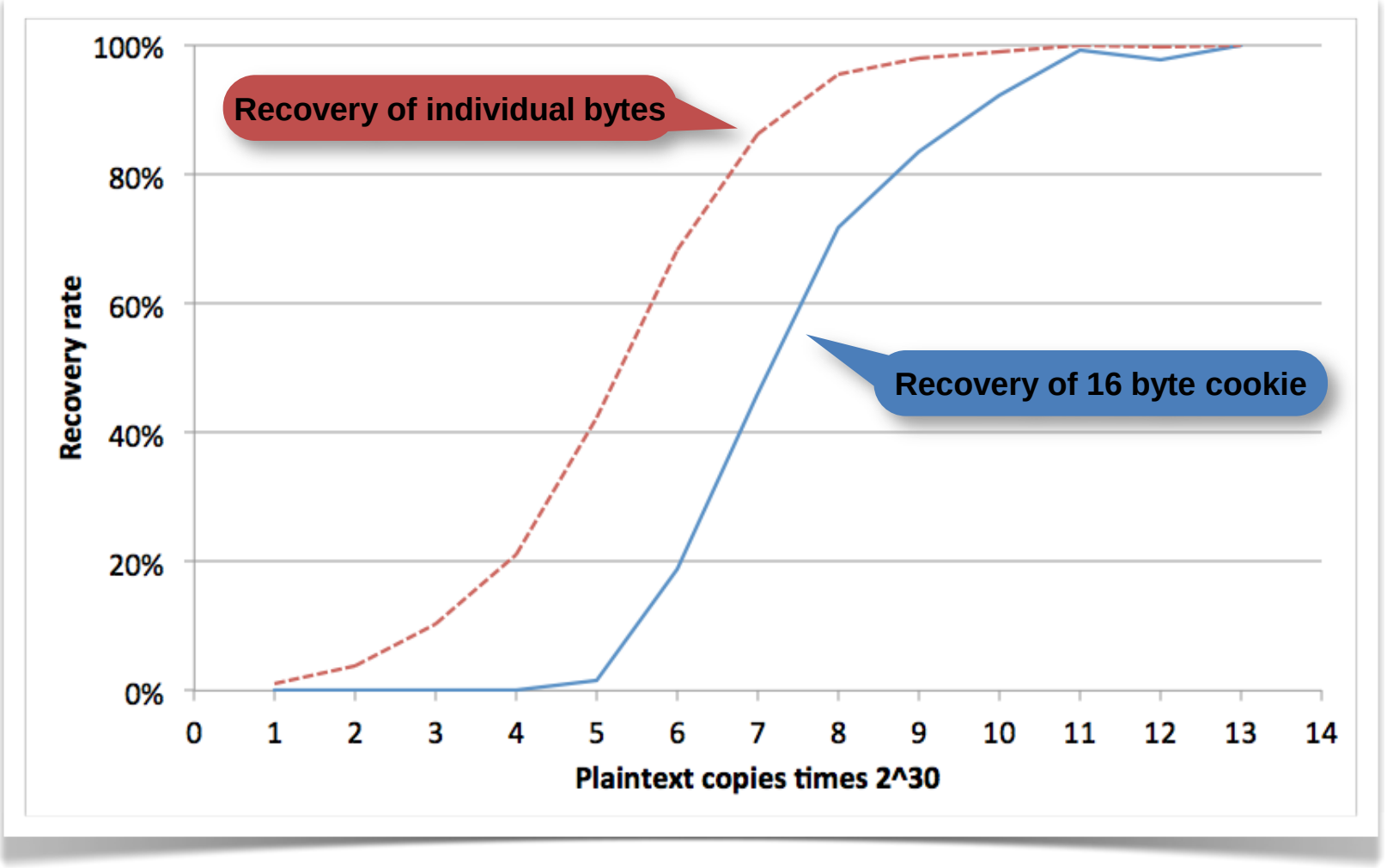


⇒

Approximate likelihood for $P = P_1P_2P_3P_4P_5P_6$

Recovery algorithm:
Viterbi-style algorithm to determine P with highest approximate likelihood

Success Probability



Countermeasures

- Possible countermeasures against the attacks
  - Discard initial keystream bytes (RC4-DropN).
  - Fragment initial records at the application layer.
  - Add random amounts of padding to HTTP.
  - Limit lifetime of cookies or number of times cookies can be sent.
  - (None of these is really effective.)
  - **Stop using RC4 in TLS and switch to another stream cipher.**

Vendor Responses

- Opera has implemented a combination of countermeasures.
- Google focused on implementing TLS 1.2 and AES-GCM in Chrome, now deployed.
- Microsoft: RC4 is disabled by default for TLS in Windows 8.1 and latest Windows server code.
- Development of standards for alternative stream ciphers in TLS underway in IETF.
  - Salsa20/ChaCha20.

CRIME

- Duong and Rizzo [DR12] found a way to exploit TLS's optional compression feature.
  - Similar to idea in 2002 paper by Kelsey [K02].
- 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.
- Countermeasure: disable compression.

BREACH

- BREACH: similar ideas to CRIME, now applied to HTTP compression.
  - http://breachattack.com/
- So now problem arises in the application layer, not crypto layer.
- Cannot so easily disable HTTP compression.

- Bottom-line: we do not yet have a good theoretical handle on how compression interacts with symmetric encryption.
  - A research opportunity!

TLS: Where Do We Stand?

- 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 really TLS 1.2!
  - Support for AES-GCM, AES-CCM.
  - Now available in most main browsers; server-side still patchy.
  - But TLS vulnerable to version rollback attack.
  - Expect further examination of AES-GCM in TLS implementations.

TLS – Current Status?

“This is a dead parrot.”

“He’s not dead. He’s just resting.”

Lessons

- RC4 was known to be weak for many years.
  - Actual exploitation of weaknesses in a TLS context went unexplored.
  - [ABPPS13] needed multi-session mechanism (BEAST technology) to make the attack plausible.

- 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!

Introduction to SSH

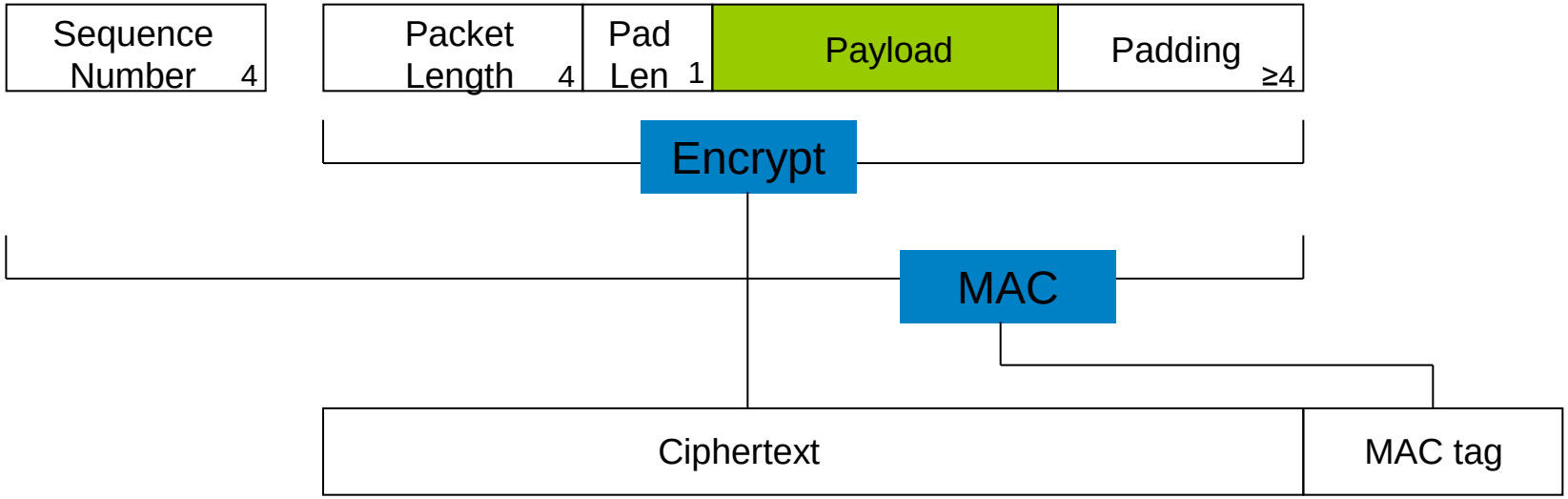
Secure Shell or SSH is a network protocol that allows data to be exchanged using a secure channel between two networked devices. Used primarily on Linux and Unix based systems to access shell accounts, SSH was designed as a replacement for TELNET and other insecure remote shells, which send information, notably passwords, in plaintext, leaving them open for interception. ***The encryption used by SSH provides confidentiality and integrity of data over an insecure network, such as the Internet****.*

– Wikipedia

Introduction to SSH

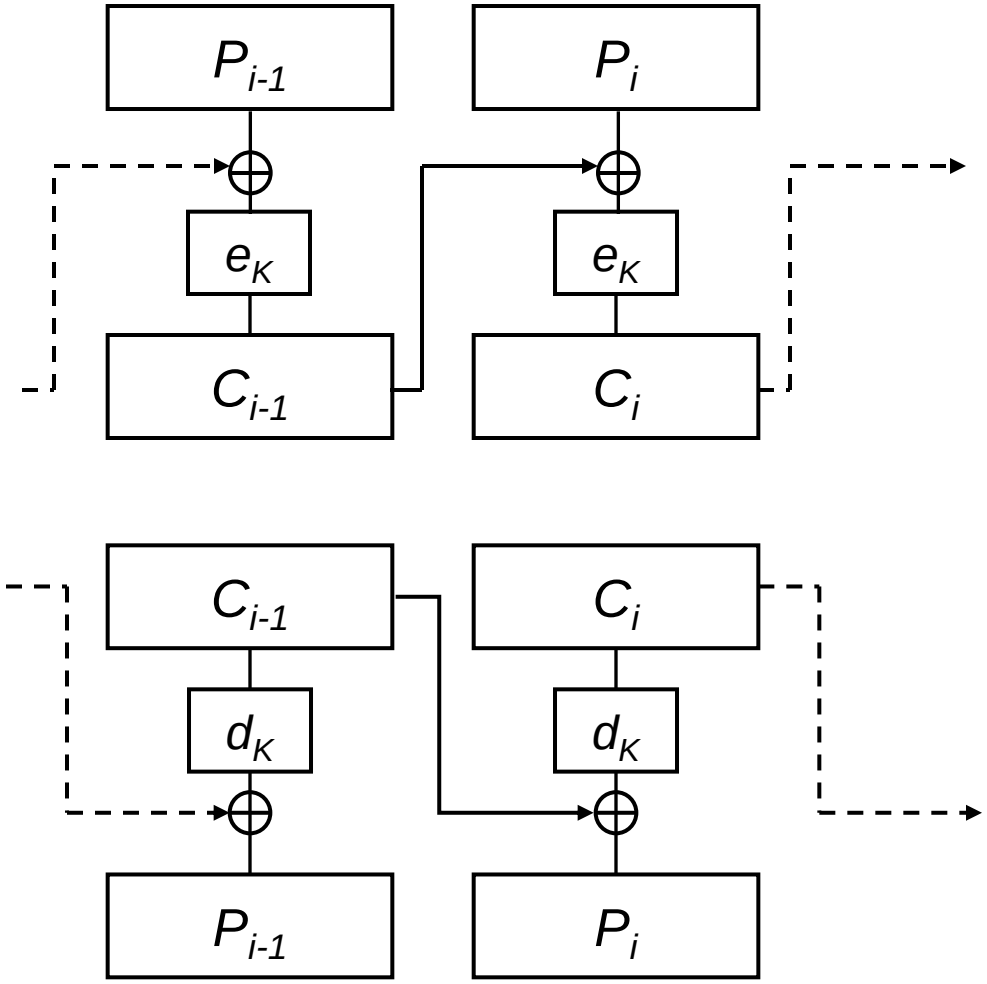
- SSHv1 had several security flaws.
  - Worst ones arising from use of CRC algorithm to provide integrity.
  - Enabling, for example, traffic injection attacks.
- SSHv2 was standardised in 2006 by the IETF in RFCs 4251-4254.
  - But basic specification dates from the late 1990s.
- SSHv2 is widely regarded as providing strong security.
  - One minor flaw that *in theory* allows distinguishing attacks ([D02]; [BKN02]).
  - Simple countermeasure adopted in, for example, OpenSSH.
  - Dozens of different implementations of SSH.

The SSH BPP



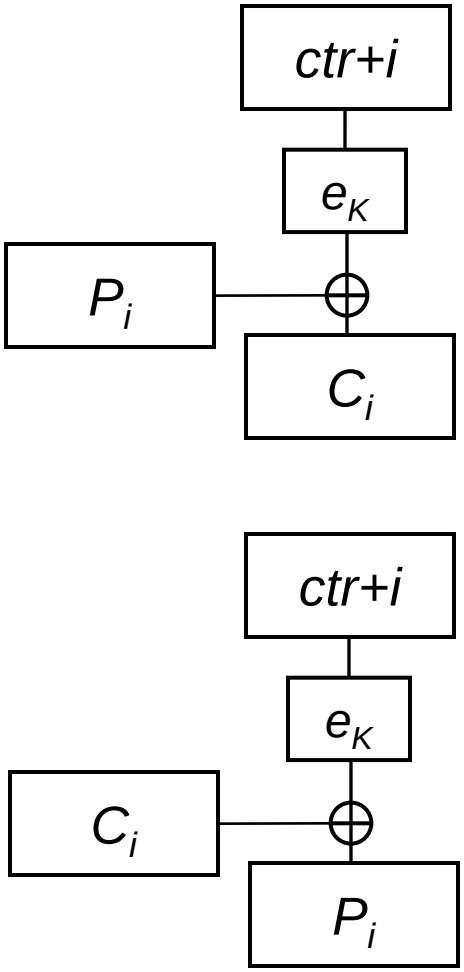
- Encode-then-Encrypt&MAC construction, **not** generically secure.
- Packet length field measures the size of the packet on the wire in bytes and is encrypted to hide the true length of SSH packets.
- Variable length padding is permissible; padding needed for CBC mode and carried over to CTR mode.

CBC Mode in SSH



- RFC 4253 mandates 3DES-CBC and recommends AES-CBC.
  - In fact, all originally specified optional configurations involve CBC mode, and ARCFOUR was the only optional stream cipher.
- SSH uses a chained IV in CBC mode:
  - IV for current packet is the **last** ciphertext block from the **previous** packet.
  - Effectively creates a single stream of data from multiple SSH packets.

CTR Mode in SSH



- CTR mode uses block cipher to build a stream cipher.
- CTR mode for SSH standardised in RFC 4344.
  - Initial value of counter is obtained from handshake protocol.
  - Packet format is preserved from CBC case.
  - Recommends use of AES-CTR with 128, 192 and 256-bit keys, and 3DES-CTR.

Security of the SSH BPP

- Attack of [D02], [BKN02] exploits chained IVs in CBC mode.
  - Same attack vector as Rogaway's 1995 observation.
  - Breaks IND-CCA security of SSH BPP.
  - Low success probability against SSH implementations because of specifics of packet format.
  - Prevented in OpenSSH by optional use of dummy packets to hide IVs until it is too late for attacker to make use of them.
- **Basic message**: SSH BPP using CBC mode with chained IVs is **in**secure according to the standard theoretical notion of security.

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.

Security of the SSH BPP

- Using their models, [BKN02] proved the security of variants of the SSH BPP under reasonable assumptions concerning:
  - The encryption component.
    - Essentially, IND-CPA security.
  - The MAC component.
    - Strong unforgeability and pseudo-randomness.
  - The randomness of the padding scheme.
  - Collision properties of the encoding scheme.
    - In practice, for SSH BPP, this means not too many packets can be encrypted.

Security of the SSH BPP

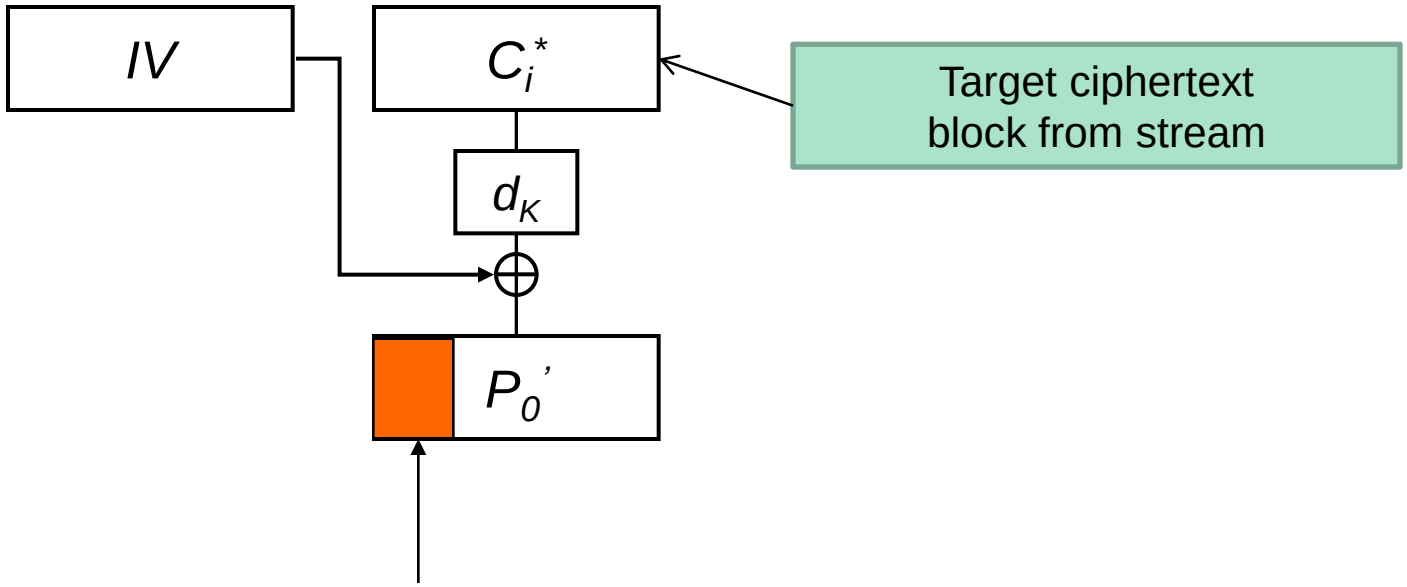
- In particular, [BKN02] established the IND-sfCCA security of SSH-$NPC and SSH-CTR.
  - SSH-$NPC = SSH using a block cipher in CBC mode with explicit, per-packet, random IV and with random padding.
    - In contrast to chained IVs used in SSH BPP.
  - SSH-CTR = SSH using a block cipher in counter mode, with counter maintained at sender and receiver.

Attacking the SSH BPP

- [APW09]: plaintext recovery attacks against SSH BPP when using CBC mode.
  - Much stronger than distinguishing attack of [D02], [BKN02]!

- These attacks exploit the interaction of the following features of the BPP *specification*:
  - The attacker can send data on an SSH connection in small chunks (TCP).
  - A MAC failure is visible on the network.
  - The packet length field encodes how much data needs to be received before the MAC is received and the integrity of the packet can be checked.

Attacking the SSH BPP (Theory)

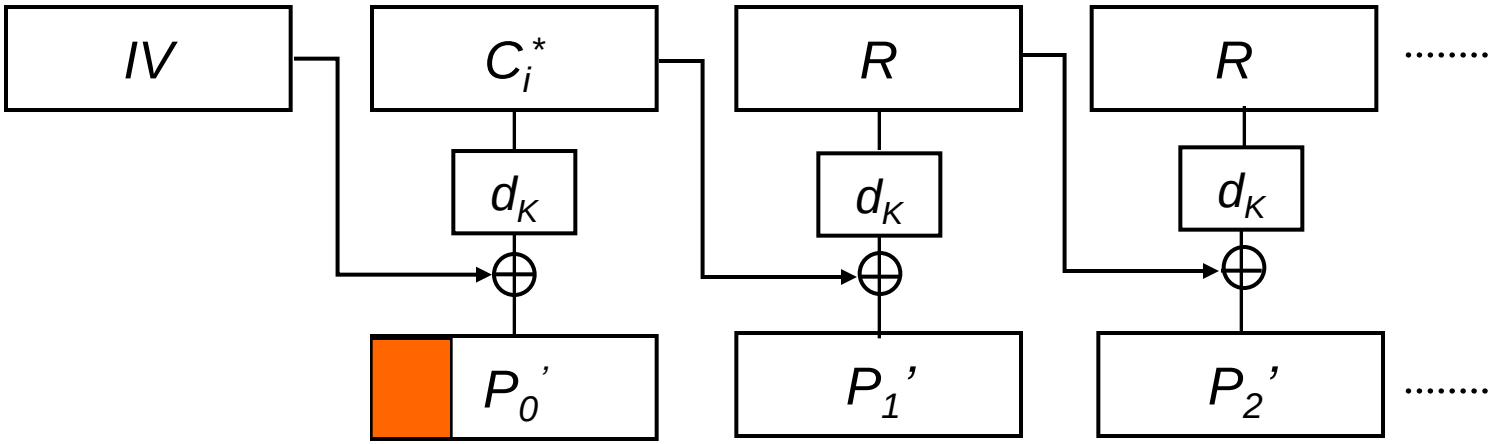


- The receiver will treat the first 32 bits of the calculated plaintext block as the packet length field for the new packet.
- Here:

$$P_0' = IV \oplus d_K(C_i^*)$$

where *IV* is known from the previous packet.

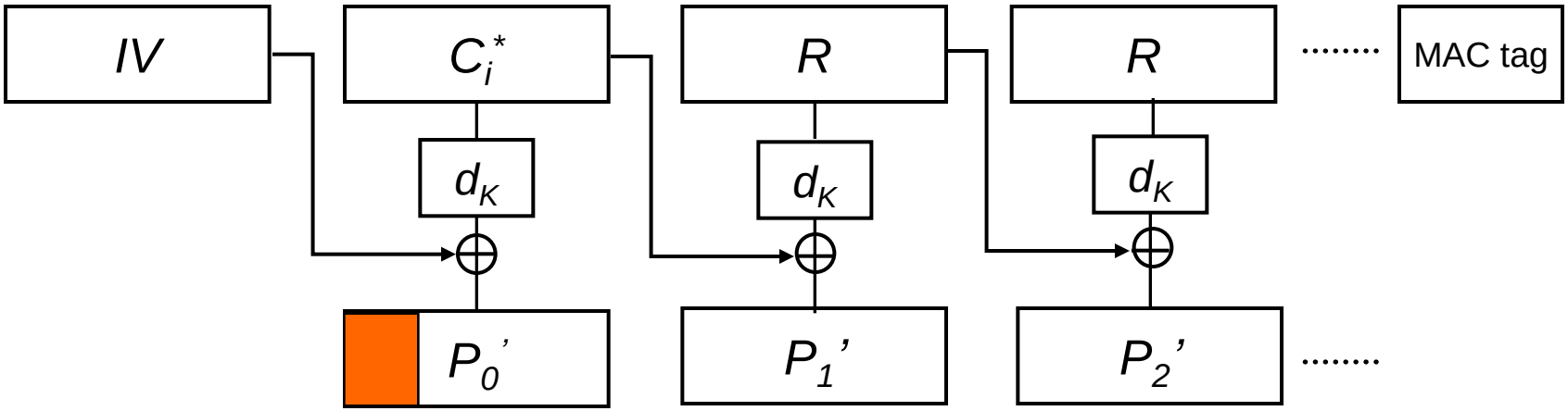
Attacking the SSH BPP (Theory)



The attacker then feeds random blocks to the receiver.

– One block at a time, waiting to see what happens at the server when each new block is processed.

Attacking the SSH BPP (Theory)



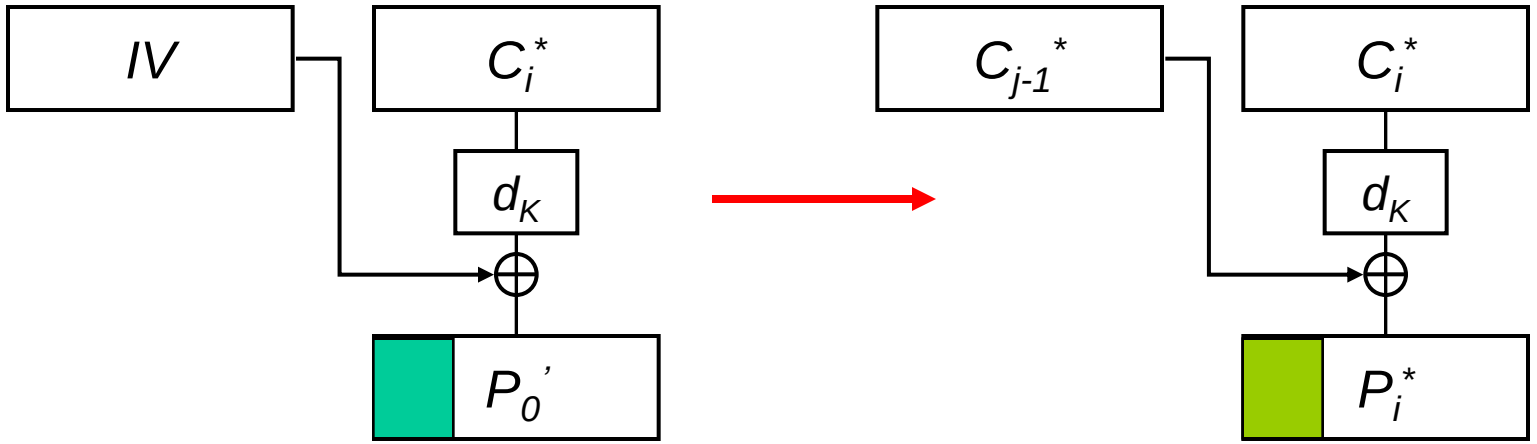
- Eventually, once enough data has arrived, the receiver will receive what it thinks is the MAC tag.
- The receiver will then check the MAC.
  - This check will fail with overwhelming probability.
  - Consequently the connection is terminated (with an error message).
- How much data is “enough” so that the receiver decides to check the MAC?

Attacking the SSH BPP (Theory)

- The receiver **has** to use the packet length field to decide when the MAC tag has arrived.
- Hence an attacker who counts the number of bytes needed to cause connection termination learns the packet length field.
- That is, the attacker learns the first 32 bits of:

$$P_0' = IV \oplus d_K(C_i^*).$$

Attacking the SSH BPP (Theory)



- Knowing IV and 32 bits of P_0', the attacker can now recover 32 bits of the **target** plaintext block:

$$P_i^* = C_{i-1}^* \oplus d_K(C_i^*) = C_{i-1}^* \oplus IV \oplus P_0'$$

Attack Performance (Theory)

- As described, this simple attack succeeds in recovering 32 bits of plaintext from an arbitrary ciphertext block with probability 1.
  - But requires the injection of about 2^{31} random bytes to trigger the MAC check.
  - And leads to an SSH connection tear-down.
- Still, the attack breaks the SSH BPP.
- The attack still works if a fresh IV is used for each new SSH packet.
  - Breaking SSH-$NPC that was proven secure in [BKN02].

Attacking OpenSSH

- OpenSSH is the most popular implementation of the SSH RFCs.
  - Open-source, distributed as part of OpenBSD.
  - OpenSSH webpages state that OpenSSH accounts for more than 80% of all deployed SSH servers.
  - www.openssh.org/usage/index.html

- [APW09] worked with OpenSSH 5.1.
  - Version 5.2 released 23/02/2009 partly as a consequence of their work, current version is 6.4.

Attacking OpenSSH

- In OpenSSH 5.1, two sanity checks are carried out on the packet length field after the first block is decrypted.
  - $5 \leq \text{packet\_length} \leq 2^{18}$
  - $\text{packet\_length} + 4 \,\%\, \text{block\_length} = 0$
- When either of the checks fails, the SSH connection is terminated.
  - But in subtly different ways that leaks some plaintext information.
- If the length checks pass, then OpenSSH 5.1 waits for more bytes.
- Finally, when the MAC check fails, a third type of connection termination is seen.

Attacking OpenSSH

- The manner in which OpenSSH 5.1 behaves on failure allows:
  - A first attack verifiably recovering 14 bits of plaintext with probability 2^{-14}.
  - A second attack verifiably recovering 32 bits of plaintext with probability 2^{-18} (for a 128-bit block cipher).
  - The attacks require injection of (roughly) 2^{18} bytes.
- Boost success rate in multi-session attack.
- The attacks in [APW09] worked in practice.
  - Implemented in a virtualized environment with server code patched to boost success rate.

Possible Countermeasures

- Use counter mode.
  - The attack no longer applies.
  - But *stateful* version of counter mode needed.
    - If there's an explicit counter in packets, then a version of the attacks still works.
  - As standardised in RFC 4344.
- *Enforce* use of counter mode.
  - Not standards compliant with the RFCs as they are currently written.
  - Some implementations do not support counter mode at all, creating backwards compatibility issue.
  - "*Only a cryptographer would suggest this...*"

What Went Wrong with the Theory?

- The security model of [BKN02] *does* model errors arising during the BPP decryption process.
  - Connection teardown is modeled by disallowing access to decryption and encryption oracles after any error event.
  - Errors can arise from decryption, decoding or MAC checking.
- But only a single type of error message is output.
  - The 2^{-14} attack against OpenSSH exploits the fact that different error events *are* distinguishable.
- And the model assumes that decoding errors arise *before* MAC errors.
  - While the OpenSSH implementation only does decoding *after* the MAC has been checked.

Limitations of [BKN02]

- The model assumes that plaintexts and ciphertexts are “atomic”.
  - All oracle queries in the model involve complete plaintexts or ciphertexts.
  - But the attacks exploit the ability to deliver ciphertexts one block (or even one byte!) at a time and observe behaviour.
- The model does not allow for *plaintext-dependent* decryption.
  - The packet length field never appears in the model.
  - But implementations *must* make use of this field during the decryption process.
  - And, as we’ve seen, the manner in which this field is treated is critical for security.
- Models are just models.

New Security Analysis of SSH

- [PW10]:
  - Develops a new security model addressing limitations of the model used in [BKN02]
    - LOR-BSF-CCA security;
  - Builds an accurate description of SSH-CTR as specified in RFCs and implemented in OpenSSH;
  - Proves the security of this description of SSH-CTR in the new model.
- [BDPS12]:
  - More general security modelling for SSH-like protocols.
  - Security against chosen-fragment attacks (IND-CFA).
  - Formalisation of boundary-hiding (BH).
  - Relationship between IND-CFA, BH and DoS-resistance.

Lessons

- SSH attack is trivial.
  - Once you see it!
  - Troubling that it lurked in specification for years.
- SSH design goals raise interesting new theory questions.
  - How do IND-CFA, BH and DoS-resistance interact with each other?
- [PW10] analysis of SSH-CTR is at the limits of (this) human's ability to generate models and proofs.
  - Complexity arises from complexity of protocol we're trying to model.
  - c.f. recent developments in TLS analysis, introduction of machine-generated/machine-checkable proofs.

Final Thoughts

- Good algorithm design is hard.
- But so is good protocol design.
- Attacks are usually obvious in retrospect.
  - But so is most theory!
- Finding attacks is high-risk, high-reward.
- Value of attacks on paper versus attacks in practice.
  - Implemented attacks needed to convince practitioners.
  - On-paper attack often the harbinger of a practical attack.

Literature

Basic theory for symmetric encryption (highly selective list):

- [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.
- [BDPS12] Boldyreva *et al.*, Eurocrypt 2012.
- [BDPS13] Boldyreva *et al.,* FSE 2013.

Literature

Attacks on TLS symmetric crypto:

- [V02] Vaudenay, Eurocrypt 2002.
- [M02] Moeller, http://www.openssl.org/~bodo/tls-cbc.txt, 2002.
- [CHVV03] Canvel *et al.*, Crypto 2003.
- [B04] Bard, eprint 2004/111.
- [B06] Bard, SECRYPT 2006.
- [PRS11] Paterson *et al.*, Asiacrypt 2013.
- [DR11] Duong and Rizzo, “Here come the XOR Ninjas”, 2011.
- [DR12] Duong and Rizzo, CRIME, 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.

Literature

Attacks on IPsec symmetric crypto:

- [B97] Bellovin, https://www.cs.columbia.edu/~smb/papers/badesp.pdf.
- [PY06] Paterson and Yau, Eurocrypt 2006.
- [DP07] Degabriele and Paterson, IEEE S&P 2007.
- [DP10] Degabriele and Paterson, ACM-CCS 2010.

A selection of other interesting attacks on symmetric crypto:

- [K02] Kelsey, FSE 2002.
- [APW09] Albrecht *et al.*, IEEE S&P 2009.
- [DR11b] Duong and Rizzo, IEEE S&P 2011.
- [JS11] Jager and Somorovsky, ACM-CCS 2011.
- [JPS13] Jager *et al.*, NDSS 2013.