Vulnerabilities in TLS

Kenny Paterson

Information Security Group

@kennyog ; www.isg.rhul.ac.uk/~kp

ROYAL
HOLLOWAY
UNIVERSITY
OF LONDON

Opening Remarks

Interact!

No theory!*

Partial repetition of talks at various Summer and Winter schools!

No claims to completeness!

*Well, almost none. Which is actually a pity.

Overview

Lecture 1: Introduction to TLS; BEAST and CRIME attacks

Lecture 2: Padding oracles, Lucky 13, and more

Lecture 3: RC4 attacks

Lecture 4: tbd

Real World Cryptography 2015
London, UK, 7-9 January 2015
http://www.realworldcrypto.com/rwc2015
#realworldcrypto

Speakers to include:

Elena Andreeva (K.U. Leuven)

Dan Bogdanov (Cybernetica)

Sasha Boldyreva (Georgia Tech)

Claudia Diaz (K.U. Leuven)

Roger Dingledine (Tor project)

Ian Goldberg (U. Waterloo)

Arvind Mani (LinkedIn)

Luther Martin (Voltage Security)

Elisabeth Oswald (U. Bristol)

Scott Renfro (Facebook)

Ahmad Sadeghi (TU Darmstadt)

Elaine Shi (UMD)

Brian Sniffen (Akamai)

Nick Sullivan (CloudFlare)

TLS Overview

Introducing TLS

SSL = Secure Sockets Layer.

Developed by Netscape in mid 1990s.

SSLv2 now deprecated; SSLv3 still widely supported.

TLS = Transport Layer Security.

IETF-standardised version of SSL.

TLS 1.0 = SSLv3 with minor tweaks, RFC 2246 (1999).

TLS 1.1 = TLS 1.0 + tweaks, RFC 4346 (2006).

TLS 1.2 = TLS 1.1 + more tweaks, RFC 5246 (2008).

TLS 1.3?

6

Introducing TLS

Originally for secure e-commerce, now used much more widely.

Retail customer access to online banking facilities.

User access to gmail, facebook, Yahoo.

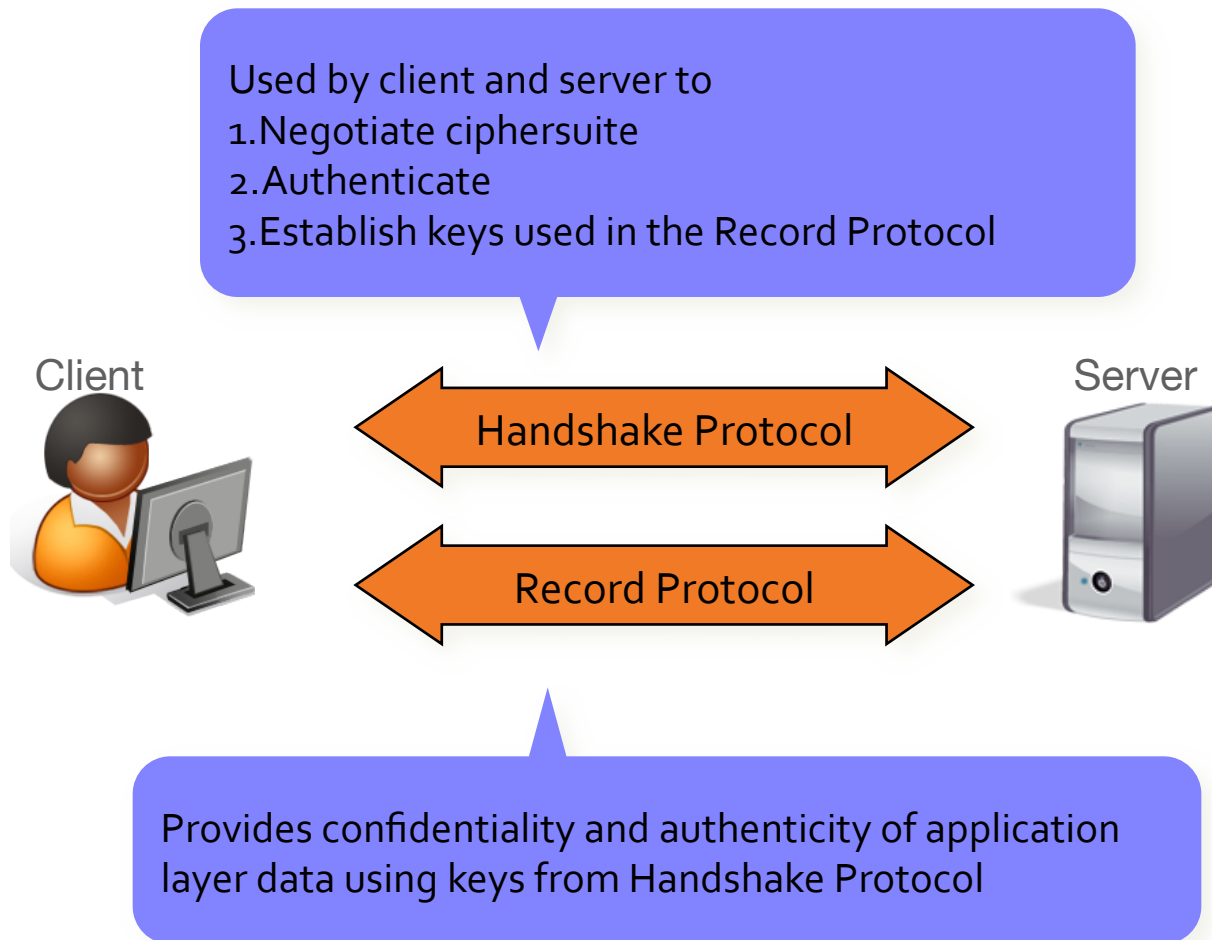Mobile applications, including banking apps.

Payment infrastructures.

User-to-cloud.

Post Snowden: back-end operations for google, yahoo, …
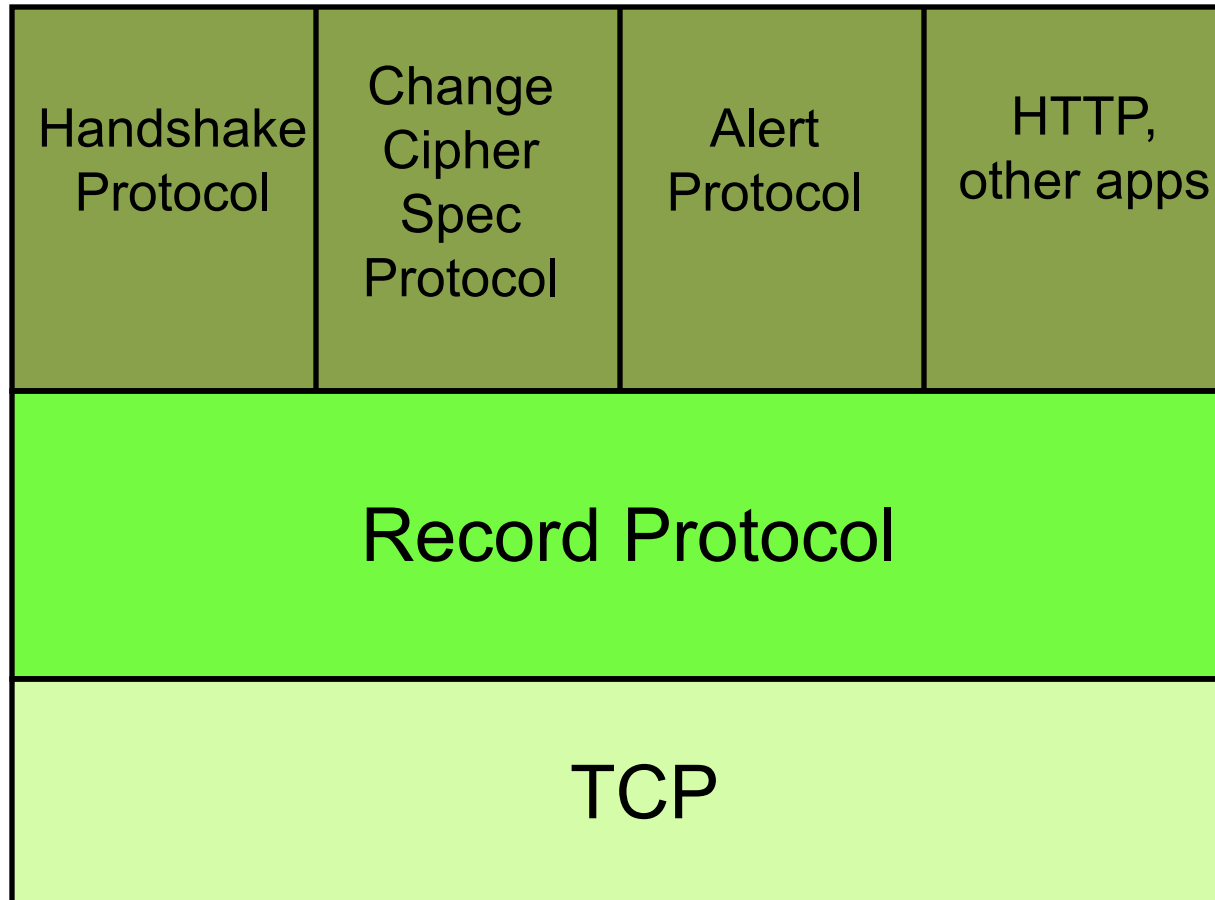
TLS has become the *de facto* secure protocol of choice.

Used by hundreds of millions of people and devices every day.

Highly Simplified View of TLS

Used by client and server to
1. Negotiate ciphersuite
2. Authenticate
3. Establish keys used in the Record Protocol

Client

Server

Handshake Protocol

Record Protocol

Provides confidentiality and authenticity of application layer data using keys from Handshake Protocol

TLS Protocol Architecture

Handshake Protocol	Change Cipher Spec Protocol	Alert Protocol	HTTP, other apps

Record Protocol

TCP

The TLS Ecosystem (1/3)

- Servers

  - Including managed service providers (CloudFlare, Akamai)

- Clients

  - Of all shapes and sizes

- Certification service providers

  - Of all shapes , sizes and levels of security

- Software vendors

  - From Google down to one-man open-source operations

  - OpenSSL somewhere in-between

- Hardware vendors

The TLS Ecosystem (2/3)

- TLS versions:

    - SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2

    - Many servers even still support SSL 2.0

- 200+ ciphersuites

    - https://www.thesprawl.org/research/tls-and-ssl-cipher-suites

    - Some very common, e.g.

        TLS_RSA_WITH_AES_128_CBC_SHA256

    - Some highly esoteric, e.g.

        TLS_KRB5_WITH_3DES_EDE_CBC_MD5

    - Some offering no security:

        TLS_NULL_WITH_NULL_NULL !

- TLS extensions

    - Too numerous to mention.

- DTLS

The TLS Ecosystem (3/3)

- IETF TLS Working Group
    - Also IETF UTA Working Group (UTA = Using TLS in Applications)
    - And CFRG (Crypto Forum Research Group)

- Growing community of researchers
    - Blackhat or Crypto?
    - Attacks or security proofs?
    - Handshake Protocol, Record Protocol or both?
    - Full protocol including session resumption, renegotiation, ciphersuite negotiation?
    - Provable security or formal methods or something else?
    - Game-based, UC or constructive cryptography?

- The TLS ecosystem has become very complex and vibrant.

The TLS Ecosystem (4/3) (bonus slide)

A Newsworthy Protocol

TLS has been in the news…..

BEAST (2011), CRIME (2012), Lucky 13, RC4 attacks (both 2013).

Renegotiation attack (2009), triple Handshake attack (2014).

Poor quality of implementations (particularly in certificate handling).

"Why Eve and Mallory Love Android" (2012)

"The most dangerous code in the world" (2012)

Apple *goto fail* (2013)

GnuTLS certificate processing bug (2013)

Truncation and cookie cutter attacks (2013, 2014)

OpenSSL CCS bug (2014)

Frankencerts (2014)

Mostly tech press, with limited crossover to mainstream media.

A Newsworthy Protocol
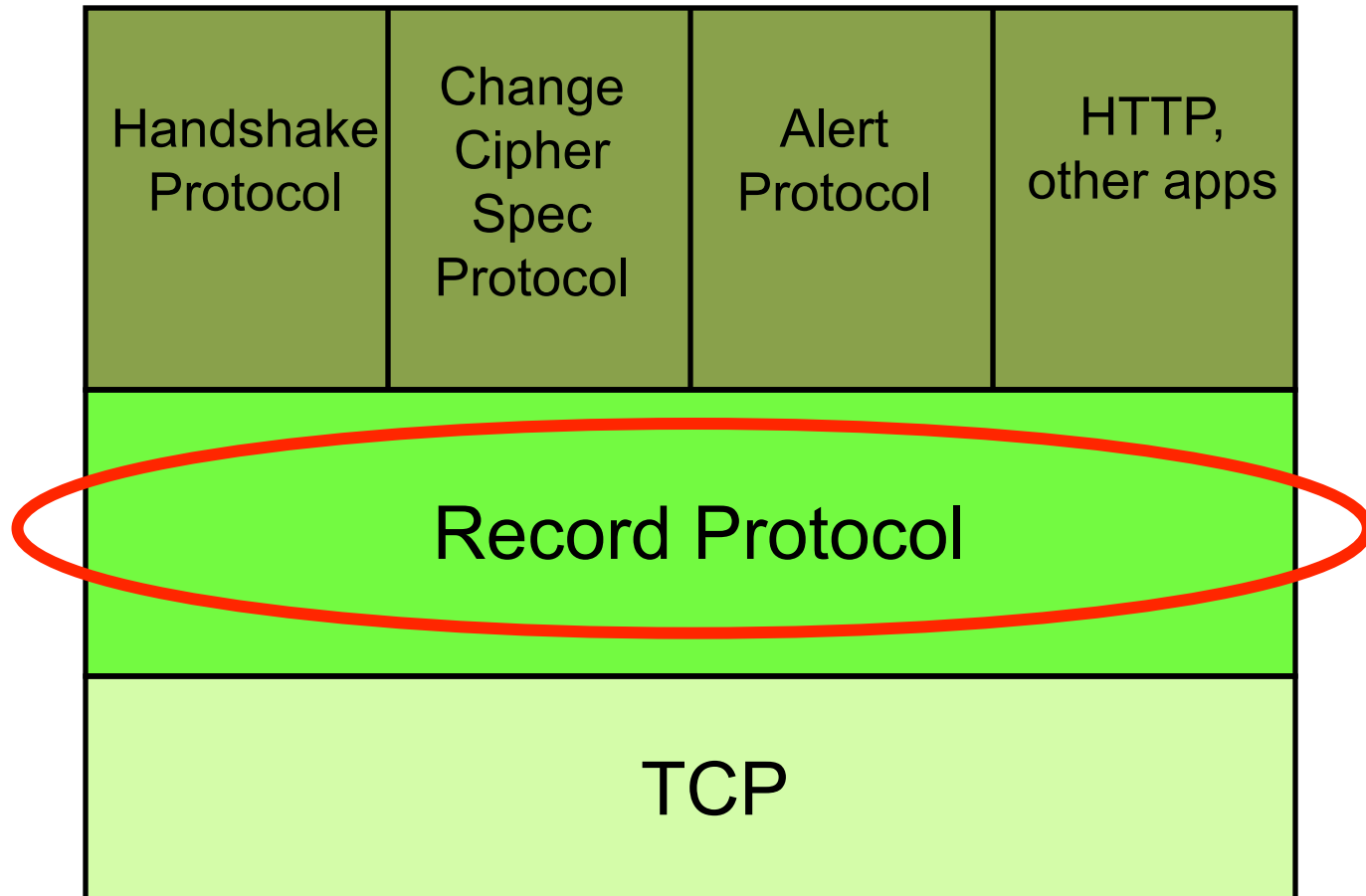
TLS has *really* been in the news.....

.... Heartbleed bug.

What is it about Heartbleed that caught the wider media's imagination?

- Pressure built and the dam finally broke?

- Severity of the threat (leakage of private information, inc. server private keys)?

- Widespread use of OpenSSL.

- A good logo?

TLS Protocol Architecture

Handshake Protocol	Change Cipher Spec Protocol	Alert Protocol	HTTP, other apps

Record Protocol

TCP

Uploaded By: anonymous

TLS Record Protocol

TLS Record Protocol provides:
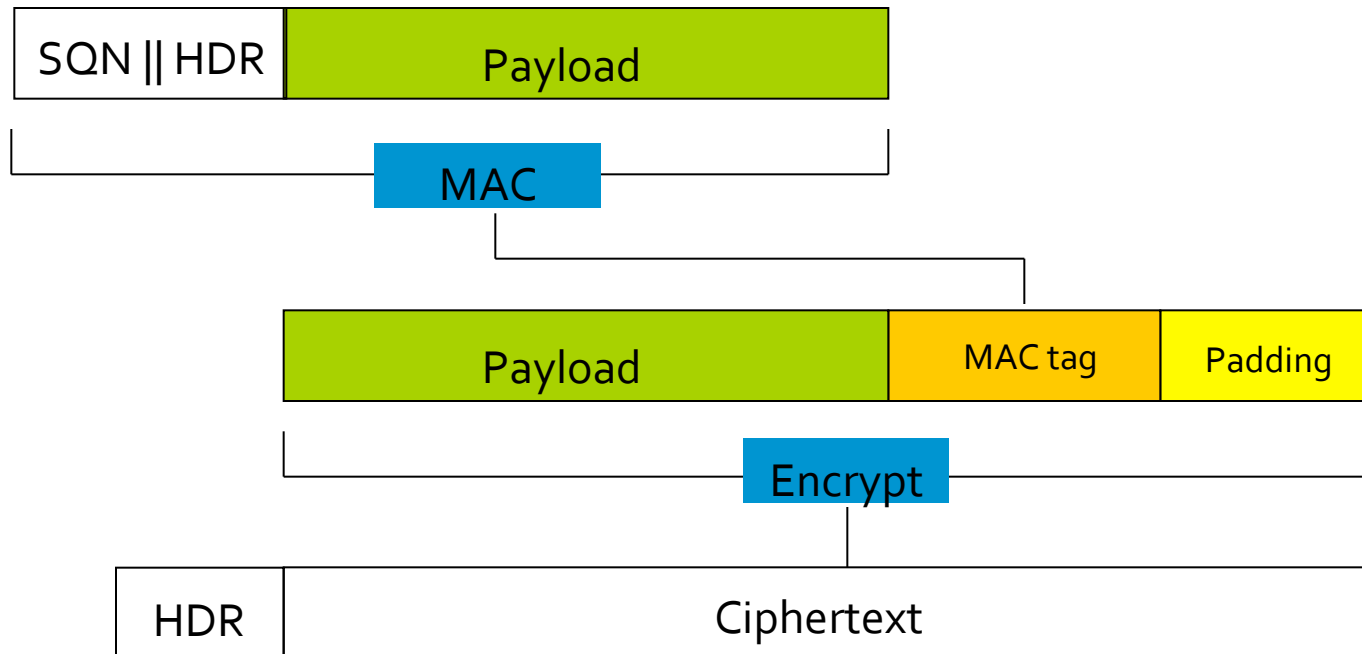
Data origin authentication, integrity using a MAC.

Confidentiality using a symmetric encryption algorithm.

Anti-replay using sequence numbers protected by the MAC.

Optional compression.

Fragmentation of application layer messages.

TLS Record Protocol: MAC-Encode-Encrypt

SQN || HDR	Payload

MAC

Payload	MAC tag	Padding

Encrypt

HDR	Ciphertext

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

Operation of TLS Record Protocol

- Data from layer above is received and partitioned into fragments (max size 2^{14} bytes).

- Optional data compression.

  - Default option is no compression.

- Calculate MAC on sequence number, header fields, and data, and append MAC to data.

- Pad (if needed by encryption mode), then encrypt.

- Prepend 5-byte header, containing:

  - Content type (1 byte, indicating content of record, e.g. handshake message, application message, etc),

  - SSL/TLS version (2 bytes),

  - Length of fragment (2 bytes).

- Submit to TCP.

Uploaded By: anonymous

Operation of TLS Record Protocol

In-bound processing steps reverses these steps:

1. Receive message, of length specified in HDR.

2. Decrypt.

3. Remove padding.

4. Check MAC.

5. (Decompress payload.)

6. Pass payload to upper layer

   (note: no *de*fragmentation; TLS provides a stream of fragments to the application).

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

All of these are fatal errors in TLS.

AEAD and TLS Record Protocol

Dedicated *Authenticated Encryption with Associated Data* (AEAD) algorithms are supported in TLS 1.2, in addition to MEE.

- Single algorithm providing both confidentiality and integrity/data origin (authentication)

- Need not conform to MEE template.

- General AEAD interface specified in RFC 5116.

- AES-GCM specified in RFC 5288.

- AES-CCM specified in RFC 6655.

AEAD and TLS Record Protocol

42.6% of the Alexa top 200k websites support TLS 1.2.

(Up from 17% one year ago and 5% two years ago.)

(source: ssl pulse, Sept. 2014)

TLS 1.2 support in browsers:

Chrome: since release 30.

Firefox: since release 28.

IE: since IE11.

Safari: since iOS5 and OS X 10.9.

(source: wikipedia, Nov. 2013)

But stronger, modern AE designs are not yet in widespread use….

Current AEAD Usage in TLS

Snapshot from ICSI Certificate Notary Project (Sept. 2014):
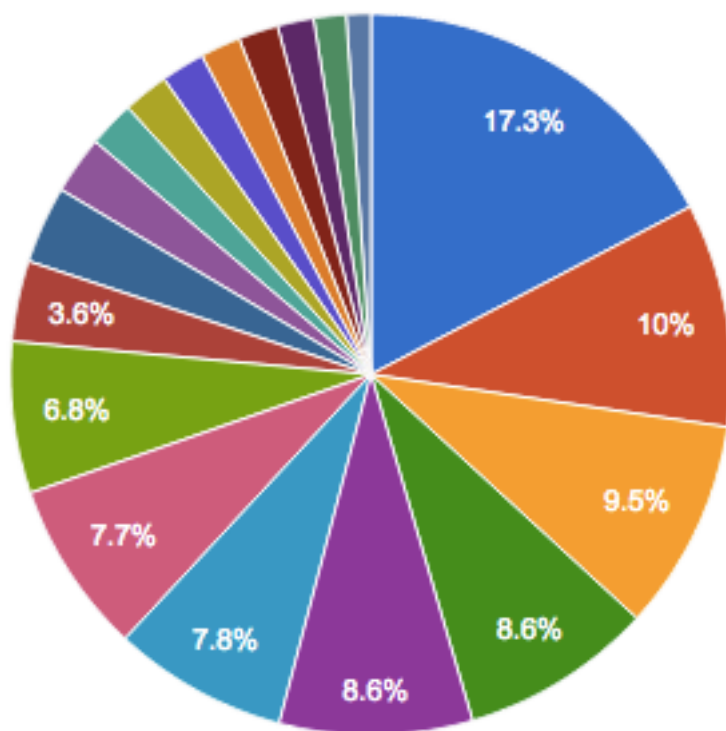
TLS Record Protocol Sequence Numbers

Sequence number is 64 bits in size and is incremented for each new message.

Sequence number not transmitted as part of message.

>> Each end of connection maintains its own view of the current value of the sequence number.
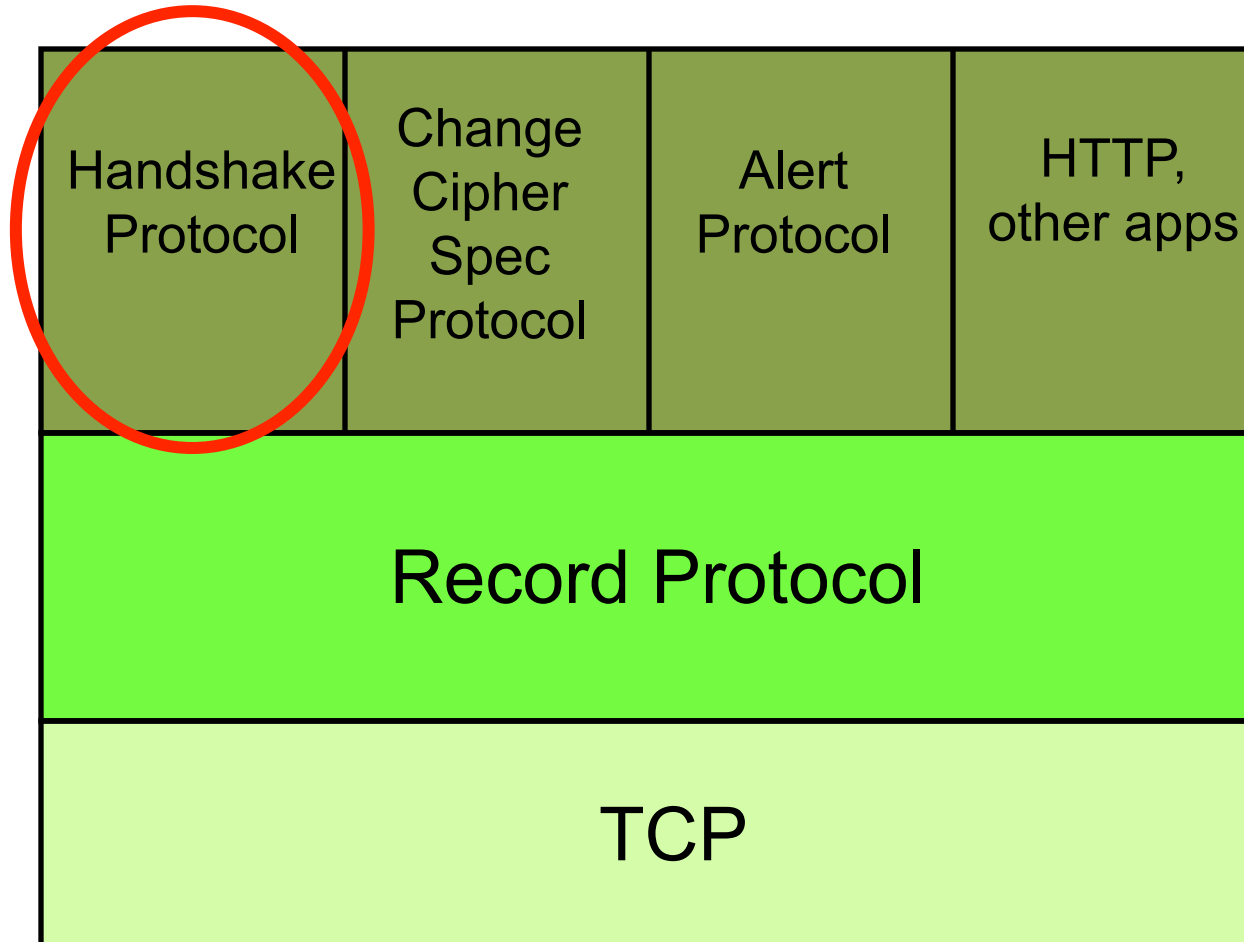
>> TLS is reliant on TCP to deliver messages in order.

Wrong sequence number leads to failure of MAC verification

>> A fatal error leading to TLS connection termination.

Creates stateful encryption scheme.

>> Preventing replay, insertion, reordering attacks,…

TLS Protocol Architecture

TLS Handshake Protocol

TLS consumes symmetric keys:

> MAC and encryption algorithms in Record Protocol.

> Different keys in each direction.

TLS also needs initialization vectors (IVs) for some encryption algorithms.

These keys and IVs are established by the Handshake Protocol and subsequent key derivation.

The TLS Handshake Protocol is itself a complex protocol with many options…

TLS Handshake Protocol Security Goals

Entity authentication of participating parties.

> Participants are called *client* and *server*.

> Reflects typical usage in e-commerce.

> Server nearly always authenticated, client more rarely.

> Appropriate for most e-commerce applications.

Establishment of a fresh, shared secret.

> Shared secret used to derive further keys.

> For confidentiality and authentication/integrity in SSL Record Protocol.

TLS Handshake Protocol Security Goals

Secure negotiation of all cryptographic parameters.

SSL/TLS version number.

Choice of encryption and hash algorithms.

Choice of authentication and key establishment methods.

To prevent version rollback and ciphersuite downgrade attacks.

TLS Handshake Protocol – Key Establishment

TLS supports several different key establishment mechanisms.

Method used is negotiated during the Handshake Protocol itself.

> Client sends list of *ciphersuites* it supports in ClientHello; server selects one and tells client in ServerHello.
>
> e.g. TLS_RSA_WITH_AES_256_CBC_SHA256 or TLS_KRB5_WITH_3DES_EDE_CBC_MD5

Very common choice is RSA encryption.

> Client chooses `pre_master_secret`, encrypts using public RSA key of server, sends to server.
>
> RSA encryption based on PKCS#1v1.5 padding method.
>
> Source of Bleichenbacher attack and much sadness.

TLS Handshake Protocol – Key Establishment

Can also create `pre_master_secret` from:

Static Diffie-Hellman

Server certificate contains DH parameters (group, generator g) and static DH value g^x.

Client chooses y, computes g^y and sends to server.

pre_master_secret = g^{xy}.

Ephemeral Diffie-Hellman

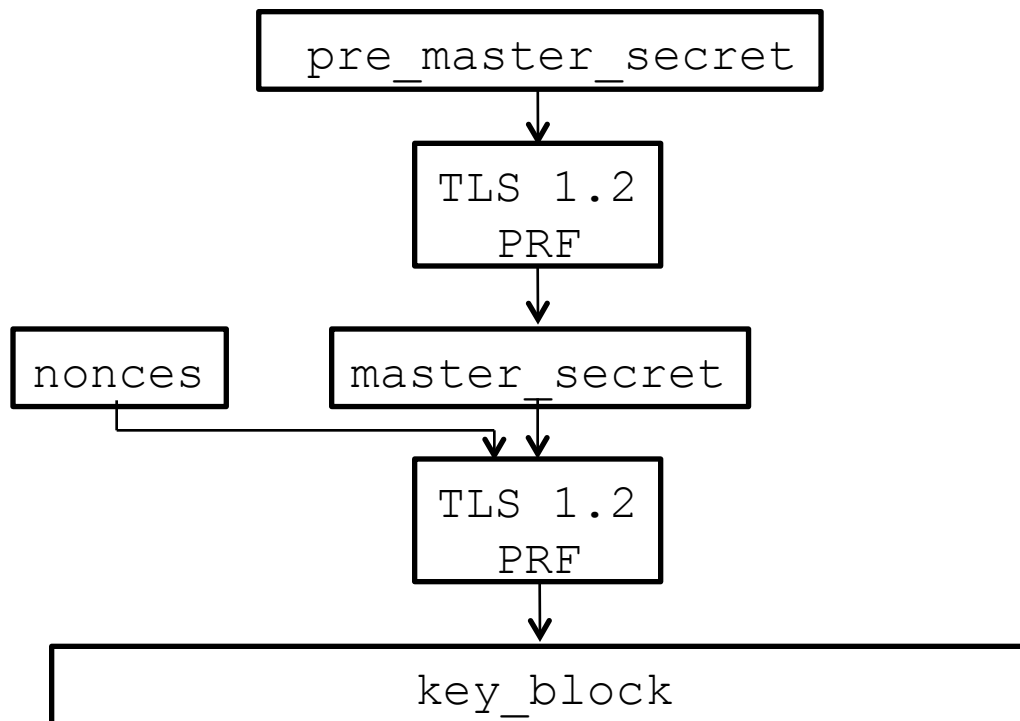Server and Client exchange fresh Diffie-Hellman components in group chosen by server.

Signed (usually only by server) to provide authentication.

Anonymous Diffie-Hellman

Each side sends Diffie-Hellman values in group chosen by server, but no authentication of these values.

Vulnerable to man-in-middle attacks.

TLS Key Derivation (in TLS 1.2)

```
          ┌─────────────────────────┐
          │    pre_master_secret     │
          └─────────────────────────┘
                      │
                      ▼
              ┌──────────────┐
              │   TLS 1.2    │
              │     PRF      │
              └──────────────┘
                      │
                      ▼
┌──────────┐  ┌──────────────────┐
│  nonces  │  │  master_secret   │
└──────────┘  └──────────────────┘
     │                 │
     └────────┐  ┌──────┘
              ▼  ▼
        ┌──────────────┐
        │   TLS 1.2    │
        │     PRF      │
        └──────────────┘
              │
              ▼
┌───────────────────────────────────┐
│            key_block              │
└───────────────────────────────────┘
```

STUDENTS-HUB.com

TLS Key Derivation

Keys used by MAC and encryption algorithms in the Record Protocol are derived from `pre_master_secret`:

Derive `master_secret` from `pre_master_secret` using TLS PRF.

PRF used can be negotiated during the Handshake Protocol (TLS1.2).

Default PRF for TLS1.2 is built by iterating HMAC-SHA256 in a specified way.

Previous versions use an *ad hoc* construction based on MD5 and SHA1.

Derive `key_block` from `master_secret` and client/server nonces exchanged during Handshake Protocol.

Again using the TLS PRF in TLS1.2.

Split up `key_block` into MAC keys, encryption keys and IVs for use in Record Protocol as needed.

TLS Handshake Protocol – Entity Authentication

TLS supports several different entity authentication mechanisms for clients and servers.

Method used is negotiated along with key exchange method during the Handshake Protocol itself.

> Specified in ciphersuites.

Most common server authentication method is based on RSA.

> Ability of server to decrypt `pre_master_secret` using its private key and then generate correct PRF value in `finished` message using key derived from `pre_master_secret` authenticates server to client.

Authentication Based on RSA Encryption (simplified)

Client   **Server**

ClientNonce $\longrightarrow$

$\longleftarrow$ ServerNonce, ServerCert

```
1.Check ServerCert
2.Extract PK from ServerCert
3.Select random pms
4.Compute Enc_PK(pms)
```

$Enc_{PK}(pms)$ $\longrightarrow$

```
1.Decrypt and extract pms
2.Derive ms from pms and
    nonces
3.Compute ServerFin =
    PRF(ms,transcript)
```

$\longleftarrow$ ServerFin

```
Check correctness
of ServerFin
```

TLS Handshake Protocol – Entity Authentication

Less common authentication methods:

- Ability of server to derive key from server's static (private) DH value (in server certificate) and client's ephemeral (public) DH value.

- ECDSA, DSA or RSA signatures on nonces (and other fields, e.g. Diffie-Hellman values).

- Pre-shared key.

- Shared password.

- …

TLS Handshake Protocol Overview

M1: C → S: `ClientHello`

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

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

M4: S → C: `ChangeCipherSpec, ServerFinished`

`[]` denotes optional/situation-dependent field.

(`ChangeCipherSpec` messages are technically not part of Handshake Protocol.)

TLS Handshake Protocol – Additional Features

- TLS Handshake Protocol supports ciphersuite renegotiation and session resumption.

- Ciphersuite renegotiation allows re-keying and change of ciphersuite during a session.

    - E.g., to force strong client-side authentication before access to a particular resource on the server is allowed.

    - Initiated by client sending `ClientHello` or server sending `ServerHelloRequest` over existing Record Protocol.

    - Followed by full run of Handshake Protocol.

- Session resumption allows authentication and shared secrets to be reused across multiple connections in a single session.

    - E.g., allows fetching next web-page from same website without re-doing full, expensive Handshake Protocol.

TLS Handshake Protocol – Session Resumption

Client and server run lightweight version of Handshake Protocol:

1.   C → S: `ClientHello`

     (quoting existing `SessionID`, new `ClientNonce` and list of ciphersuites).

2.   S → C: `ServerHello`

     (repeating `SessionID`, new `ServerNonce` and selected ciphersuite),

     `ChangeCipherSpec`, `Finished`.

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

- New `key_block` is derived by each side after receipt of `ChangeCipherSpec` messages.
- New keys and IVs are derived from newly-exchanged nonces and existing `master_secret`.
- The session resumption exchange is protected by existing Record Protocol ciphersuite.
- No public key operations are involved in session resumption.

TLS Sessions and Connections

Session concept:

- Sessions are created by the Handshake Protocol.

- Session state defined by session ID and set of cryptographic parameters (encryption and hash algorithm, master secret, certificates) negotiated in Handshake Protocol.

- Each session can carry multiple *sequential* connections.

Connection concept:

- Keys for multiple sequential connections are derived from a single `master_secret` created during one run of the full Handshake Protocol.

- Session resumption Handshake Protocol runs exchange new nonces.

- These nonces are combined with existing `master_secret` to derive `key_block` and keys for each new connection.

TLS Protocol Architecture

Other TLS Protocols

Alert protocol.

> Management of SSL/TLS connections and sessions, error messages.

> Fatal errors and warnings.

> Defined actions to ensure clean session termination by both client and server.

Change cipher spec protocol.

> Technically not part of Handshake Protocol.

> Used to indicate that entity is changing to recently agreed ciphersuite.

Both protocols run over Record Protocol (so are peers of Handshake Protocol).

TLS Handshake Complexity

- Recall simplistic view of TLS:

    - Handshake Protocol followed by Record Protocol.

- Reality is much more complex:

    - Initial Handshake Protocol over Record Protocol with no keys.

    - Change Cipher Spec. Protocol message, switch on new keys.

    - Completion of Handshake via exchange of `Finished` messages, now running over keyed Record Protocol.

    - Followed by arbitrary sequences of Session Resumption and Renegotiation runs.

    - Most of this activity is hidden from applications.

- This complexity has turned out to have serious negative consequences:

    - Ray-Rex-Dispensa Renegotiation Attack (2009).

    - Triple Handshake Attack (2014).

    - Considerable barrier to formal security analysis.

SSL and TLS

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

- TLS signalled by version number 3.1.

- Use of HMAC for MAC algorithm in TLS 1.0.

- Different method for deriving keying material

  - TLS 1.0 uses PRF based on HMAC with MD5 and SHA-1 operating in combination.

- Additional alert codes.

- More client certificate types.

- Variable length padding.

  - Can be used to hide lengths of short messages and so limit traffic analysis.

Evolution of TLS

TLS 1.1 (RFC 4346, 2006) obsoletes TLS 1.0 (RFC 2246).

- Uses explicit IVs instead of IV chaining to prevent attacks based on predictable IVs (see later).

- Attempts to protect against padding oracle attacks (see later).

Evolution of TLS

TLS 1.2 (RFC 5246) published in 2008 obsoletes TLS 1.1 (RFC 4346).

- Removal of dependence on MD-5 and SHA-1 hash algorithms for PRFs.

- Now negotiable in Handshake Protocol, but specific PRF based on HMAC-SHA256 as standard.

- Support for AEAD modes.

- Removed support for some cipher suites.

Adoption of TLS 1.1 and 1.2 has grown rapidly over the last 18 months.

- Largely in response to the spate of recent attacks.

TLS 1.3 now under development in IETF TLS Working Group.

TLS Extensions

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

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.

  - To manage fragmentation of long messages over multiple UDP packets.

- And changes to the Record Protocol.

  - Allow out-of-order delivery of messages by using explicit sequence numbers and sliding windows.

  - Do not treat decryption errors as fatal.

  - And do not send alerts when decryption fails.

BEAST and CRIME Attacks

BEAST

IV chaining in SSLv3 and TLS 1.0 CBC mode 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.

- BEAST = Browser Exploit Against SSL/TLS

- 16-year demonstration that attacks do get better with time.

CBC Mode Reminder
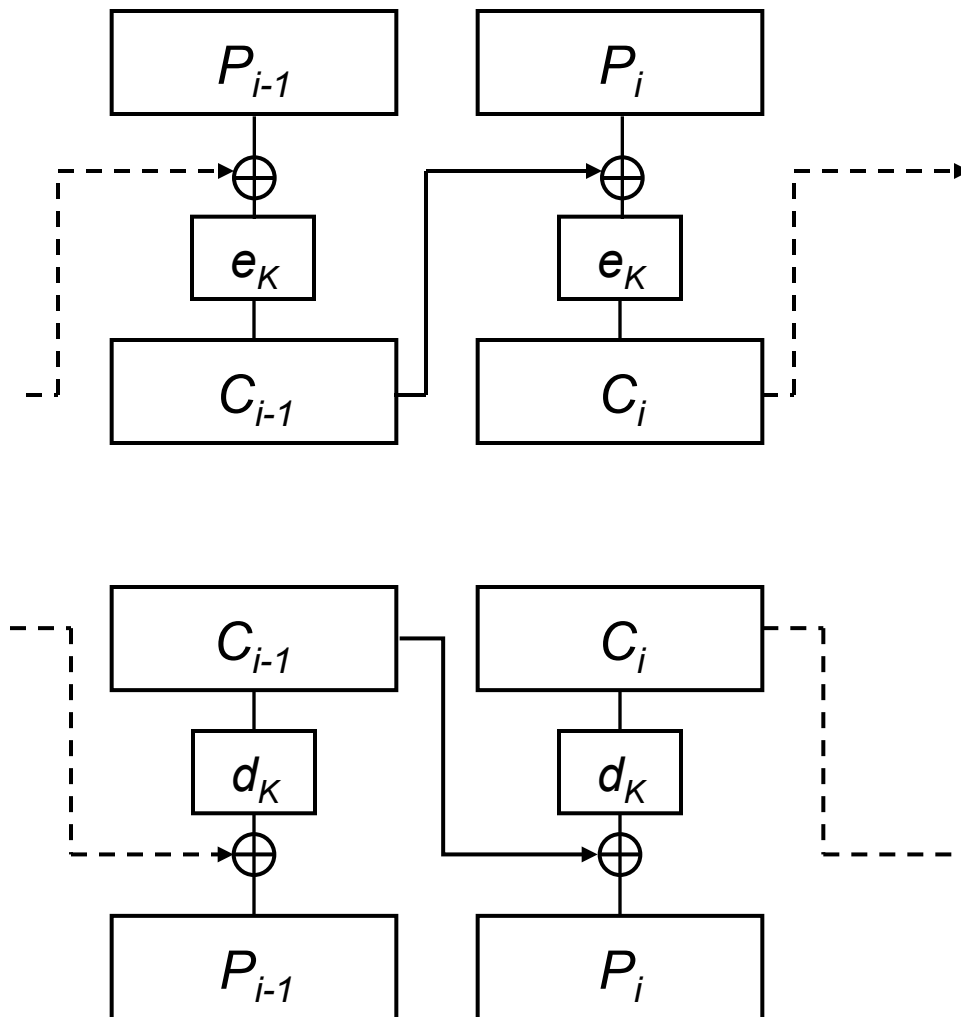
Initialisation Vector (IV):

- Defines C_o for processing first block.
- Chained IVs are common in applications, and in SSL 3.0 and TLS 1.0 in particular.
- TLS 1.1 and 1.2 require to be random.

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

- More on padding later.

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 where some known, previous block C_0 was used as the IV.
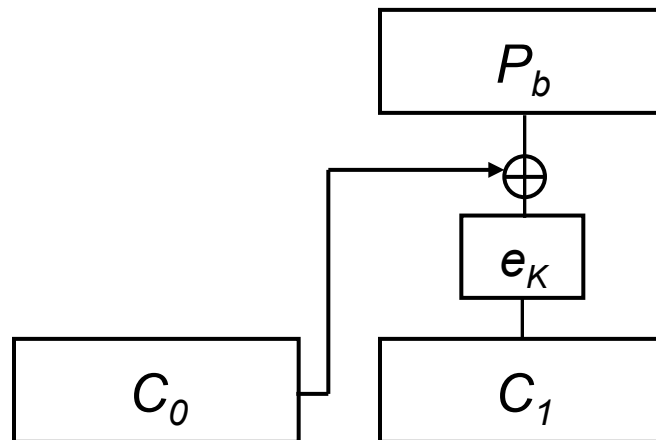
Attacking Chained IVs

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

$$\boxed{P_b} \qquad \boxed{P_0 \oplus C_0 \oplus C_1}$$
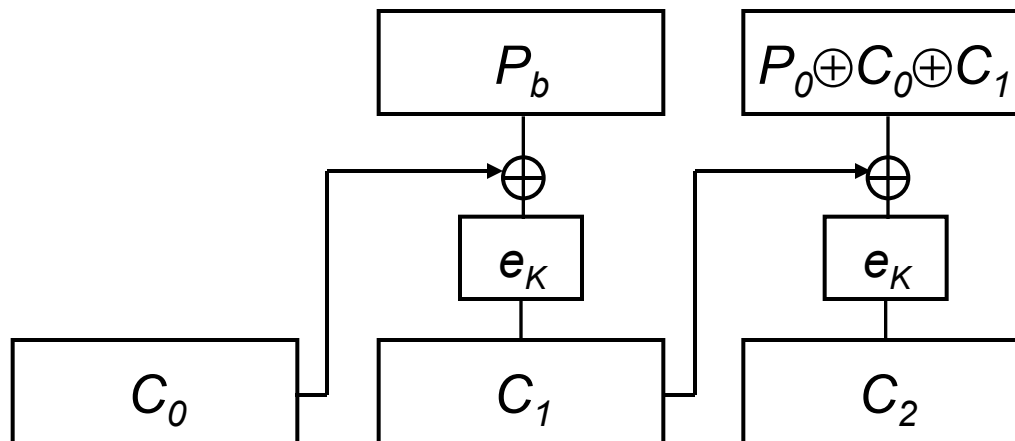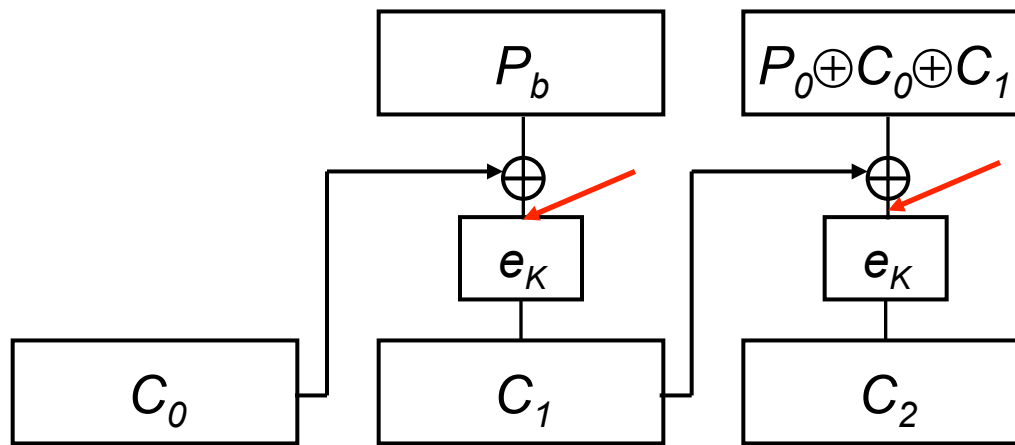
- If $P_b = P_o$, then inputs to block cipher are the same in both encryptions.

- Hence, if $P_b = P_o$, 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_o 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
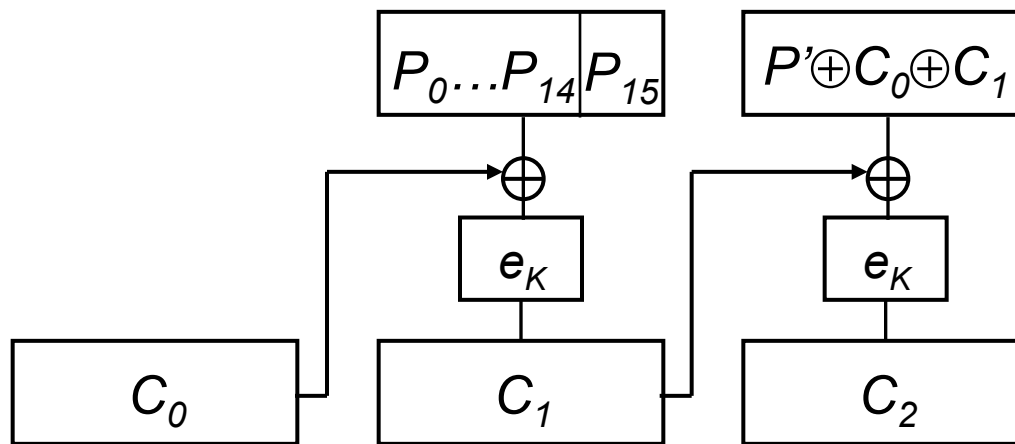
- Assume bytes P_0, P_1,... P_{14} are known, try to recover P_{15}.
- Use $P_0 P_1 ... P_{14}$ as first 15 bytes of P'.
- Iterate over 256 possible values in P'_{15}.
- $P'_{15} = P_{15}$ if and only if $C_1 = C_2$.
- So average of 128 trials (each involving one chosen plaintext) to extract P_{15} when remaining bytes in block are known.

The BEAST – Part 2
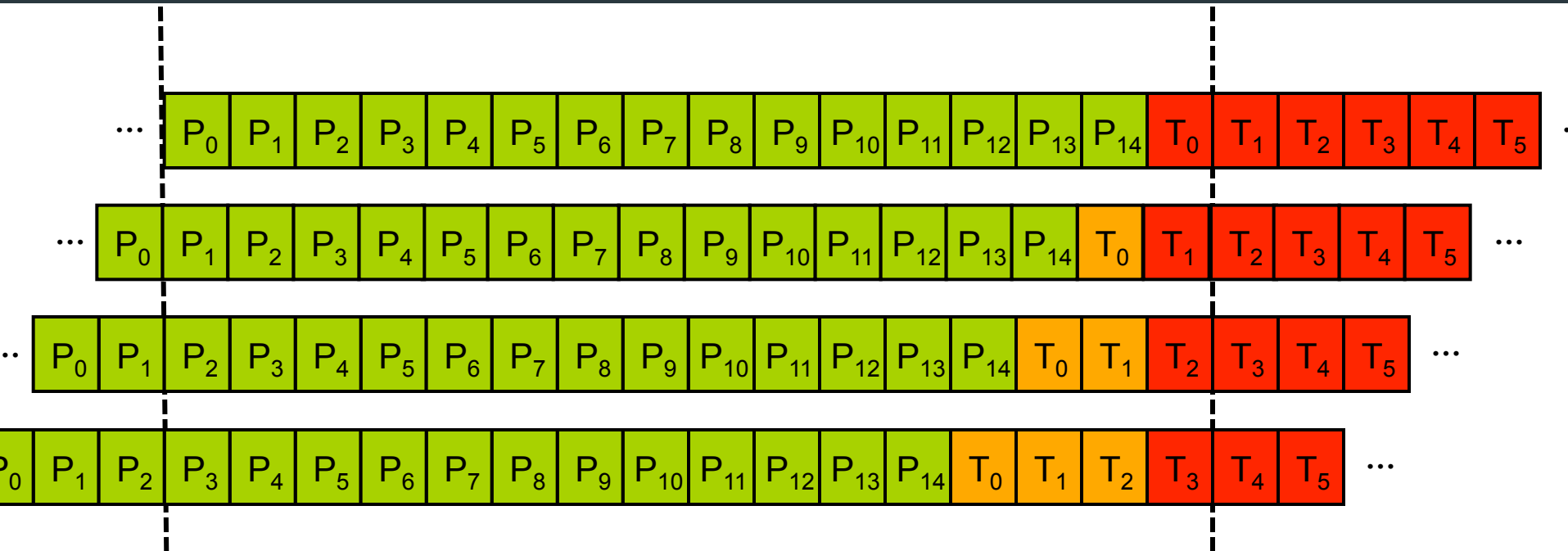
Now assume attacker can control position of unknown bytes in stream with respect to CBC block boundaries (chosen boundary privilege).

Repeat previous single-byte recovery attack with sliding bytes.

    Green: initially known bytes.

    Red: unknown (target) bytes.

    Orange: recovered bytes.

Cookie for remote site

Browser

TLS tunnel
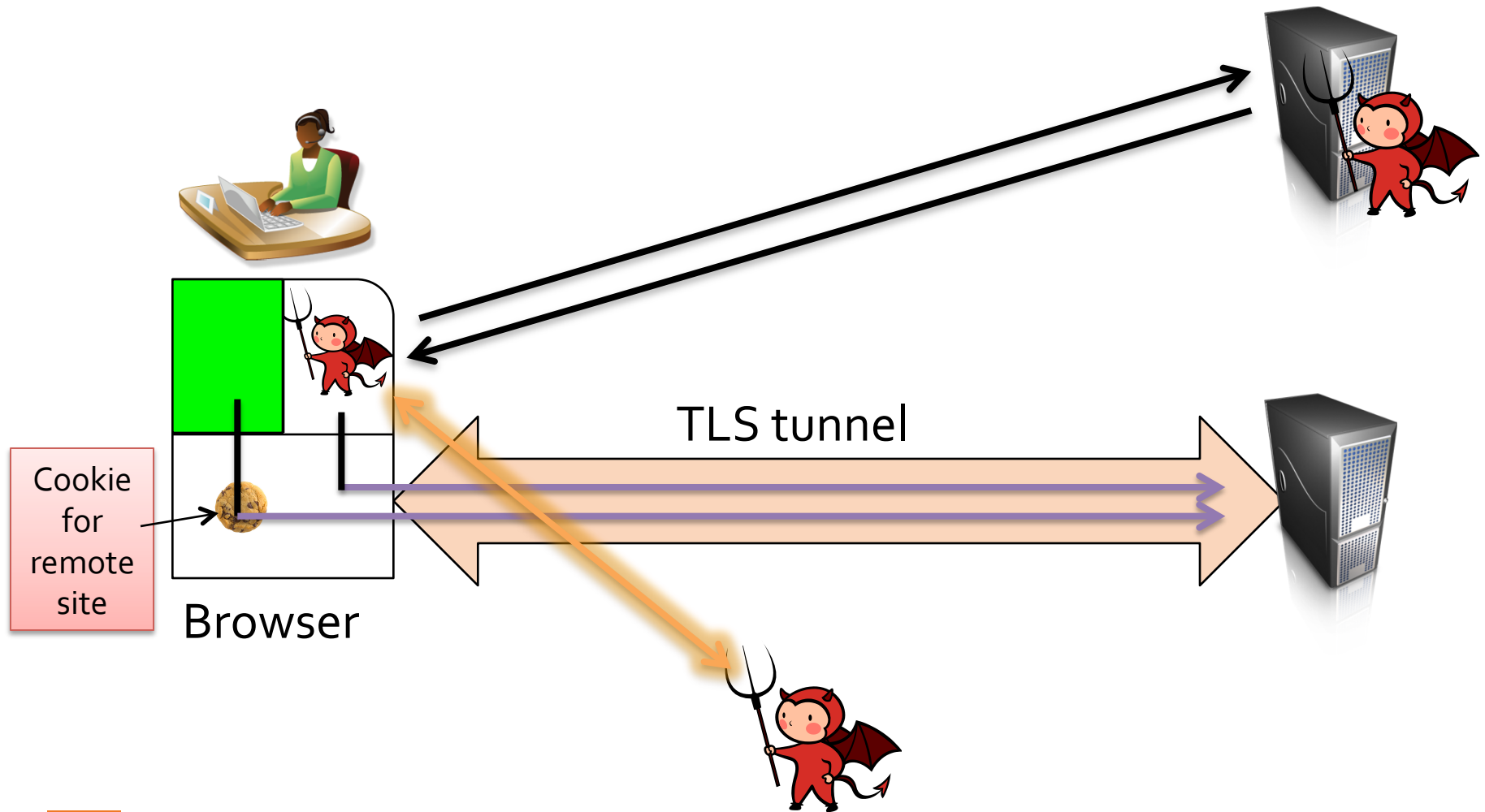
BEAST – Key Features

- 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.

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.

BEAST – 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 in 2011.

Tools from the wider security field were needed to make the attacks headline news.

> Man-in-the-browser via Javascript.
>
> Fair game given the huge range of ways in which TLS get used.
>
> Maybe those tools can be used elsewhere?

CRIME

Exploits use of optional compression in TLS.

Theoretical attack known since 2004, made practical by Duong and Rizzo in 2012.

Idea:

Plaintext length leaks through ciphertext length.

But plaintext length leaks amount of compression.

And amount of compression leaks a tiny amount about plaintext.

Recovery of HTTP session cookies (and more).

Mitigated by switching off TLS compression.

Application layer compression still problematic.

CRIME – Lessons

A theoretical vulnerability pointed out in 2004 became a practical attack in 2012.

Attacks *really* do get better with time.

Tools developed for BEAST *were* reused in committing CRIME.

And maybe they can be used yet elsewhere?

Real World Cryptography 2015
London, UK, 7-9 January 2015
http://www.realworldcrypto.com/rwc2015
#realworldcrypto

Speakers to include:

Elena Andreeva (K.U. Leuven)

Dan Bogdanov (Cybernetica)

Sasha Boldyreva (Georgia Tech)

Claudia Diaz (K.U. Leuven)

Roger Dingledine (Tor project)

Ian Goldberg (U. Waterloo)

Arvind Mani (LinkedIn)

Luther Martin (Voltage Security)

Elisabeth Oswald (U. Bristol)

Scott Renfro (Facebook)

Ahmad Sadeghi (TU Darmstadt)

Elaine Shi (UMD)

Brian Sniffen (Akamai)

Nick Sullivan (CloudFlare)

Padding Oracle Attacks

TLS Record Protocol: MAC-Encode-Encrypt
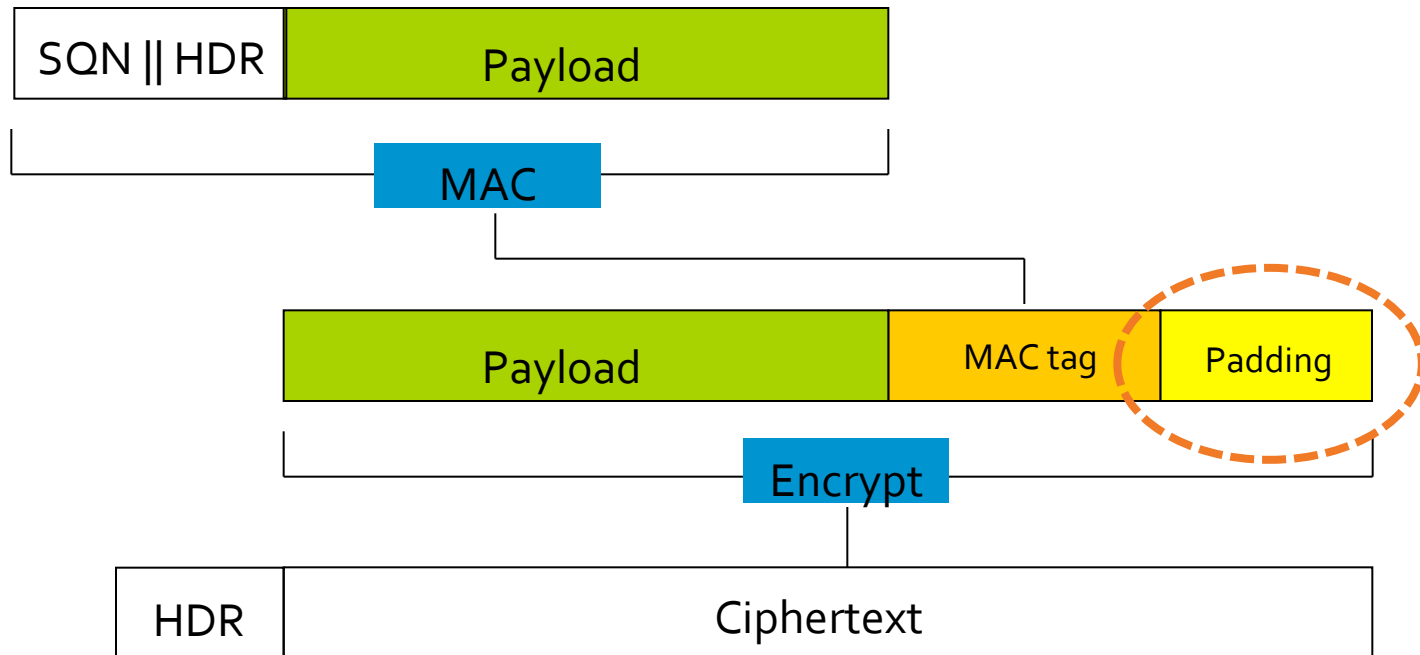
SQN || HDR	Payload

MAC

Payload	MAC tag	Padding

Encrypt

HDR	Ciphertext

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"

66

TLS Record Protocol Padding

Padding in TLS 1.0 and up has a particular format:

- Always add at least 1 byte of padding.

- If t bytes are needed, then add t copies of the byte representation of t-1.

- So possible padding patterns in TLS are:

  00;

  01 01;

  02 02 02;

TLS Record Protocol Padding

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

- Up to 256 bytes of padding in total:
  FF FF…. FF

- From TLS 1.0:

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

Handling Padding During Decryption

- TLS 1.0 error alert:

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

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

Uploaded By: anonymous

Exploiting Weak Padding Checks – Moeller Attack

TLS 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 special TLS ciphertext containing a complete block of padding.

- So MAC ends on block boundary for this ciphertext.

- Padding is removed by inspecting last byte only.

Blocks from special ciphertext

C_{t-2} ... C_{t-1} ... C_t

d_K d_K d_K

Byte value is "0F" here

Moeller Attack

Blocks from special ciphertext

Target ciphertext block from stream

C_{t-2} ... C_{t-1} C^*

d_K d_K d_K

Enabling recovery of last byte of $d_K(C^*)$ here.

Decryption succeeds if and only if byte value is "0F" here
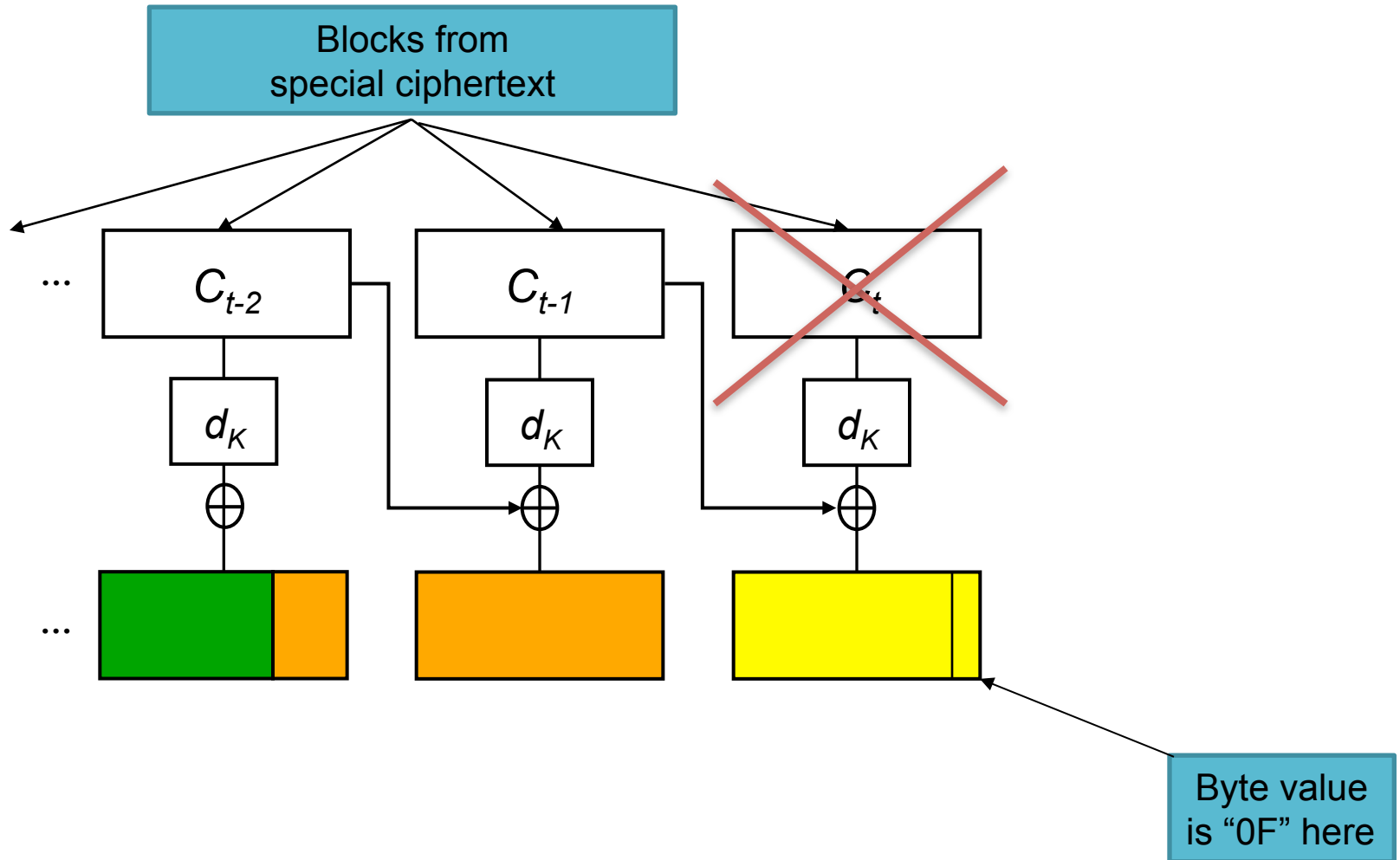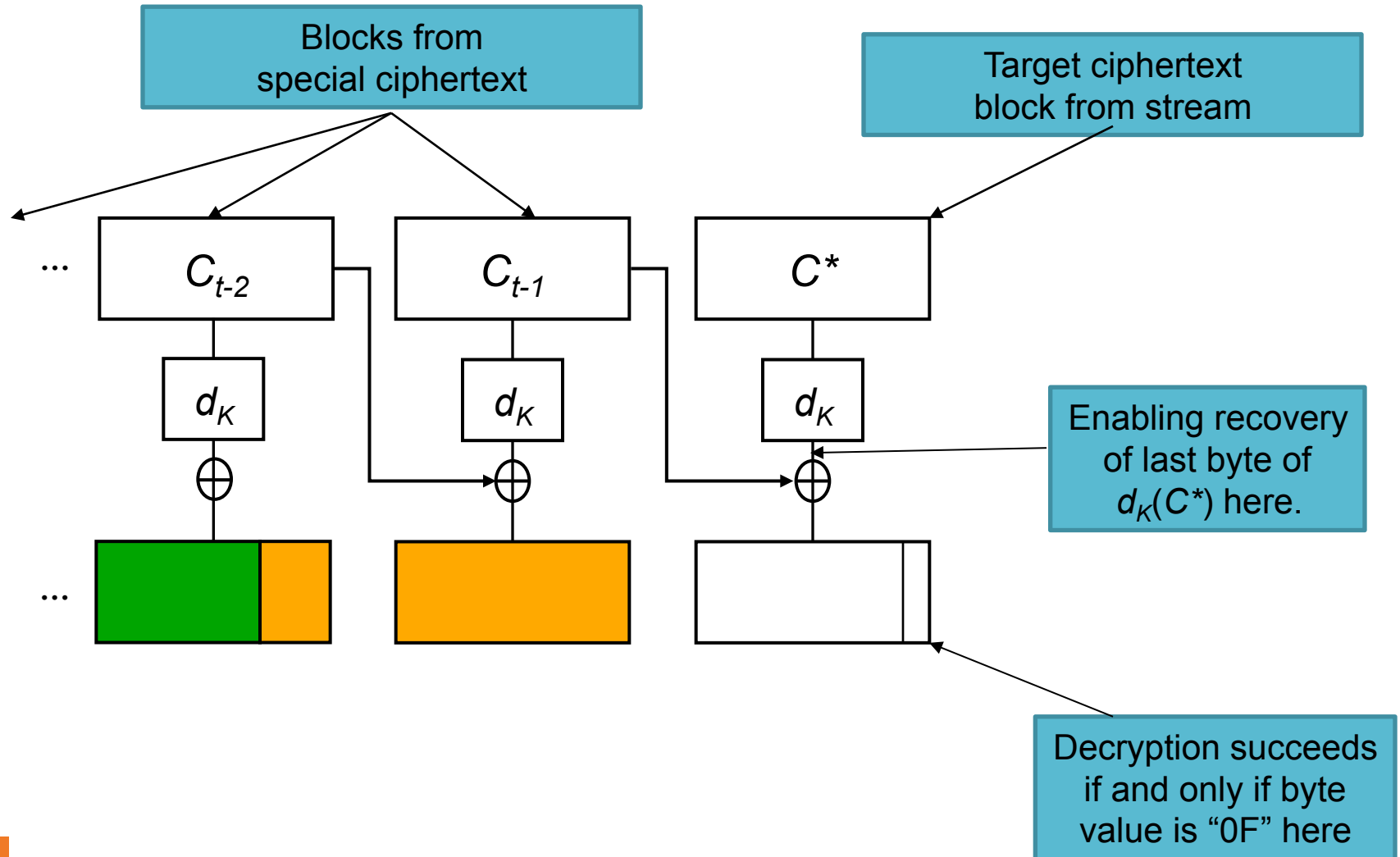
Moeller Attack

- Decryption succeeds if and only if:

$$C_{t-1} \oplus d_K(C^*) = (??,??,....??, 0F)$$

- 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, by solving system of eqns:

$$C_{t-1} \oplus d_K(C^*) = (??,??,....??,0F)$$

$$C^*_{-1} \oplus d_K(C^*) = P^*$$

  where C^*_{-1} is the block preceding C^* in the 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

- Henceforth, we suppose that TLS does a *full* padding check.

- So decryption checks that bytes at the end of the plaintext have one of the following formats:

<div align="center">

00;

01, 01;

02, 02, 02;

....

FF, FF,...........FF;

</div>

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

- NB Other "sanity" checks may also be needed during decryption.

Padding Oracles

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

Padding
Oracle

$P=Dec_K(C)$

Check
padding of P

C

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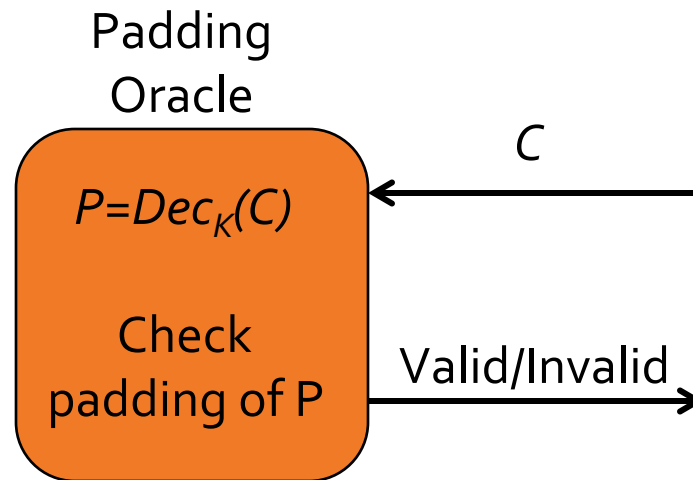
- Vaudenay showed that, for CBC mode and for certain padding schemes, a padding oracle can be used to build a decryption oracle!

XOR with various Δ here and submit to padding oracle

C_{t-1}

C_t

d_K

d_K

P_{t-1}

P_t

Recovering true plaintext byte via $P_t \oplus \Delta = (\ldots 00)$

Eventually produces valid pad "00" here

Padding Oracle Attack for TLS Padding

XOR with $\Delta_1\Delta_0$ here and submit to oracle

C_{t-1}

C_t

d_K

d_K

P_{t-1}

P_t

This byte now set to "01" by setting $\Delta_0 := \Delta \oplus 01$

Recovering true plaintext byte via $P_t \oplus (\ldots\Delta_1\Delta_0) = (\ldots 0101)$

Eventually produces valid pattern "01 01" here
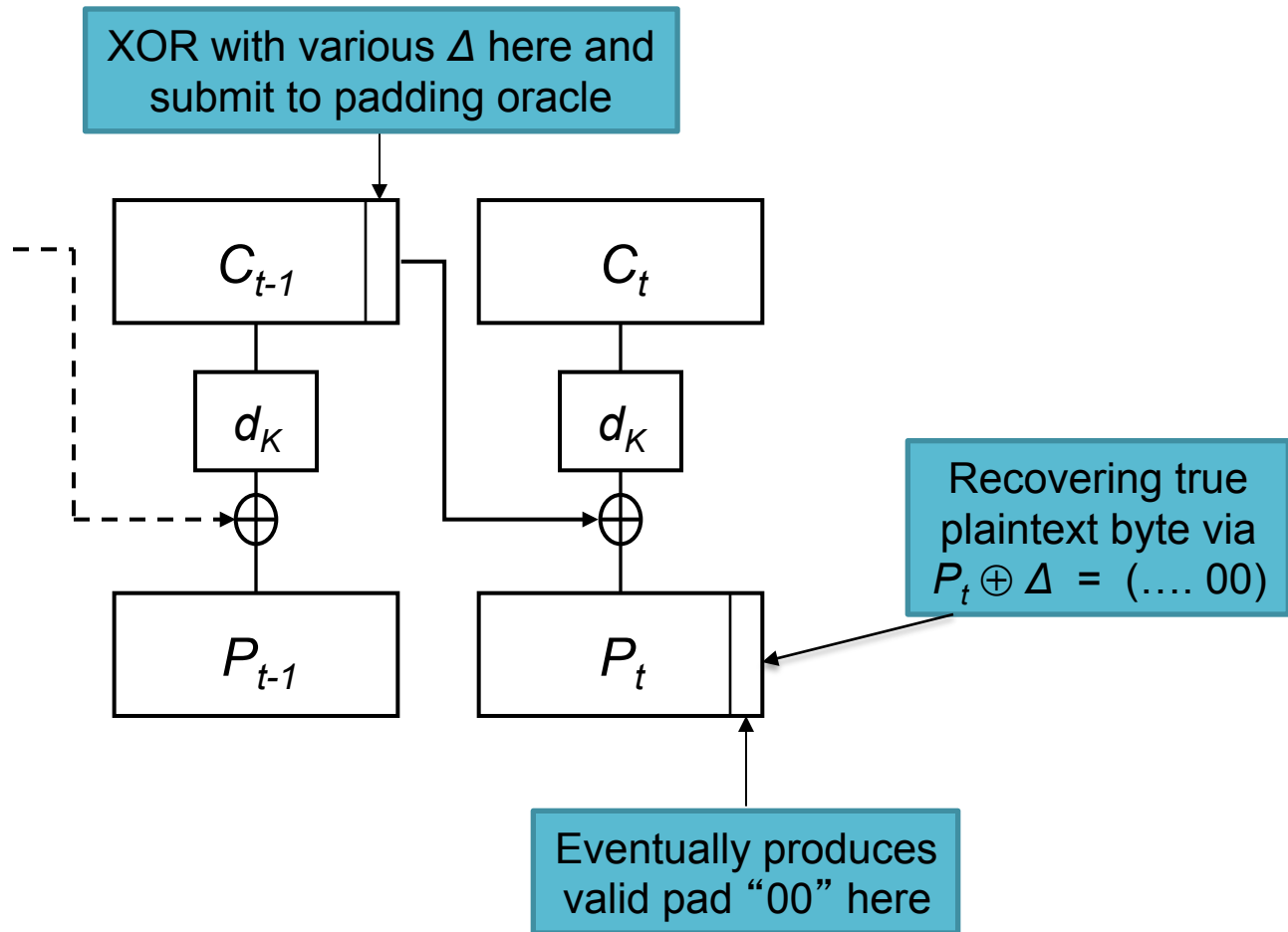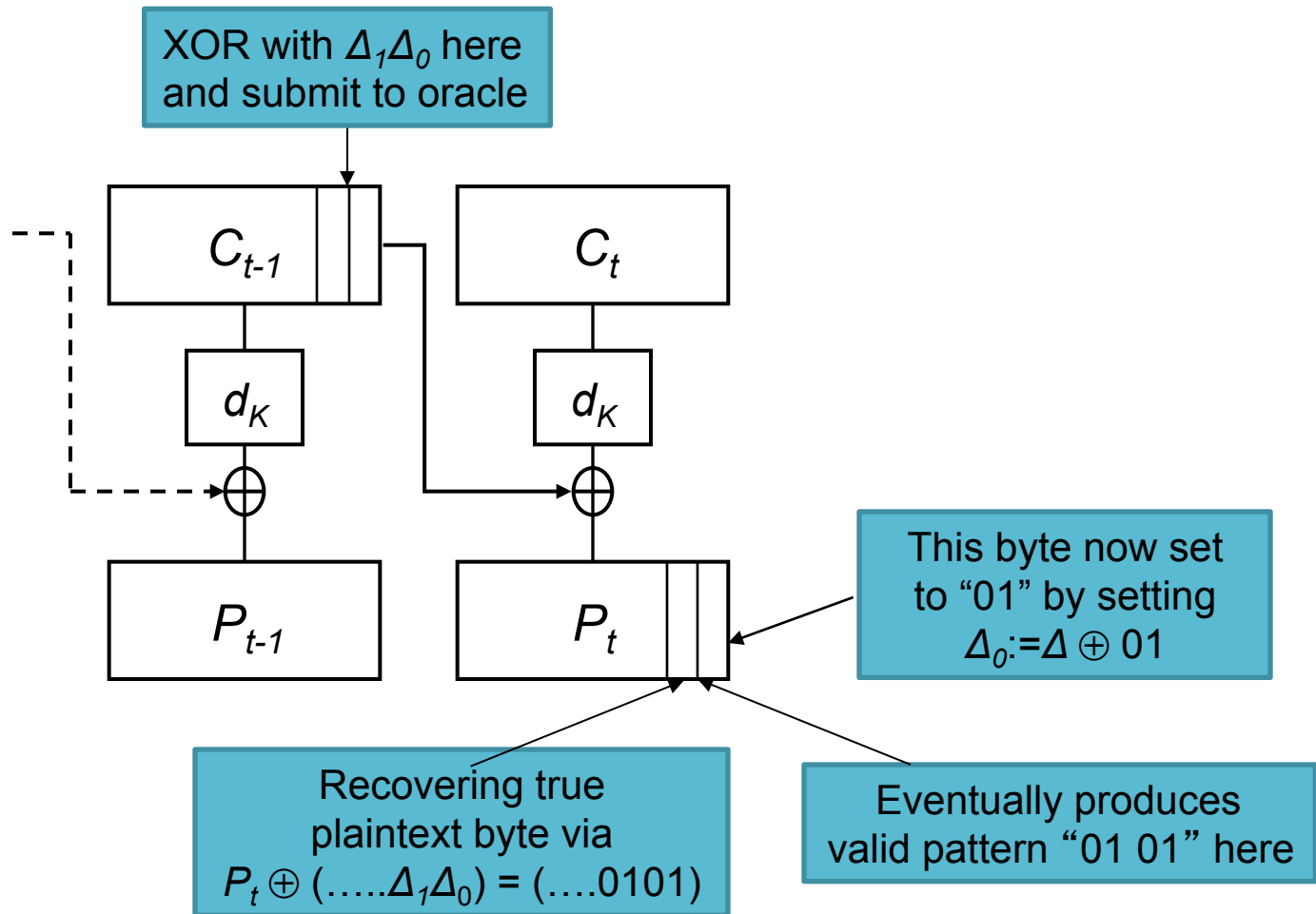
Padding Oracle Attack for TLS Padding

- An average of 128 trials are needed to extract the last byte of each plaintext block.

- Can extend to the entire block, with an average of 128 trials per byte.

- Can extend to the entire ciphertext.

  - Because attacker can place *any* target block as last block of ciphertext.

TLS Padding Oracles In Practice?

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

- Vaudenay's padding oracle attack will produce an error of one type or the other.

  - Padding failure indicates *incorrect* padding.

  - MAC failure indicates *correct* padding.

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

TLS Padding Oracles In Practice?

Good news (for the attacker):

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

    - `bad_record_mac`

    - `decryption_failed`

Bad news:

- But the error messages are encrypted, so cannot be seen by the attacker.

- And an error of either type is *fatal*, leading to immediate termination of the TLS session.

TLS Padding Oracles In Practice?

Canvel *et al.* [CHVV03] :

- A MAC failure error message will appear on the network **later** than a padding failure error message.

- Because an implementation would only bother to check the MAC if the padding is good.

- So *timing* the appearance of error messages might give us the required padding oracle.

  - Even if the error messages are encrypted!

  - Amplify the timing difference by using long messages.

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

OpenSSL and Padding Oracles

Canvel *et al.* [CHVV03]:

- The attacker can still decrypt reliably if a *fixed* plaintext is repeated in a *fixed* location across many TLS sessions.

    - e.g. password in login protocol or an HTTP session cookie.

    - A multi-session attack.

    - Modern viewpoint: use BEAST-style malware to generate the required encryptions.

- The OpenSSL implementation had a detectable timing difference.

    - Roughly 2ms difference for long messages (close to 2^{14} byte maximum).

    - Enabling recovery of TLS-protected Outlook passwords in about 3 hours.

Multi-session Version

XOR with various Δ here and submit for decryption

C_{t-1}, C_t vary in each trial, but P_t is fixed

C_{t-1}

C_t

d_K

d_K

P_{t-1}

P_t

Recovering true plaintext byte via $P_t \oplus \Delta = (\dots 00)$

Eventually produces valid pad "00" here
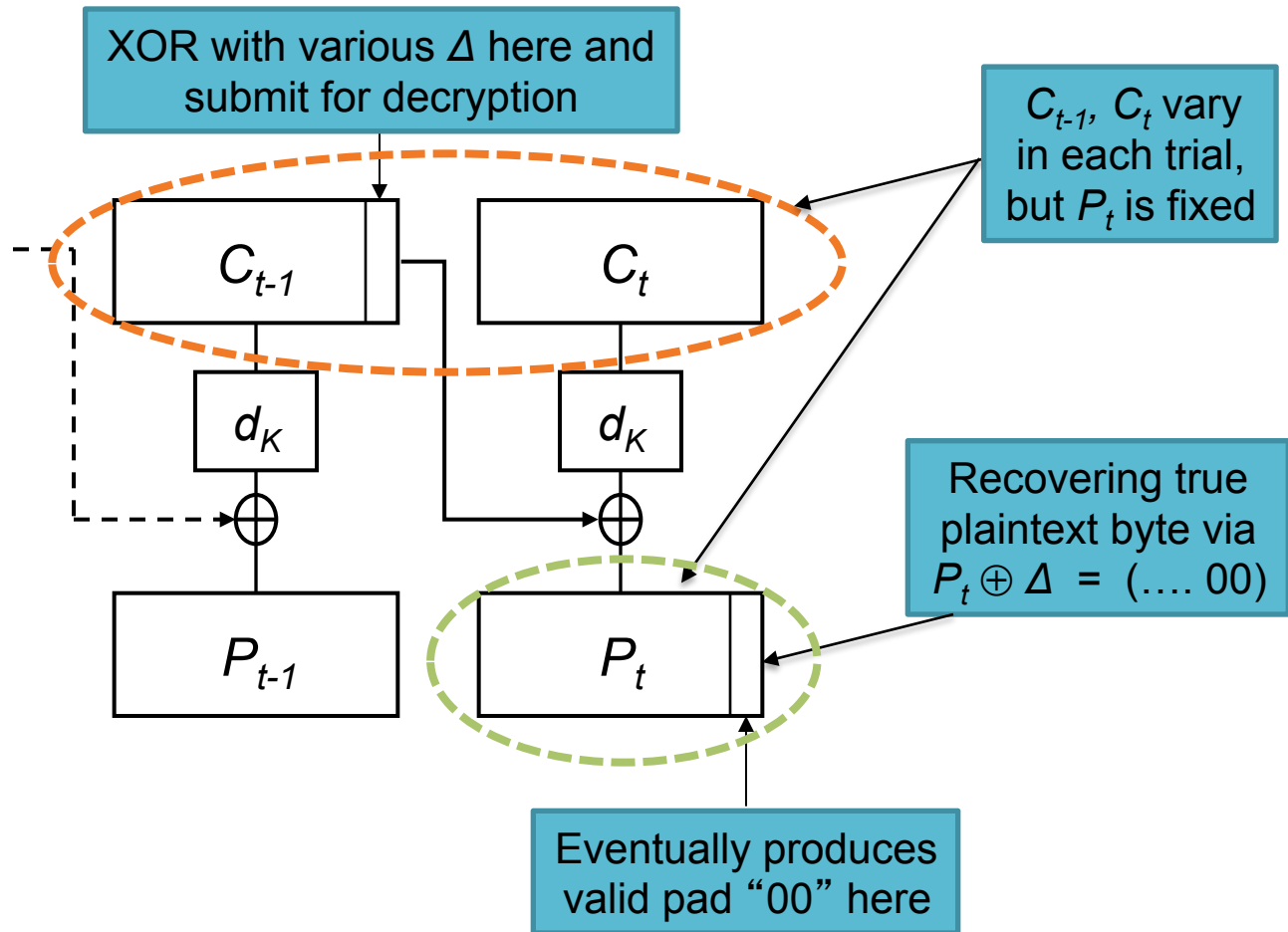
Padding Oracle Attack Countermeasures?

- ## Redesign TLS:

    - Pad-MAC-Encrypt or Pad-Encrypt-MAC.

    - Too invasive, did not happen.

- ## Switch to RC4?

- ## Or add a fix to ensure uniform errors:

    - Check the MAC anyway, even if the padding is bad.

    - If attacker can't tell difference between MAC and pad errors, then maybe TLS's MEE construction is secure?

    - Fix included in TLS 1.1 and 1.2 specifications.

Padding Oracles Post [CHVV03]

- Regarded as a solved problem for SSL/TLS.

- Steady trickle of other papers showing padding oracle attacks in other systems, e.g.

    - ISO standard for CBC-mode encryption [PY04, YPM05].

    - Trailer oracle attack on IPsec [DP07, DP10]

    - POET attack on ASP.NET [DR10].

    - XML encryption [JS11].

    - OpenSSL implementation of DTLS [AP12].

Breaking DTLS in OpenSSL

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

But surely all DTLS implementations would have learned the 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

- The OpenSSL implementations of DTLS prior to versions 0.9.8s/1.0.of did *not* check the MAC if the padding check fails.

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

- Moreover, DTLS does not treat MAC/padding errors as fatal, so a single session attack might be possible!

Breaking DTLS in OpenSSL

Bad news: no error messages to time.

- Not a major hurdle:

```
┌──────────┬──────────┐
│  Attack  │ Heartbeat│ ──────→
│  packet  │  packet  │
└──────────┴──────────┘

        ┌──────────┐
   ←──── │ Heartbeat│
        │ response │
        └──────────┘
```
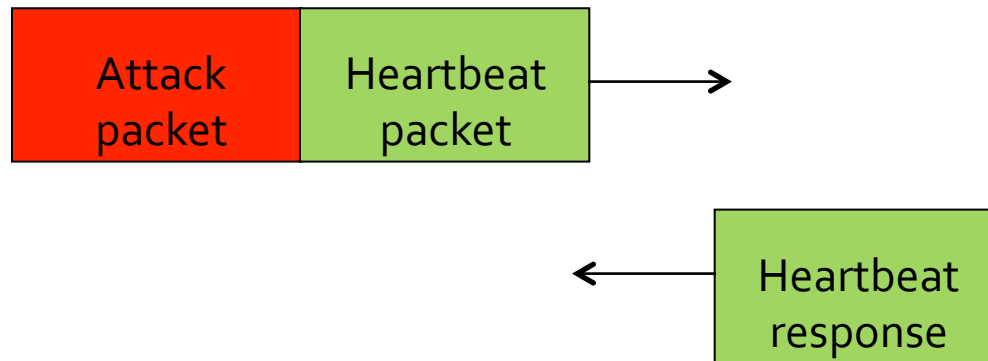
- Attack packet takes longer to process if padding is good.

- 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*.

| Attack packet | Attack packet | Attack packet | Heartbeat packet | → |

← | Heartbeat response |
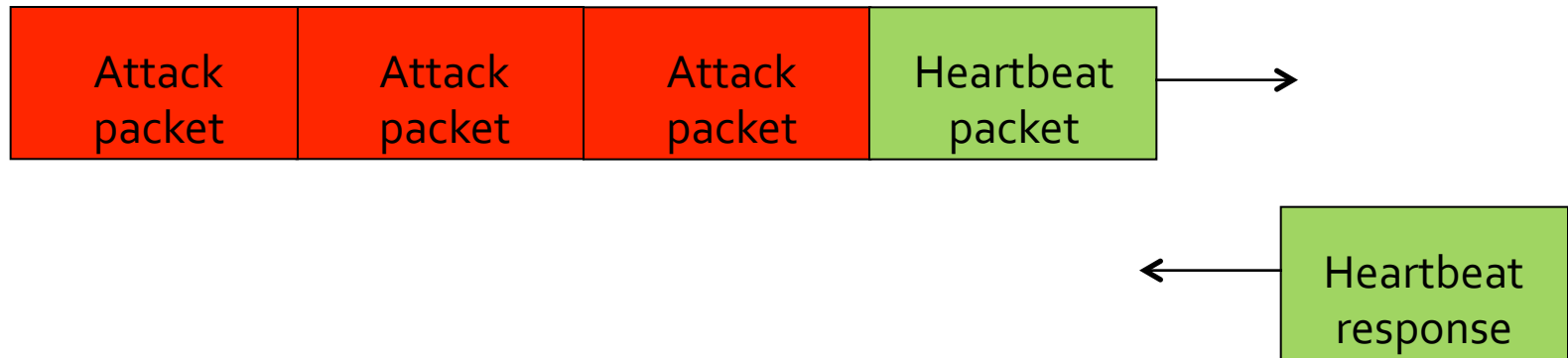
- With care, the timing difference arising from the attack packets can be made cumulative!

- Repeat over many trains and use statistical techniques to detect timing difference.

HMAC-SHA1 + CBC-AES, 10 packets per train, 1456 bytes per packet:
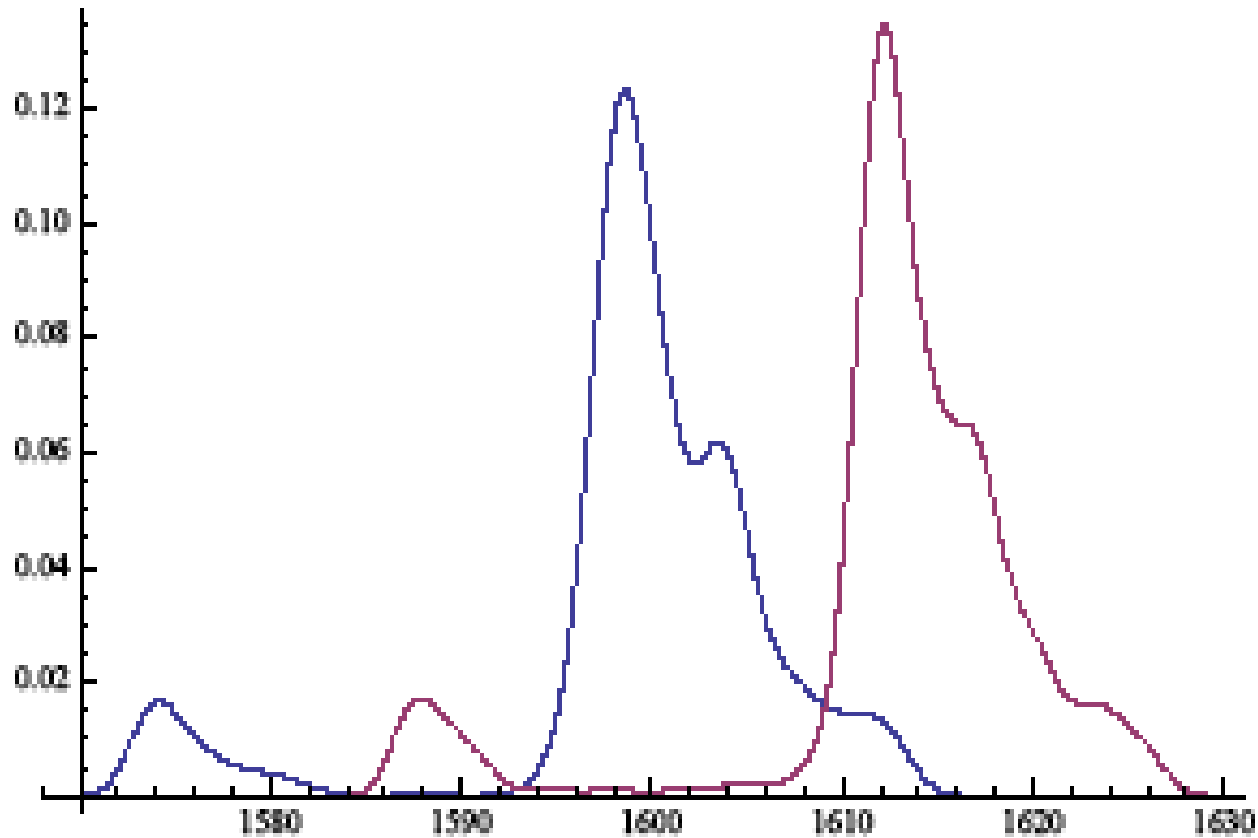
Breaking DTLS in OpenSSL: Experimental Results

- 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

Observations

- One may speculate that the OpenSSL developers did not protect DTLS implementation against timing attacks because of the lack of error messages in DTLS.

    - Nothing to time implies no attack?

- DTLS turned 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.

    - Also enabled us to sharpen our tools for what came next...

Lucky 13 Attack

Padding Oracle Countermeasures, Revisited

From the TLS 1.1 and 1.2 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?

TLS Record Protocol: MAC-Encode-Encrypt
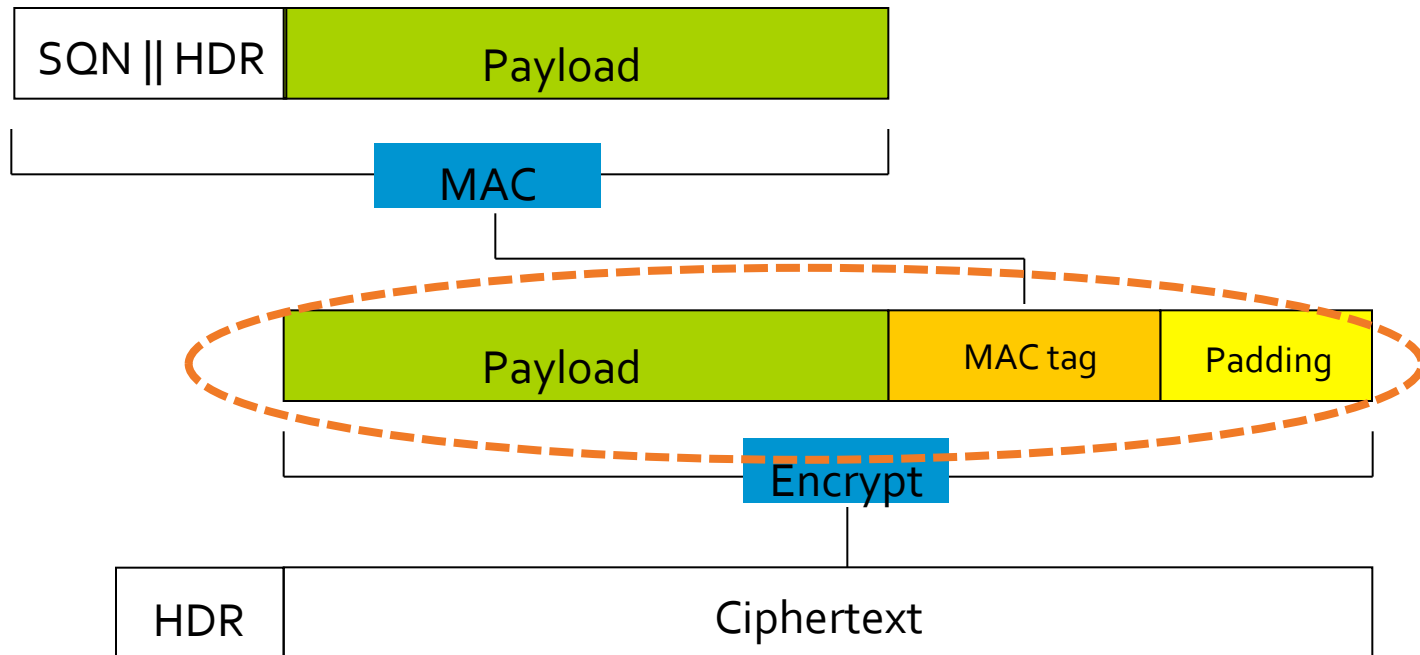
Problem is: how to parse plaintext as payload, padding and MAC fields when the padding is not one of the expected patterns 00, 01 01,... ?

Ensuring Uniform Errors

From the TLS 1.1 and 1.2 specifications:

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, ...

- Other approaches possible (GnuTLS).

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.

Enter Lucky 13 [AP13]

- Distinguishing attacks and full plaintext recovery attacks against TLS-CBC implementations following the advice in the TLS 1.1 and 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 too!

- Demonstrated in the lab against OpenSSL and GnuTLS.

- Full details at www.isg.rhul.ac.uk/tls/Lucky13.html

Lucky 13: Main 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 exact byte length of SQN || HDR || PAYLOAD:

  - $L \leq 55$ bytes: 4 compression function calls;

  - $56 \leq L \leq 119$: 5 compression function calls;

  - $120 \leq L \leq 183$: 6 compression function calls;

  - ....

Uploaded By: anonymous

Lucky 13: Distinguishing Attack
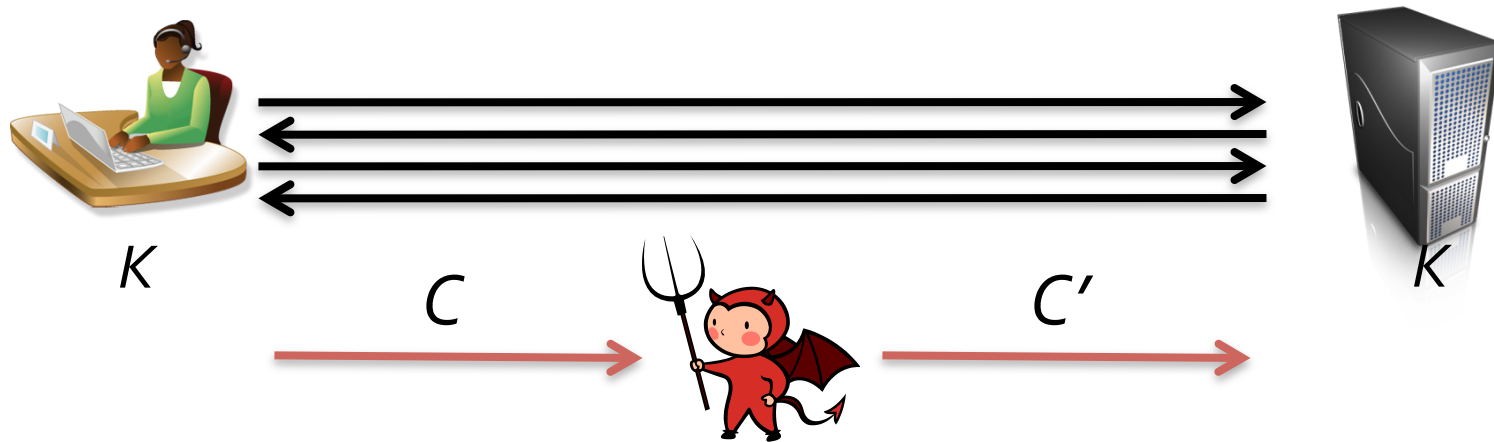
$$C = \text{Enc}_K(M) \qquad M \text{ is either } R^{287} \| 00 \text{ or } R^{32} \| \text{FF}^{256}$$

- Adversary intercepts C, mauls, and forwards on to recipient.

- Time taken to respond with error message will indicate whether $M = R^{287} \| 00$ or $M = R^{32} \| \text{FF}^{256}$.

- Ciphertext-only distinguishing attack.

R	$R\,R$ ....R oo		

IV	C

R	FF FF...............................FF		

IV	C
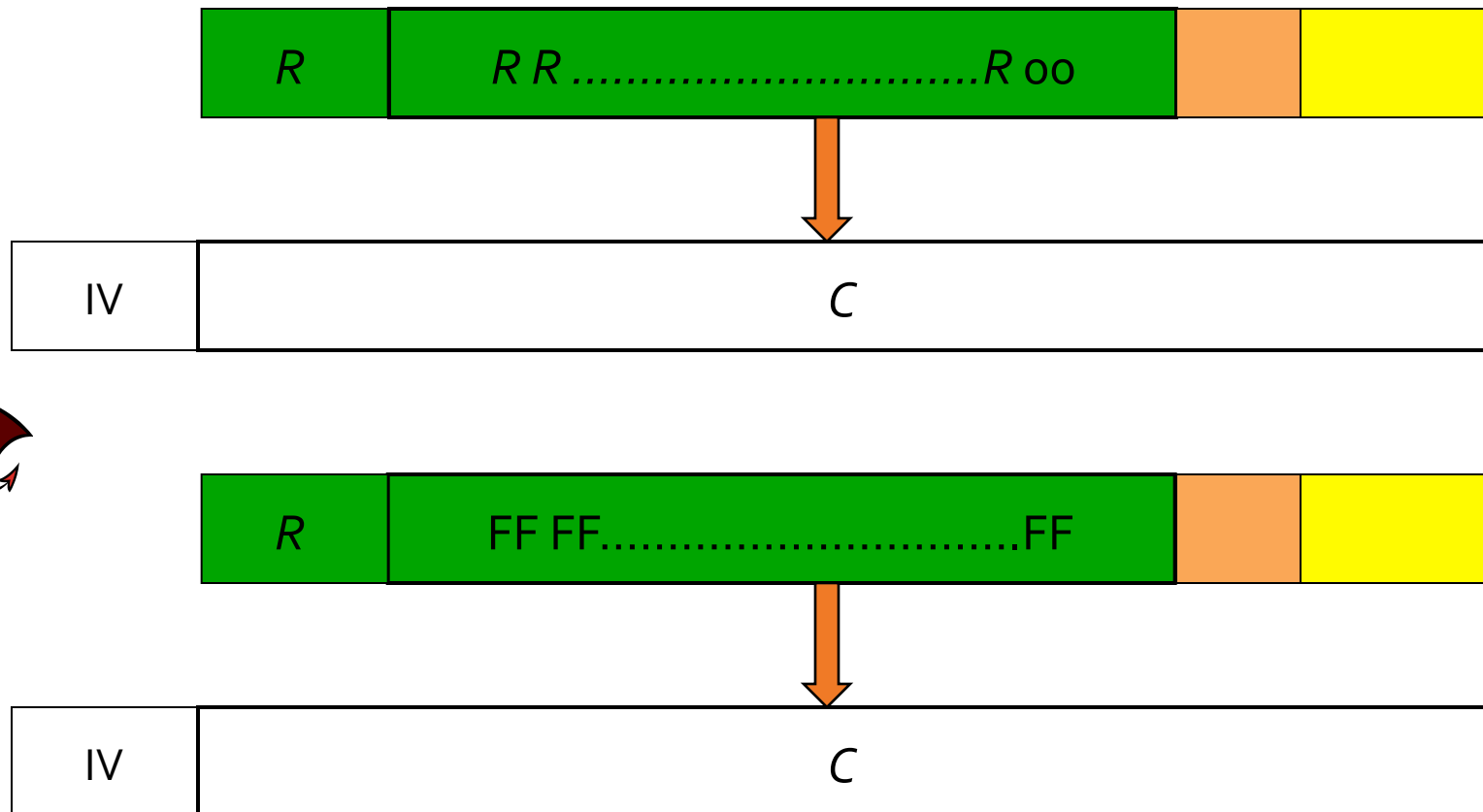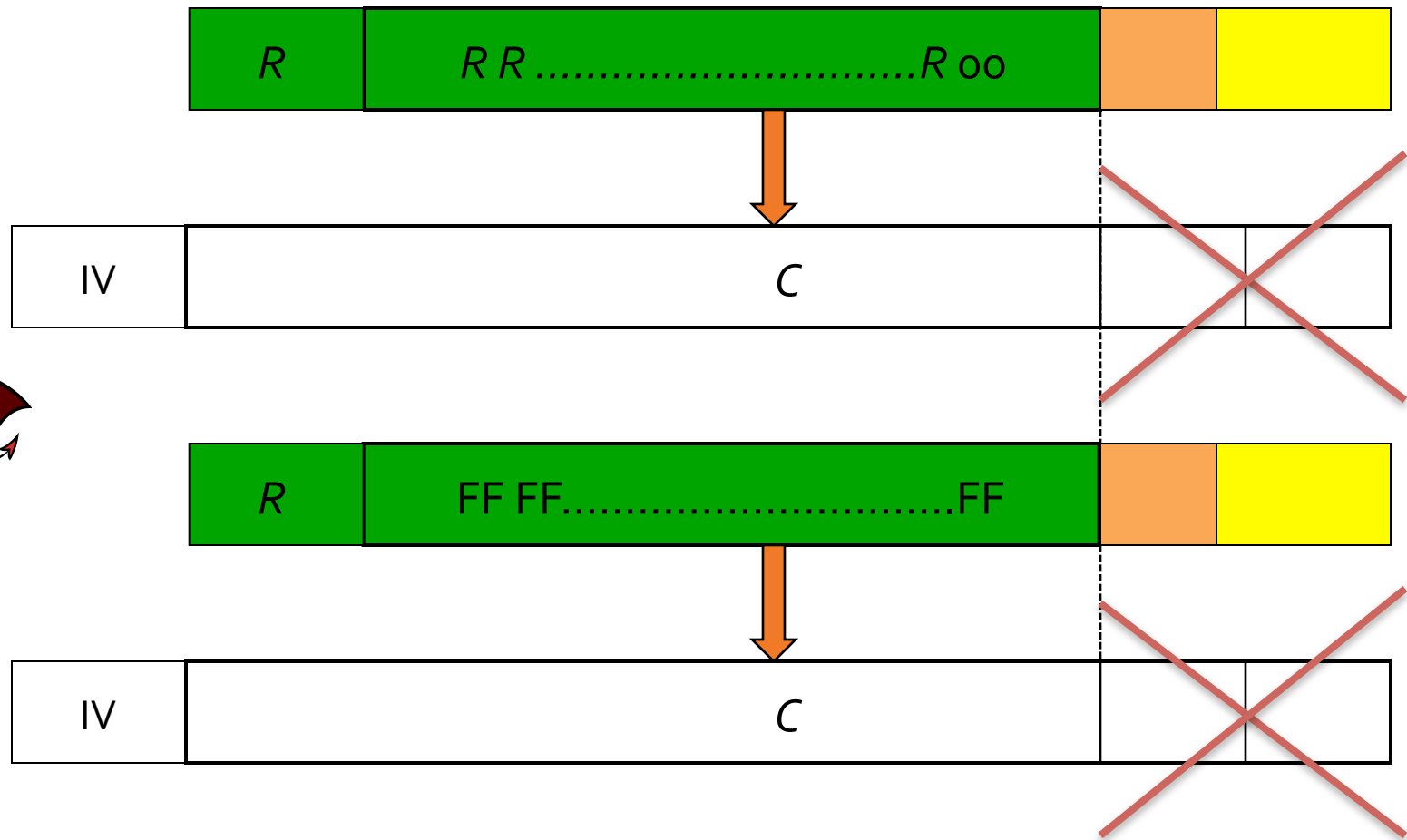
Lucky 13 Distinguishing Attack – Inject

R	R R ....R oo

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

IV	C'

R	FF FF................................FF

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

IV	C'
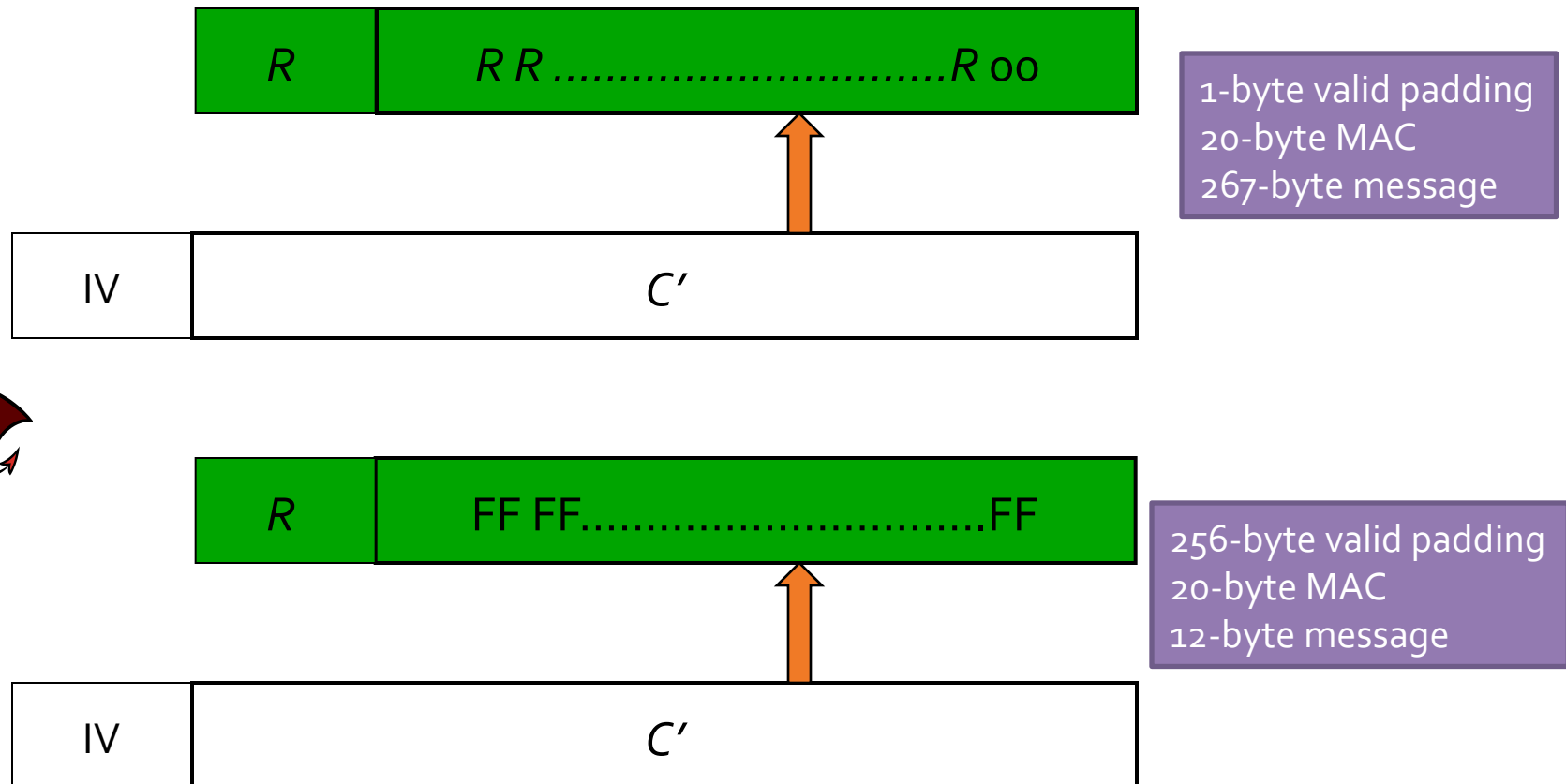
Lucky 13 Distinguishing Attack – Decrypt

SQN||HDR	R	$R\,R\,........................R$	

280 bytes
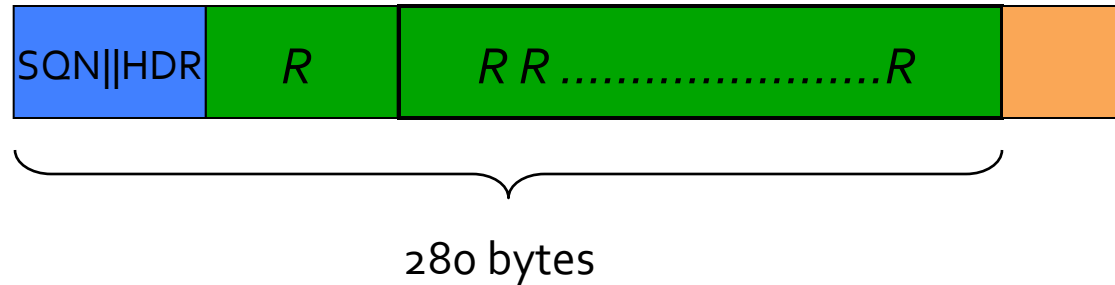
Slow MAC verification

SQN||HDR	R	

25 bytes
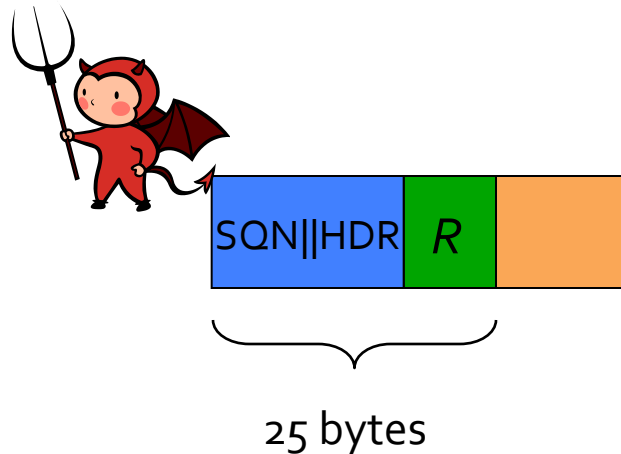
Fast MAC verification

Timing difference: 4 SHA-1 compression function evaluations

OpenSSLv1.0.1 on server running at 1.87Ghz.

100 Mbit LAN.

Difference in means is circa 3.2 µs.

Lucky 13 – Success Probability for Distinguishing Attack

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

Lucky 13 – Plaintext Recovery

My experience:

- Practitioners tend to dismiss distinguishing attacks as being artificial or theoretical.

  - Despite them having a history of leading to stronger attacks.

- They tend to pay more attention when you show them plaintext.

  - Preferably their own password in a live demonstration.

- So we tried to find a way to extend our distinguishing attack to a plaintext recovery attack.

XOR 2-byte Δ here and submit for decryption

Target ciphertext block from stream

| IV | R_1 | R_2 | C_{t-1} | C_t |

d_K d_K d_K d_K

$\oplus$ $\oplus$ $\oplus$ $\oplus$

P_t

Produces valid patterns "01 01" (or longer valid pad) or "00", OR bad pad.

(HMAC-SHA-1 + AES-CBC)

Case 1: "01 01" (or longer valid pad)

XOR 2-byte Δ here and submit for decryption

IV R_1 R_2 C_{t-1} C_t

d_K d_K d_K d_K

SQN||HDR

13 + 16 + 16 + 10 = 55 bytes

20 bytes

"01 01" (or longer valid pad)

4 SHA-1 compression function evaluations

Case 3: Bad padding

XOR 2-byte Δ here and submit for decryption

IV | R_1 | R_2 | C_{t-1} | C_t

d_K d_K d_K d_K

SQN‖HDR

57 bytes

20 bytes

zero-length pad

5 SHA-1 compression function evaluations
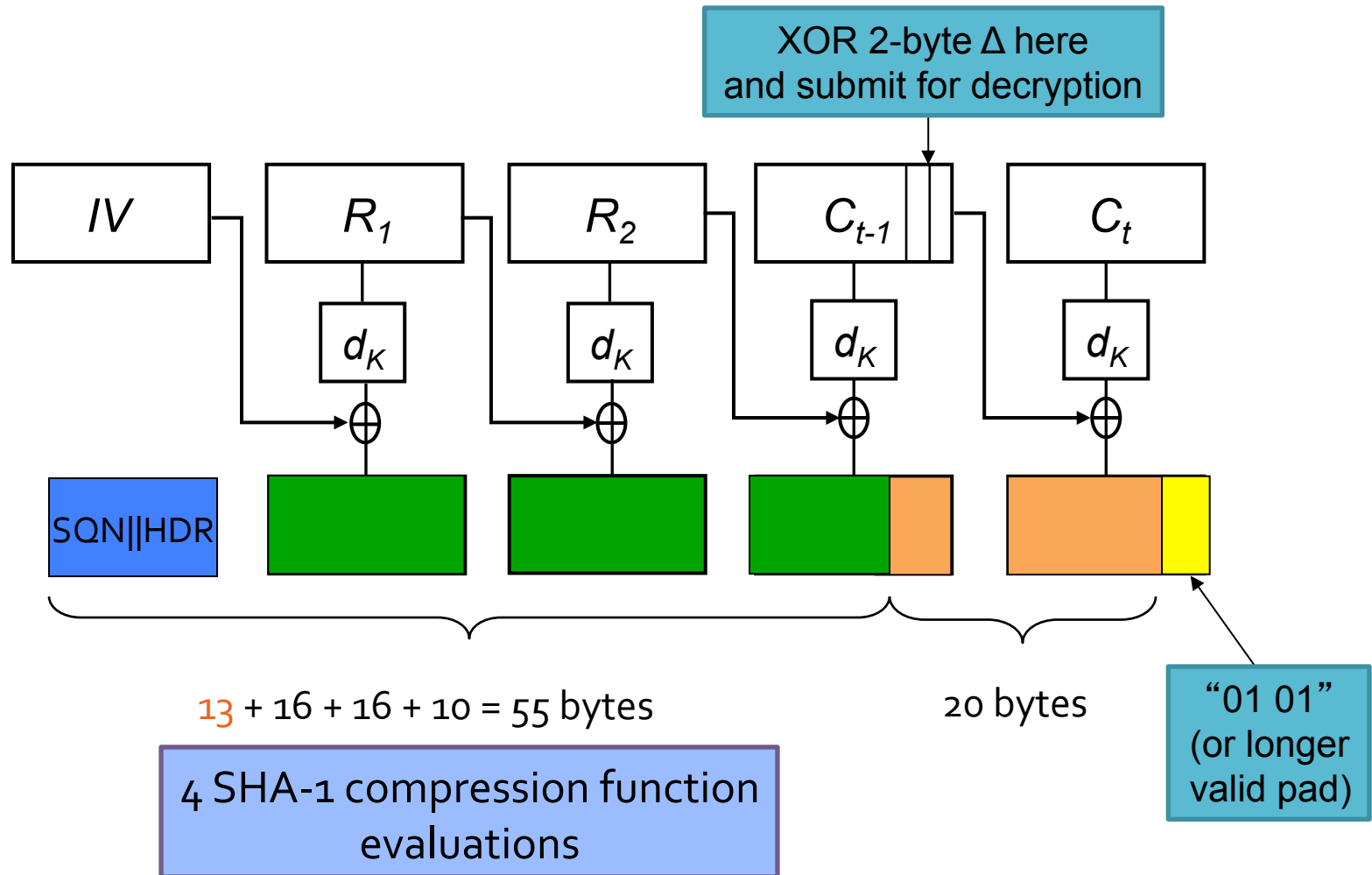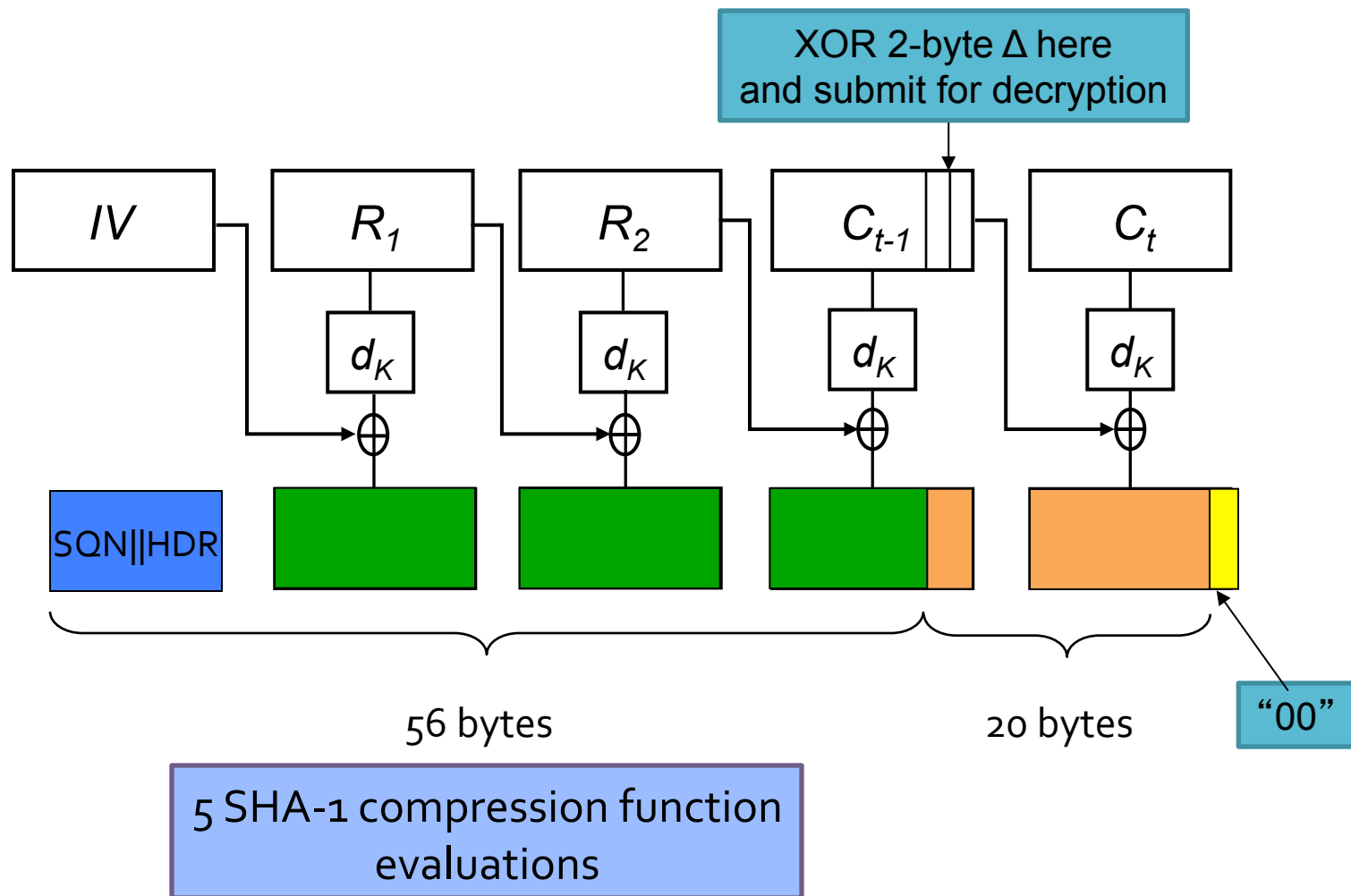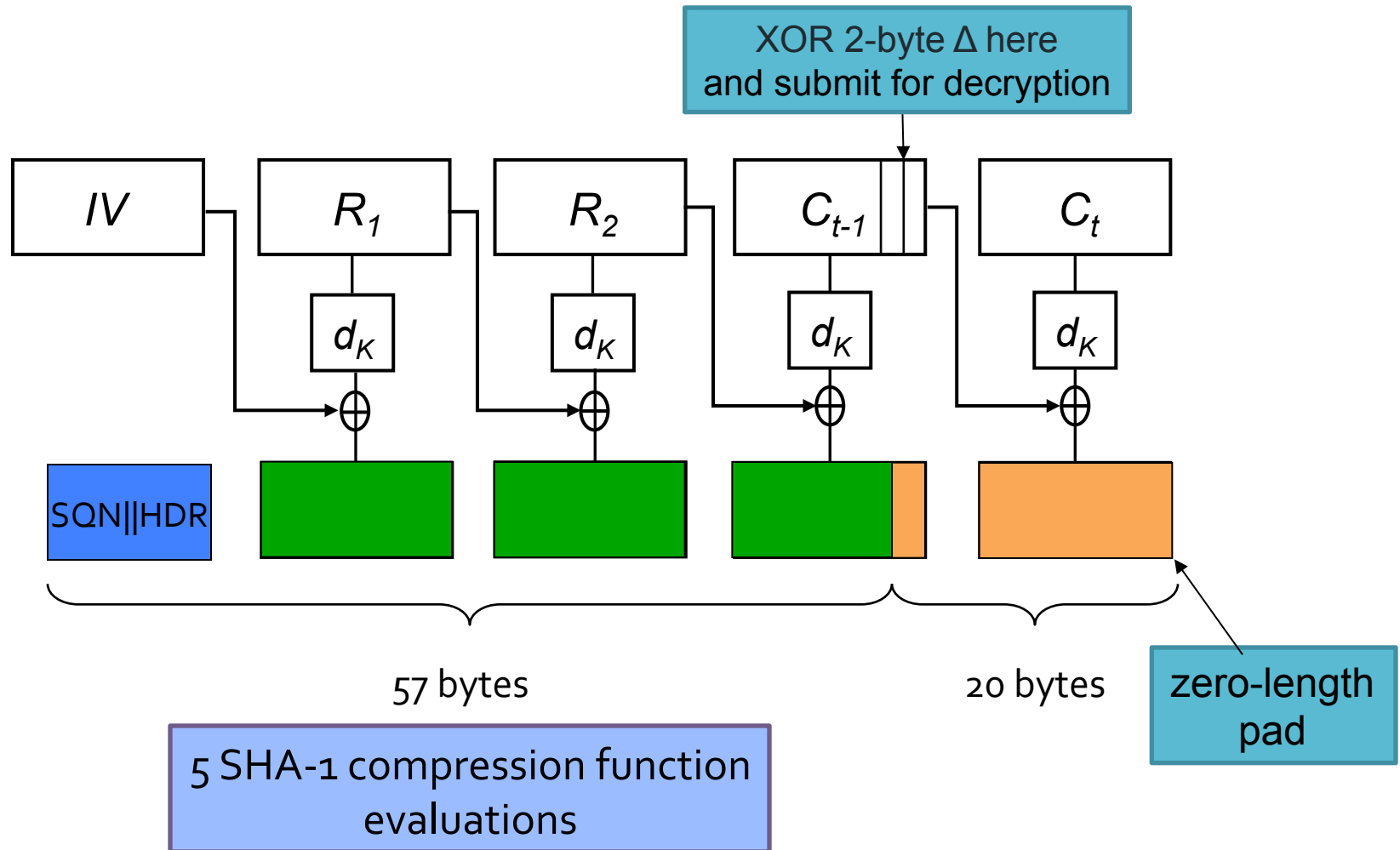
Lucky 13 – Plaintext Recovery

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, circa 1μs 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 standard CBC algebra: $P_t \oplus (.....\Delta_1\Delta_0) = (....0101)$.

Attack then extends to all bytes as in a standard padding oracle attack.

Lucky 13 – Plaintext Recovery

We need 2^{16} attempts to try all 2-byte Δ values.

And we need around 2^7 trials for each Δ value to reliably distinguish the different events.

      (Actual noise level depends on experimental set-up.)

Each trial kills the TLS session.

Hence the headline attack cost is 2^{23} sessions, all encrypting the same plaintext.

Seems rather theoretical?

Lucky 13 – Improvements (Attacks Get Better!)

If 1-out-of-2 last plaintext bytes known, then we only need 2^8 attempts per byte.

If the plaintext is base64 encoded, then we only need 2^6 attempts per byte.

> And 2^7 trials per attempt to de-noise, for a total of 2^{13}.

BEAST-style attack targeting HTTP cookies.

> Malicious client-side Javascript makes HTTP GET requests.
>
> TLS sessions are automatically generated and HTTP cookies attached.
>
> GET requests are "padded" so that 1-out-of-2 condition always holds.
>
> Cost of attack is 2^{13} GET requests per byte of cookie.
>
> Now a practical attack!
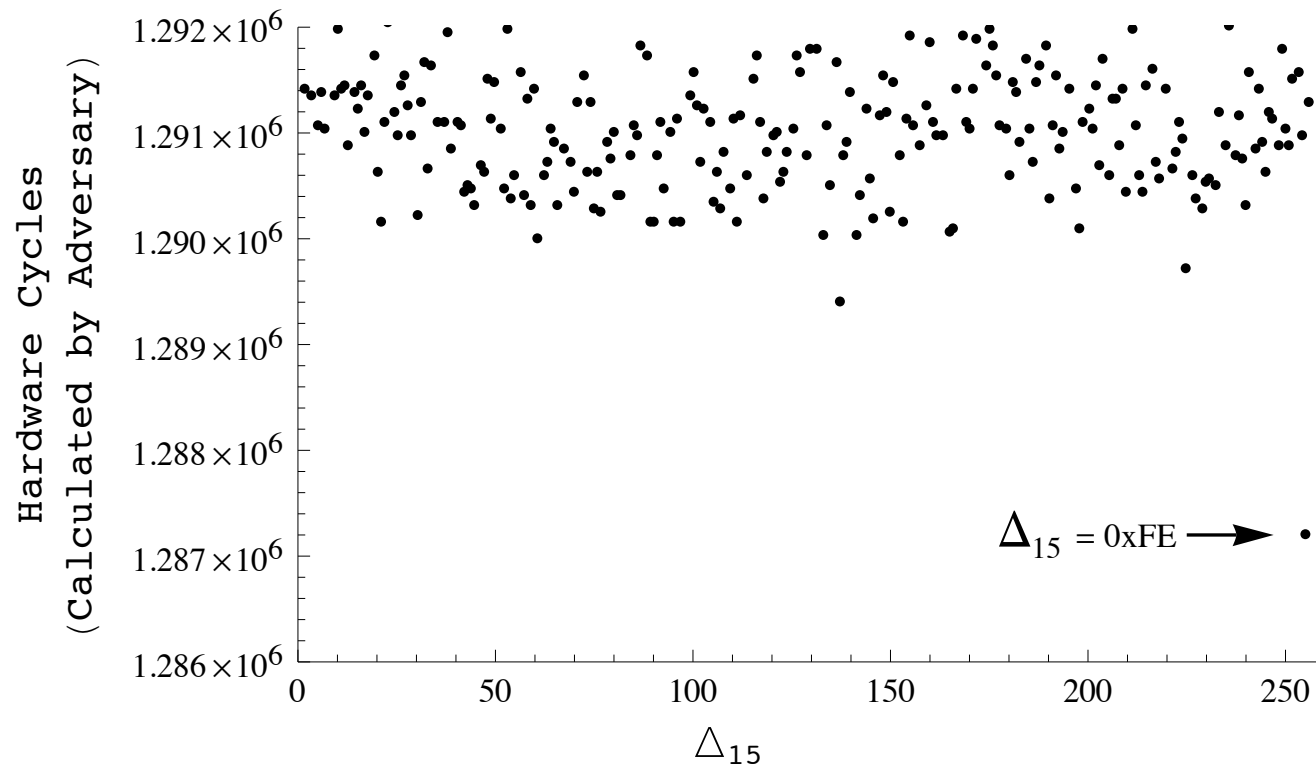
Byte 14 of plaintext set to 01; byte 15 set to FF.

Modify Δ in position 15.

OpenSSLv1.0.1 on server running at 1.87Ghz, 100 Mbit LAN.

Median times (noise not shown).

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 Basic Countermeasures

Before
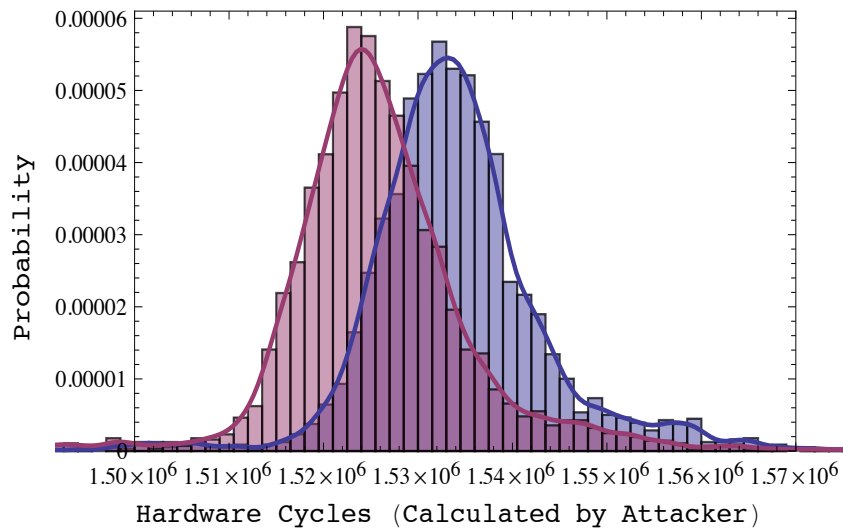
After
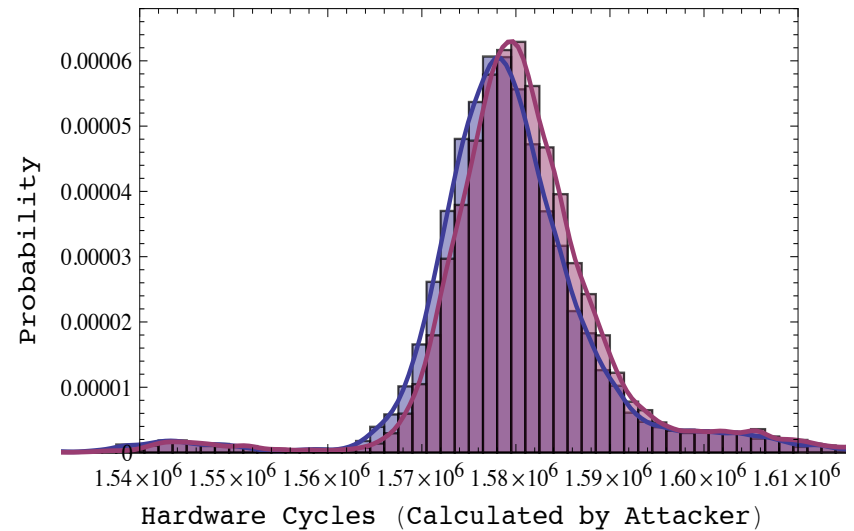
Constant Time Decryption for MEE

- Better, but still not perfect.

  - Distinguishing attack still possible.

- Proper constant-time, constant-memory access implementation is really needed.

  - Challenging to test padding correctness and do sanity checking without branching on secret data.

- See Adam Langley's blogpost at:

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

  for full details on how Lucky 13 was fixed in OpenSSL and NSS.

- Not all implementations were fixed so thoroughly...

Lucky 13 – Impact

OpenSSL patched in versions 1.0.1d, 1.0.0k and 0.9.8y, released 05/02/2013.

NSS (Firefox, Chrome) patched in version 3.14.3, released 15/02/2013.

Apple: patched in OS X v10.8.5 (iOS version tbd).

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".

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

Other Lucky 13 Countermeasures?

Introduce random delays during decryption.

Surprisingly **ineffective**, analysis in [AP13].

Redesign TLS:

Pad-MAC-Encrypt or Pad-Encrypt-MAC?

Pad-Encrypt-MAC has been developed by IETF as a TLS extension for TLS 1.1 and higher.

Will take months/years to deploy.

Switch to TLS 1.2

Has support for AES-GCM and AES-CCM.

But was not widely supported by browsers or servers at the time Lucky 13 was announced.

Switch to RC4

As recommended by many commentators (again!).

More soon…

Uploaded By: anonymous

Lucky 13 – 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.

Better to use an EtM construction from day one, or eat the cost of switching at the first sign of trouble.

> A conservative approach seems merited for such an important protocol.

> At the time TLS was first designed, EtM versus MtE debate was not so clear cut.

Yet Another Attack… Short MACs

Are We There Yet?

Implementations of TLS in CBC mode should by 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.
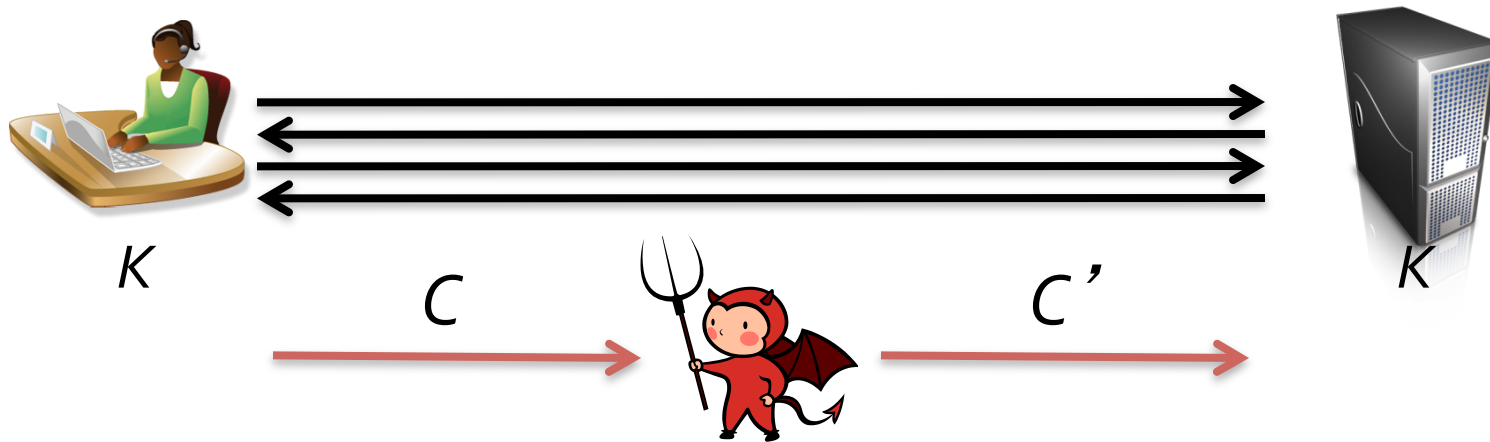
$C = \text{Enc}_K(M)$ $\qquad$ M is either "Yes" or "No"

- 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.

Byte 13 is hex for 19

Byte 12 is hex for 18

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

$$C' = C_o'C_1$$

No | 13^4 | 13^{16}

e_K | e_K

C_o | C_1 | C_2

Yes | 12^3 | 12^{16}

e_K | e_K

C_o | C_1 | C_2

No | | 03^4
13^{16}

$\oplus$ | $\oplus$

e_K | e_K

$C_0{}'$ | C_1 | C_2
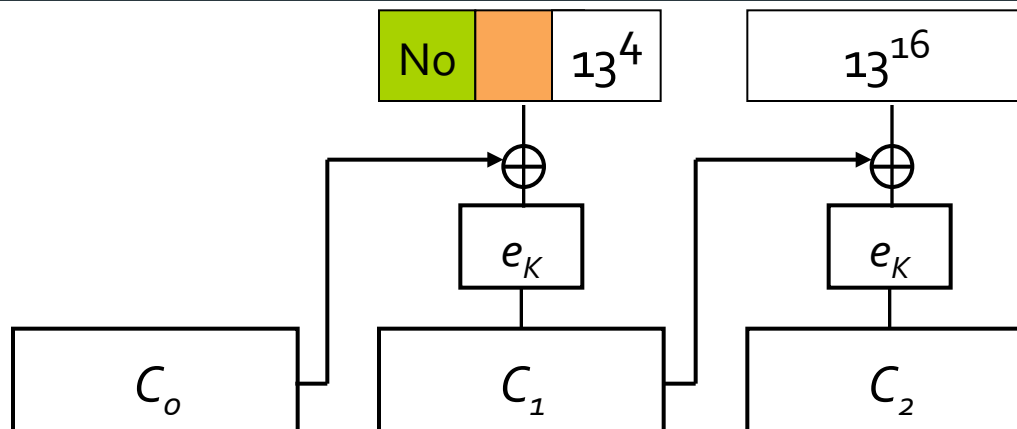
Decrypts fine to "No"

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

$C' = C_0{}'C_1$

Yes | | 12^3
12^{16}

$\oplus$ | $\oplus$

e_K | e_K

C_0 | C_1 | C_2

No | 03^4 | 13^{16}

e_K | e_K

C_o' | C_1 | C_2
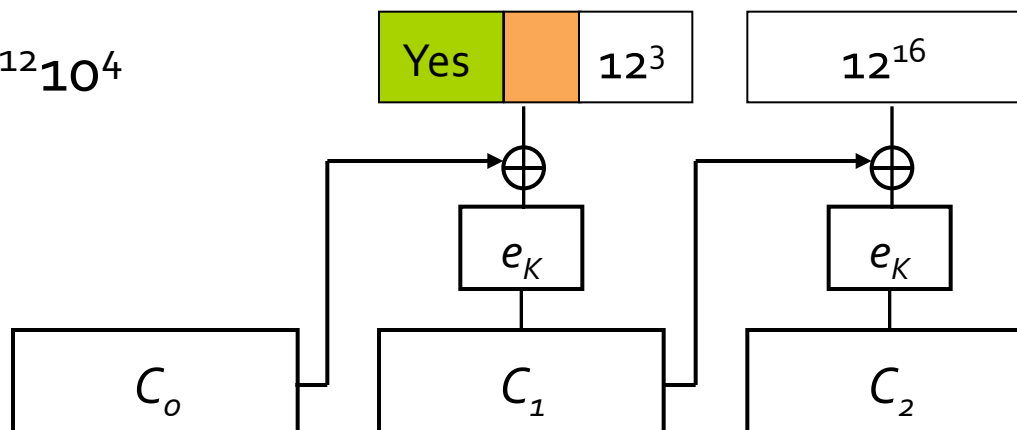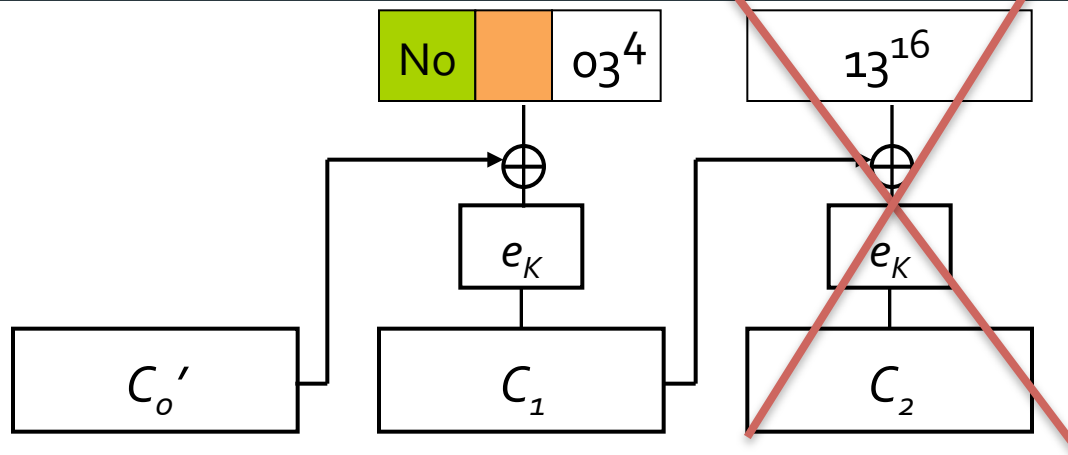
Decrypts fine to "No"

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

$C' = C_o'C_1$

Yes | ?? | 02^3 | 12^{16}

e_K | e_K

C_o' | C_1 | C_2
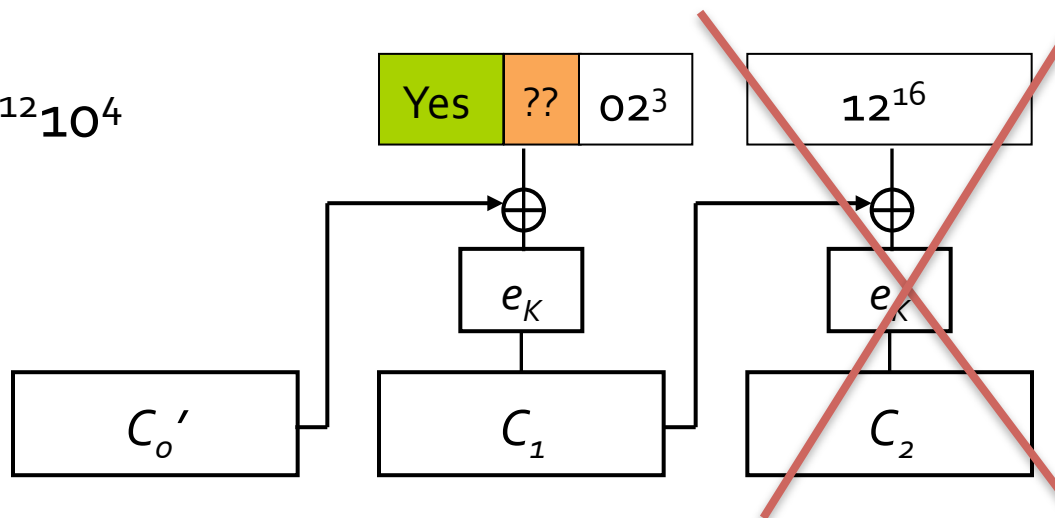
MAC will not verify, decryption fails

Where Does the Attack Apply?

For TLS 1.2:

<u>Block length</u>
n = 64   for 3DES
n = 128 for AES

<u>MAC length</u>
t = 128   for HMAC-MD5
t = 160   for HMAC-SHA1
t = 256   for HMAC-SHA256
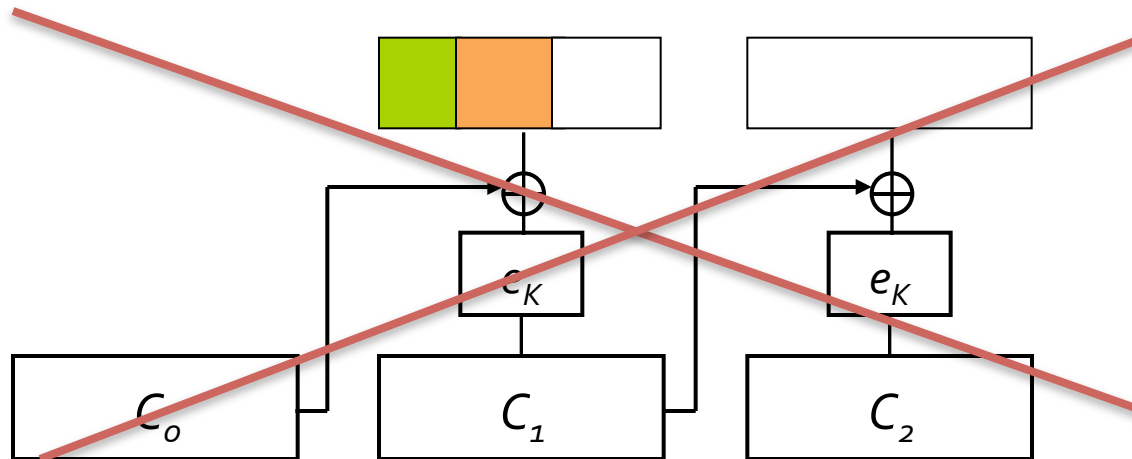
Where Does the Attack Apply?

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

<u>Block length</u>   <u>MAC length</u>

$n = 64$ 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!
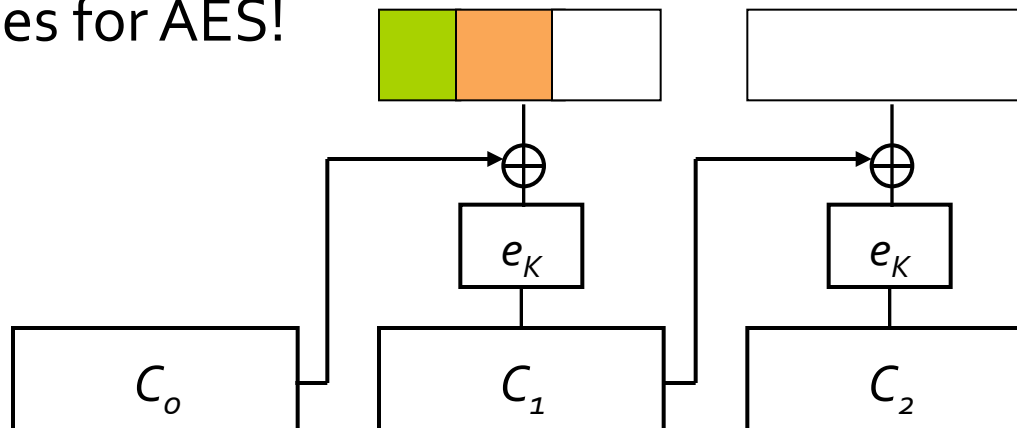
Attack Consequences

- 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 also 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.

AE Security for MEE in TLS

<u>Theorem</u> ([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.
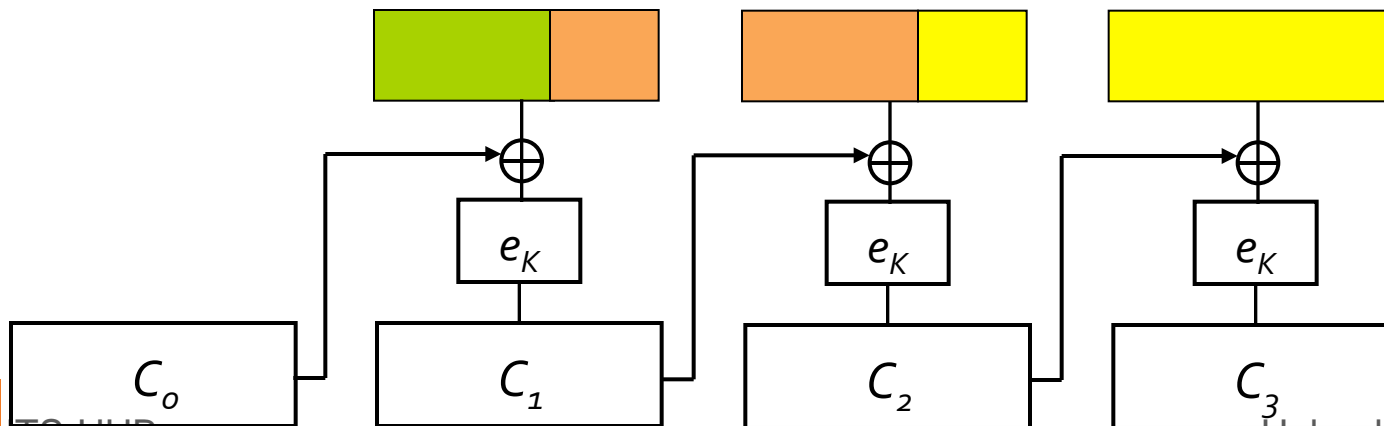
(Tag) Size Matters!

Practical attacks exist
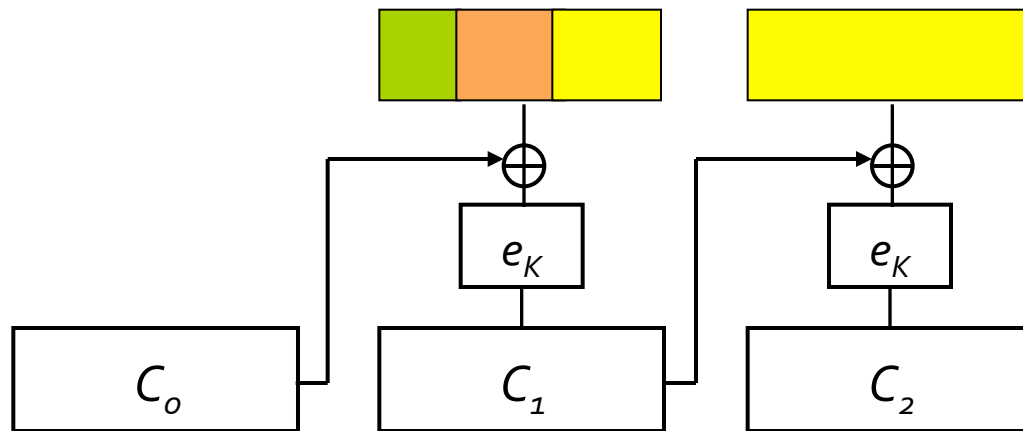
Secure in the (LH)AE model
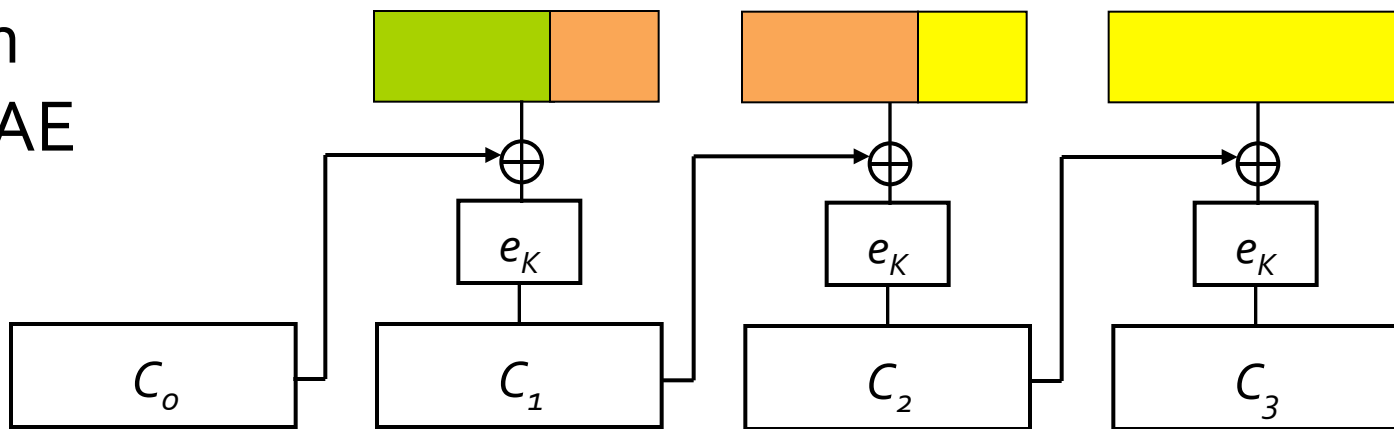
POODLE

SECURITY

Truly scary SSL 3.0 vuln to be revealed soon: sources

So worrying, no one's breathing a word until patch is out

By Darren Pauli, 14 Oct 2014   **Follow**   2,546 followers

23

Gird your loins, sysadmins: *The Register* has learned that news of yet another major security vulnerability - this time in SSL 3.0 - is probably imminent.

RELATED STORIES

Maintainers have kept quiet about the vulnerability in the lead-up to a patch release expected in in the late European evening, or not far from high noon Pacific Time.

OpenVPN open

Details of the problem are under wraps due to the severity of the vulnerability.

POODLE = BEAST + Moeller Attack

Recall (from previous lecture):

[M02]: failure to check padding format leads to a simple attack recovering the last byte of plaintext from any target block.

Uploaded By: anonymous
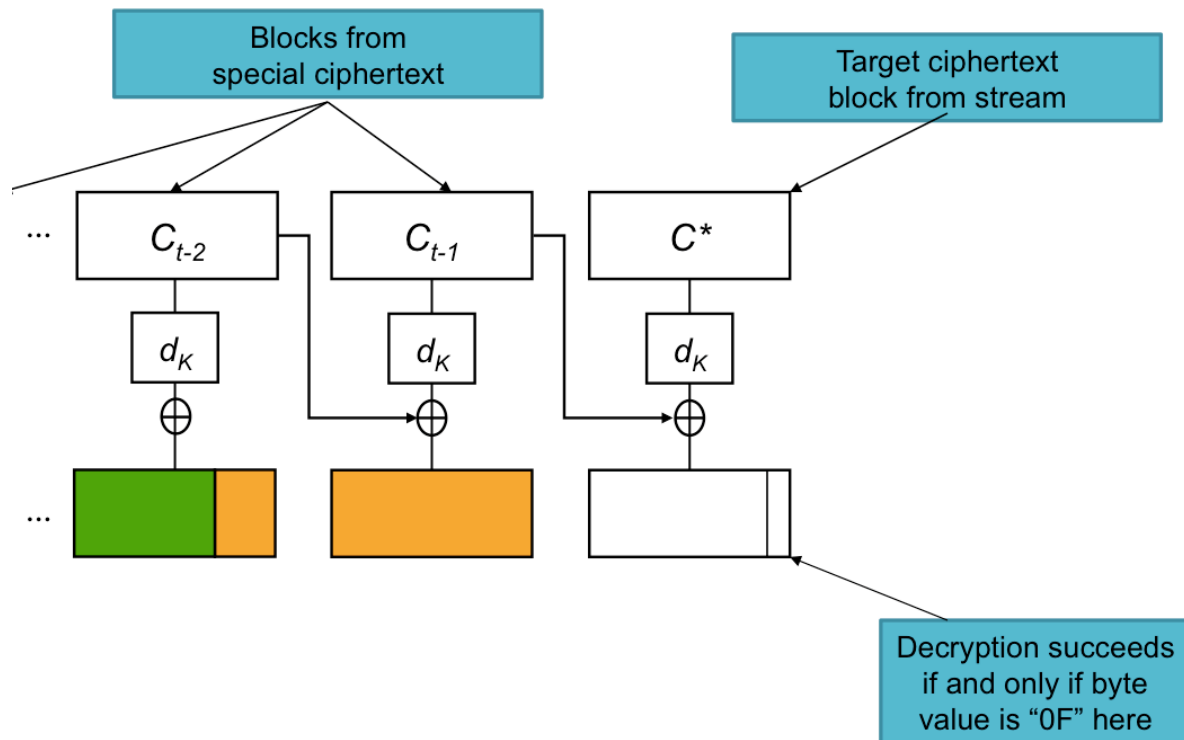
POODLE = BEAST + Moeller Attack

POODLE (https://www.openssl.org/~bodo/ssl-poodle.pdf)

In SSLv3, encryption uses random padding; only the last byte is used to remove padding.

Repeat:

1. Use Javascript in the browser to pad HTTP GET requests (as in BEAST), ensuring that target cookie byte is placed as last byte of block and that the MAC field aligns on a block boundary.

2. Do Moeller's attack with that block to recover the cookie byte with probability 1/256.

Until (all cookie bytes are recovered).

Patching against POODLE?

A patch that does not work: upgrade decryption to do full padding check and Lucky 13 protection.

- But sender may not use correct padding format (it's not required in SSLv3).

- So this would not be deployable unless ALL clients and servers upgraded simultaneously.

- Should not use RC4 either (see next section).

- No ciphersuites left.

(You just witnessed the death of SSLv3!)

But no-one is using SSLv3, right?
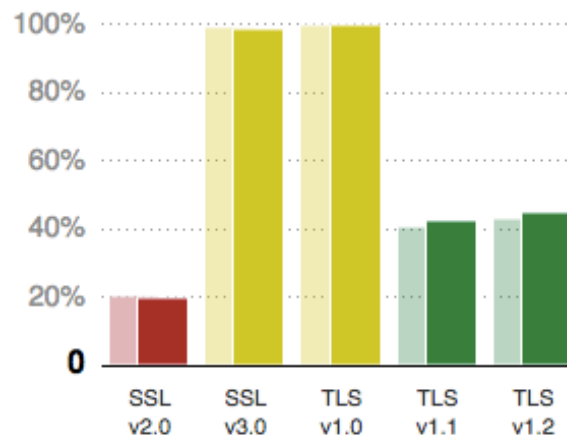
Source: SSL pulse,
15th Oct. 2014

Rich Salz (TLS mailing list, 15/10/14): Akamai sees < 1% SSLv3 traffic

But the attack is made worse because of version downgrade attack:

- Active MITM attacker can always force client and server to downgrade to SSLv3.

- Because the SSL/TLS version negotiation process is not stateful and not cryptographically protected.

Attacking RC4 in TLS

TLS Record Protocol: RC4-128

SQN || HDR	Payload

MAC

Payload	MAC tag	

Encrypt

HDR	Ciphertext

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

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

RC4 State

Byte permutation $\mathcal{S}$ and indices i and j

RC4 Key scheduling

begin
   for $i = 0$ **to** 255 **do**
      $\mathcal{S}[i] \leftarrow i$
   end
   $j \leftarrow 0$
   for $i = 0$ **to** 255 **do**
      $j \leftarrow j + \mathcal{S}[i] + K[i \bmod \mathit{keylen}] \bmod 256$
      $\mathrm{swap}(\mathcal{S}[i], \mathcal{S}[j])$
   end
   $i, j \leftarrow 0$
end

RC4 Keystream generation

begin
   $i \leftarrow i + 1 \bmod 256$
   $j \leftarrow j + \mathcal{S}[i] \bmod 256$
   $\mathrm{swap}(\mathcal{S}[i], \mathcal{S}[j])$
   $Z \leftarrow \mathcal{S}[\ \mathcal{S}[i] + \mathcal{S}[j] \bmod 256\ ]$
   return Z
end

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]:

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

[Mironov 2002]:

Described distribution of Z_1 (bias away from 0, sine-like distribution)

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

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

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

$$\Pr[Z_l = 256 - l] \geq \tfrac{1}{256} + \tfrac{1}{256^2} \qquad l = \text{keylength}$$

What's Going On Here?

Why were people still using RC4 in half of all TLS connections when we already knew it was a weak stream cipher?

"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. Experts say `use RC4'"

            "Google uses it, so it must be OK for my site".

                "There's no demonstrated attack – show me the plaintext!"

Complete Keystream Byte Distributions

Approach in [ABPPS13]:

Based on the output from 2^{45} random independent 128-bit RC4 keys, estimate the keystream byte distributions for the first 256 bytes
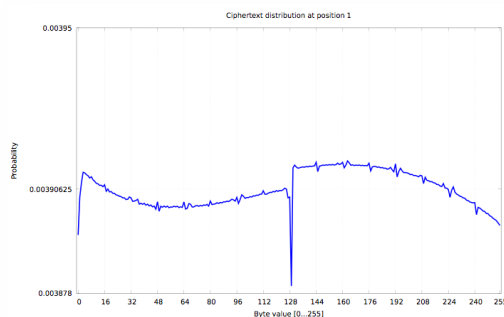
Z_1
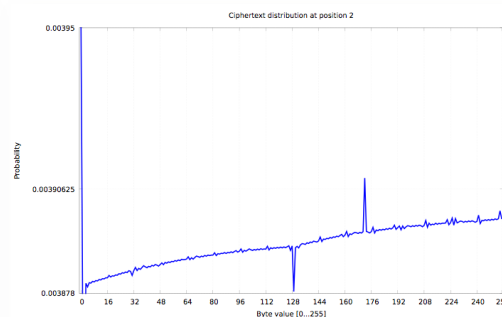
Z_2
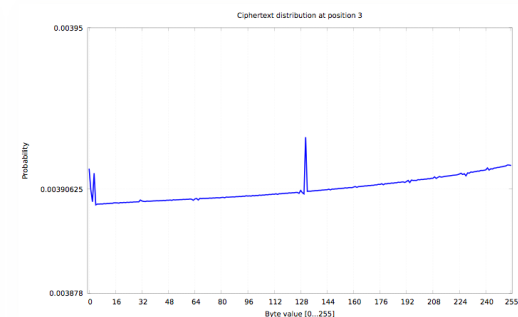
Z_3

...

...

Revealed many new biases in the RC4 keystream.

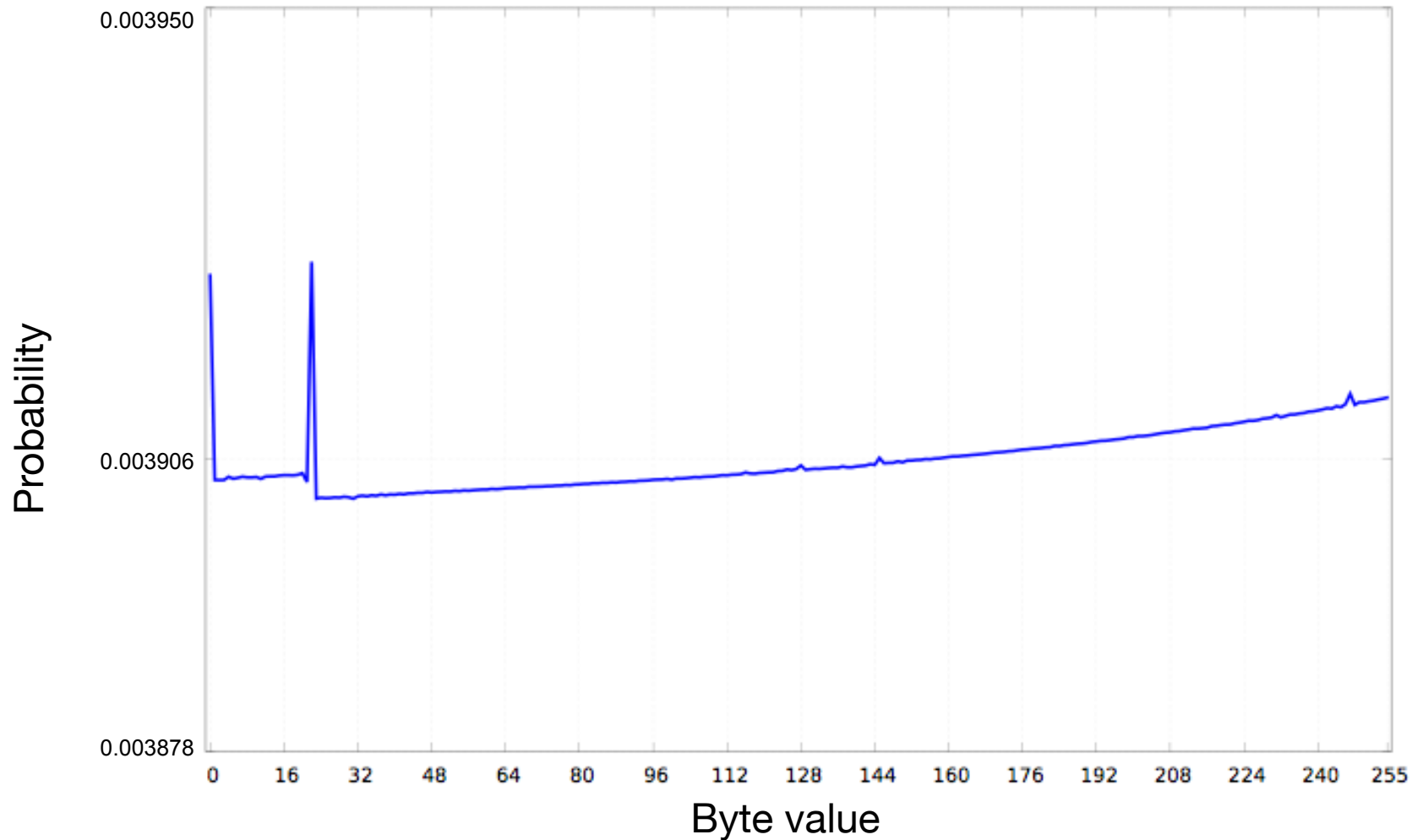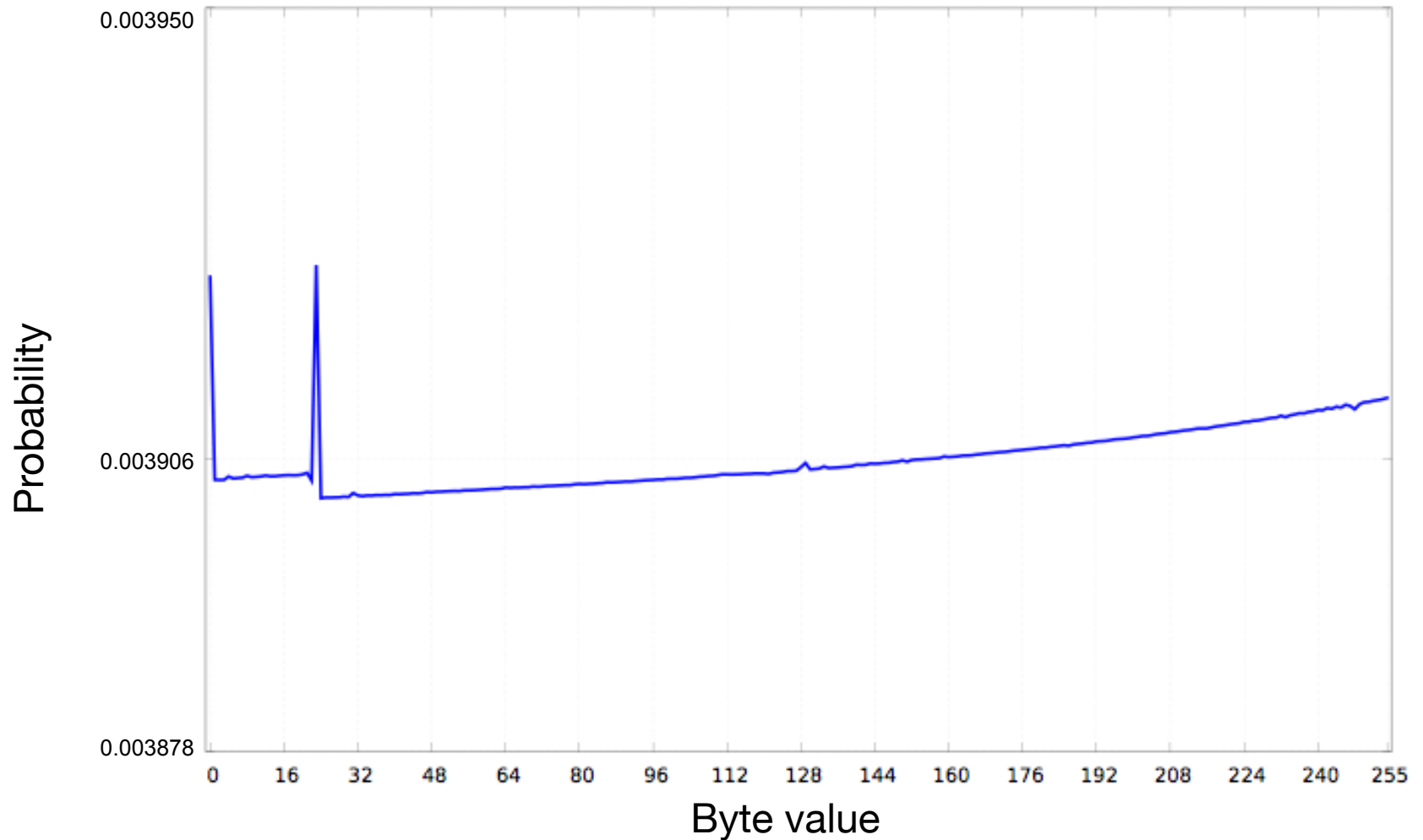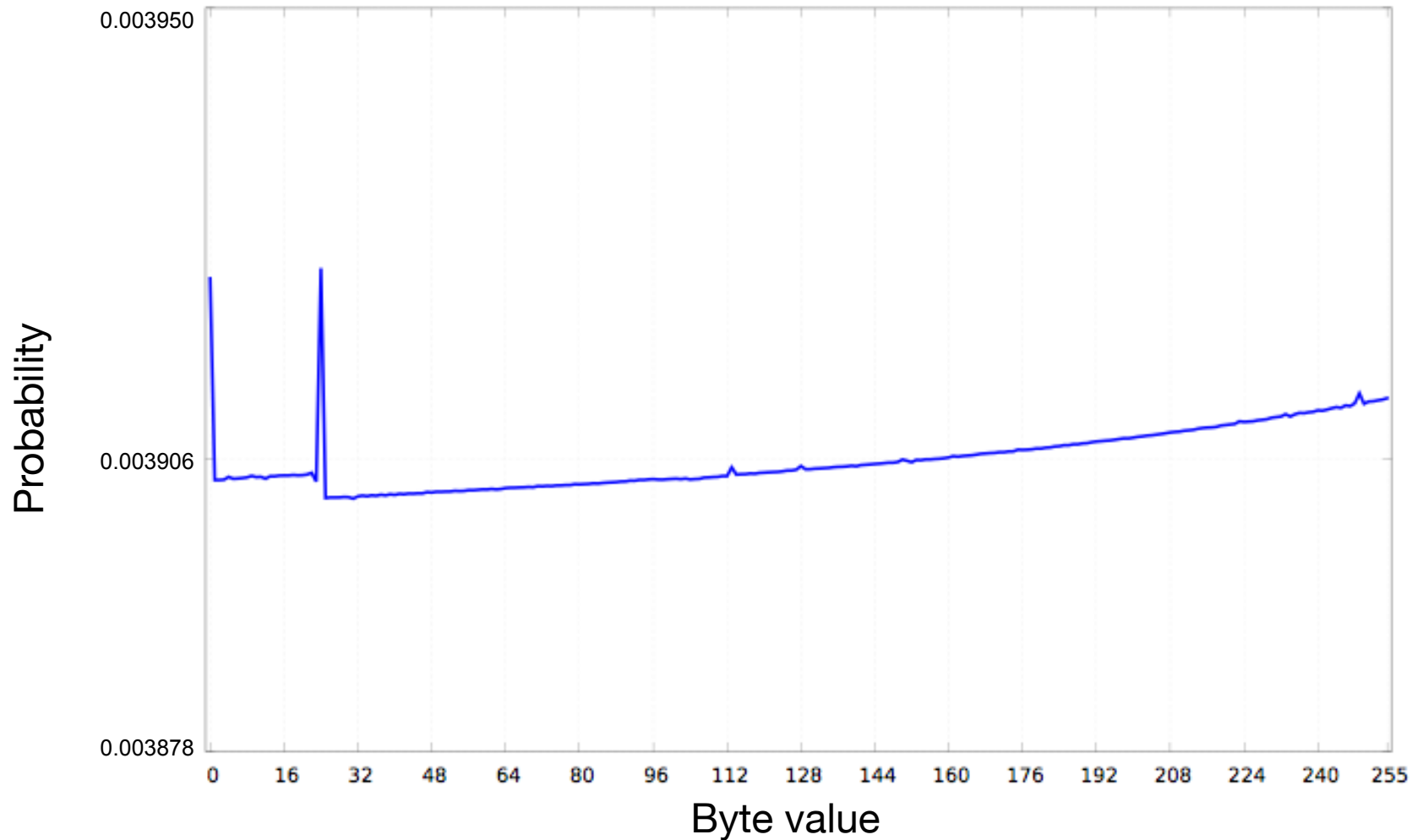(Some of these were independently discovered by Isobe *et al.*)

Keystream Distribution at Position 1
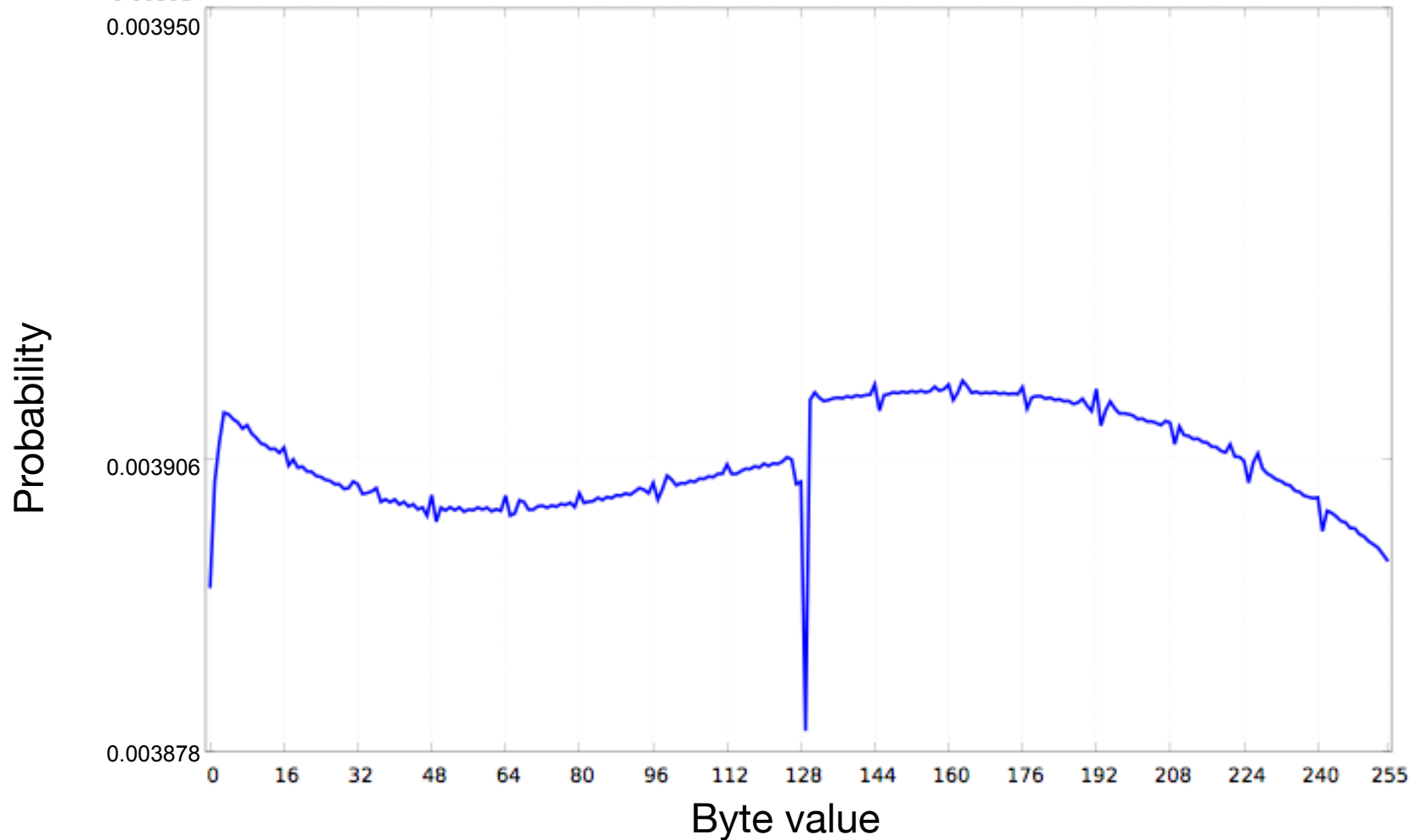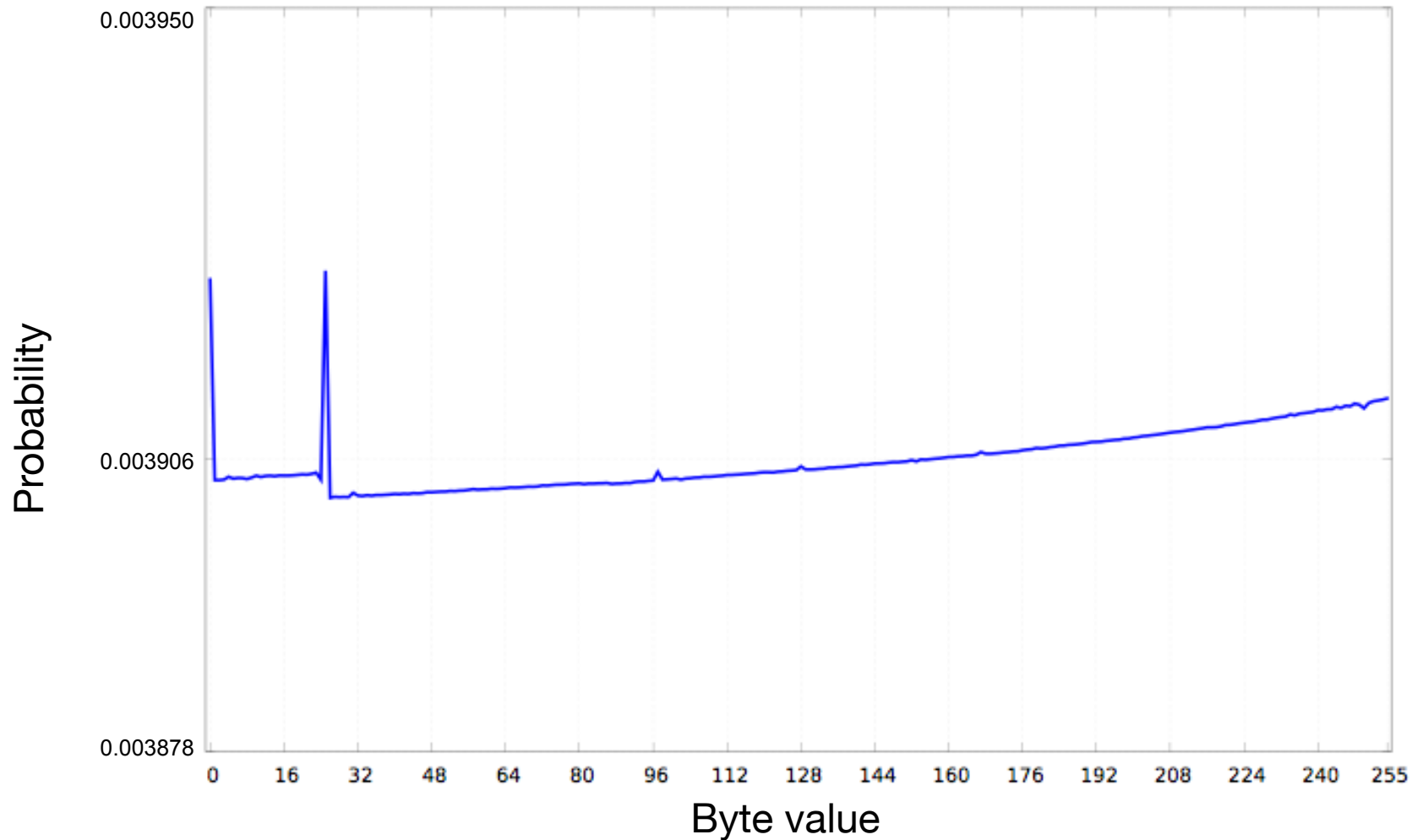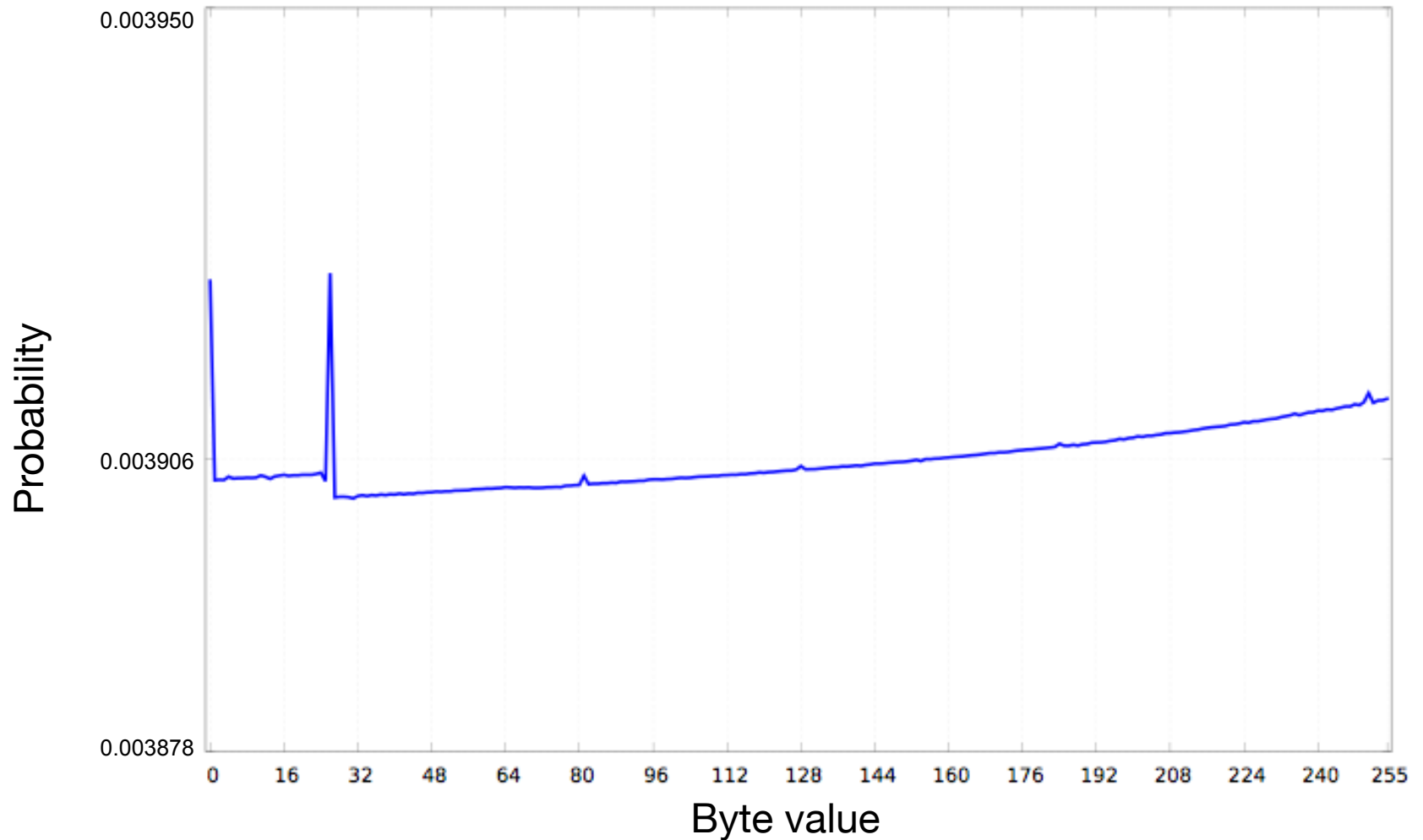
Keystream Distribution at Position 2
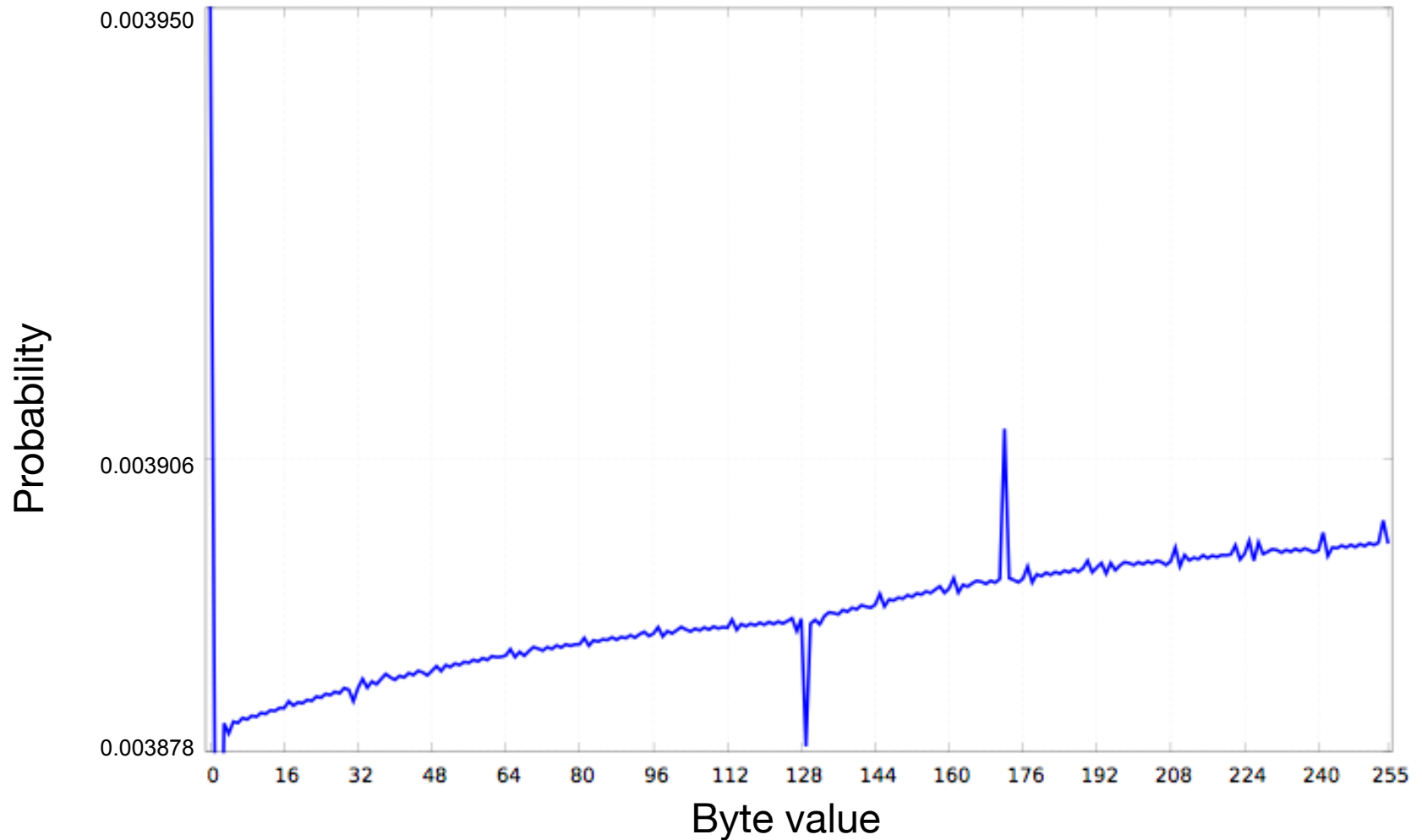
Keystream Distribution at Position 3
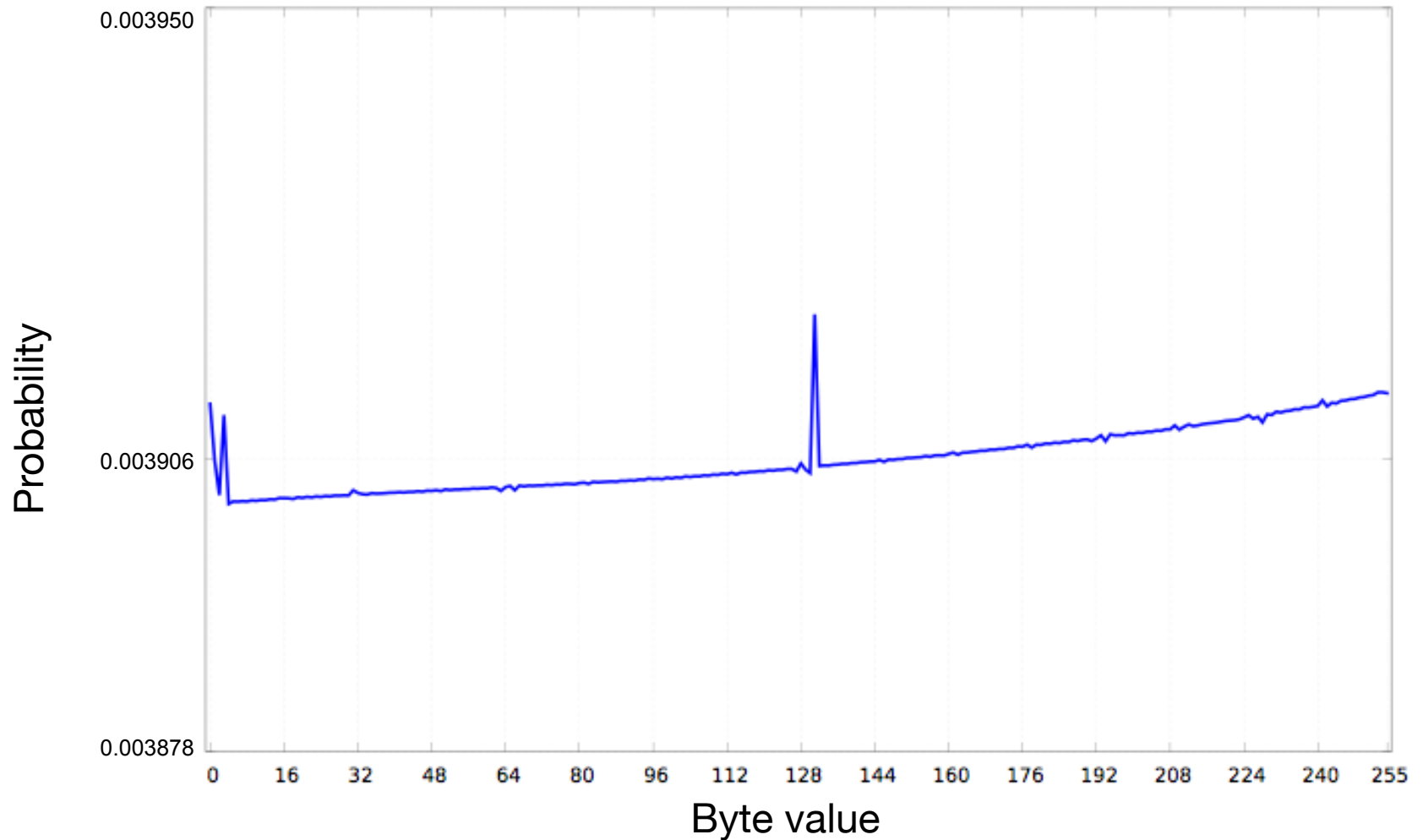
Keystream Distribution at Position 4
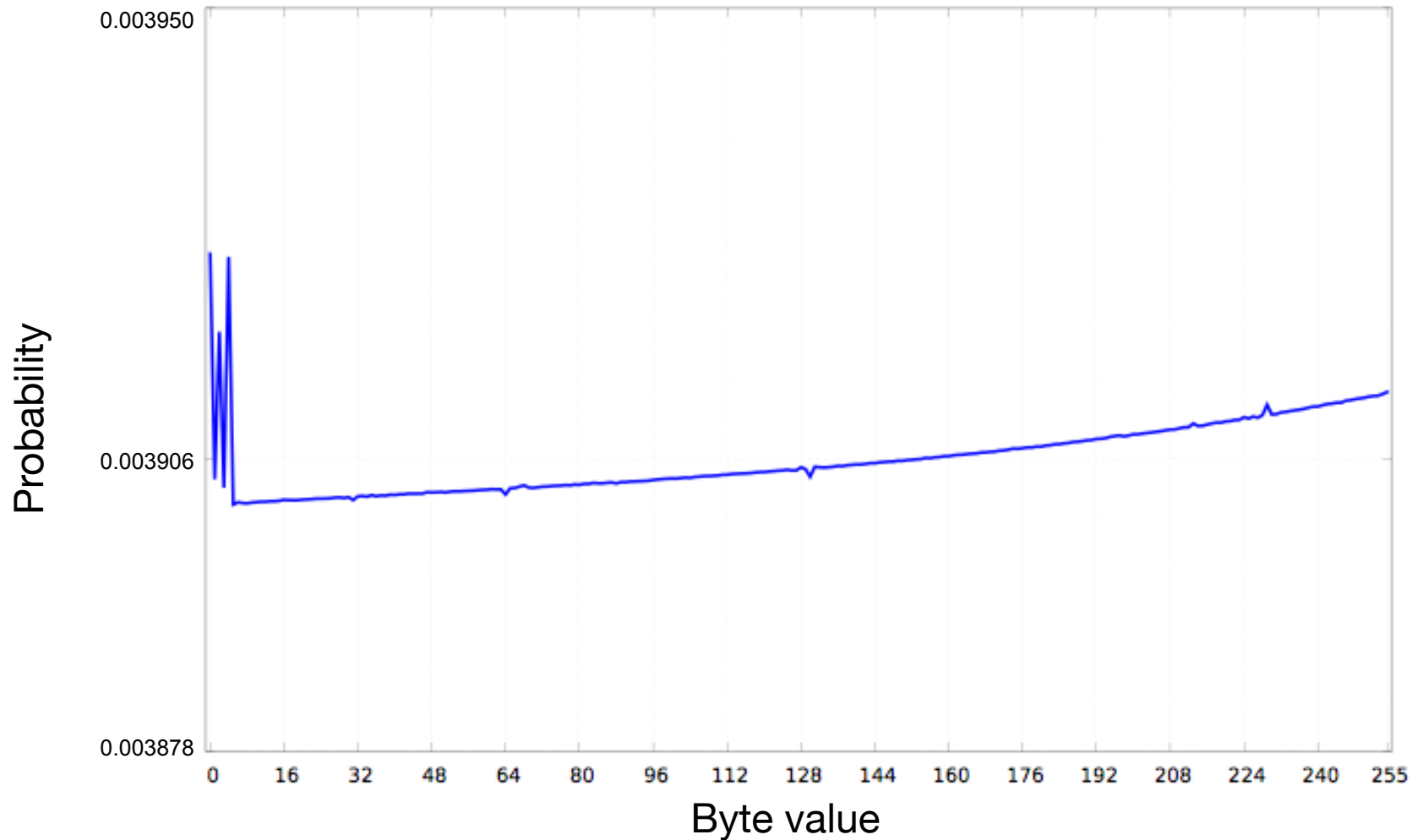
Keystream Distribution at Position 5
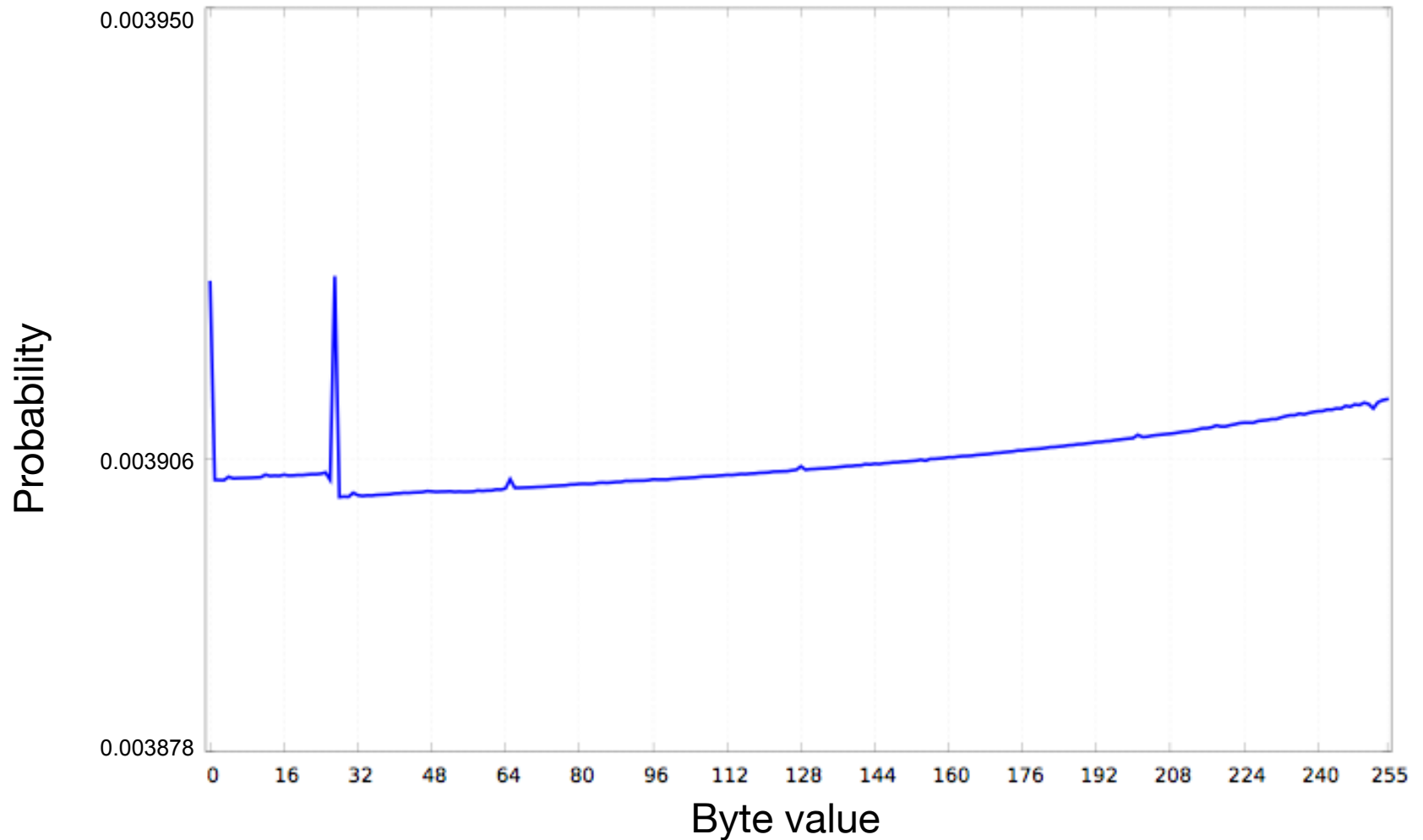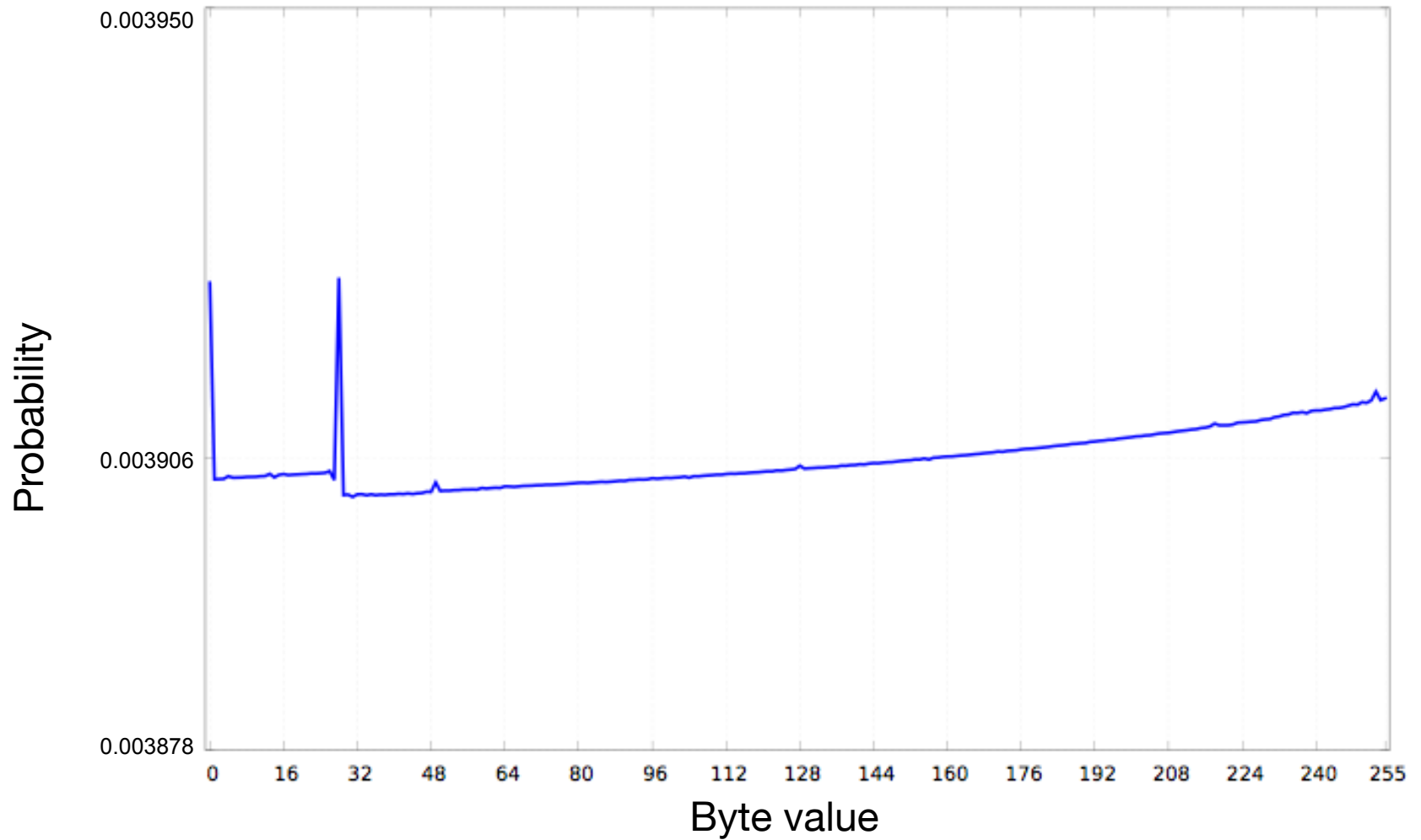
STUDENTS-HUB.com
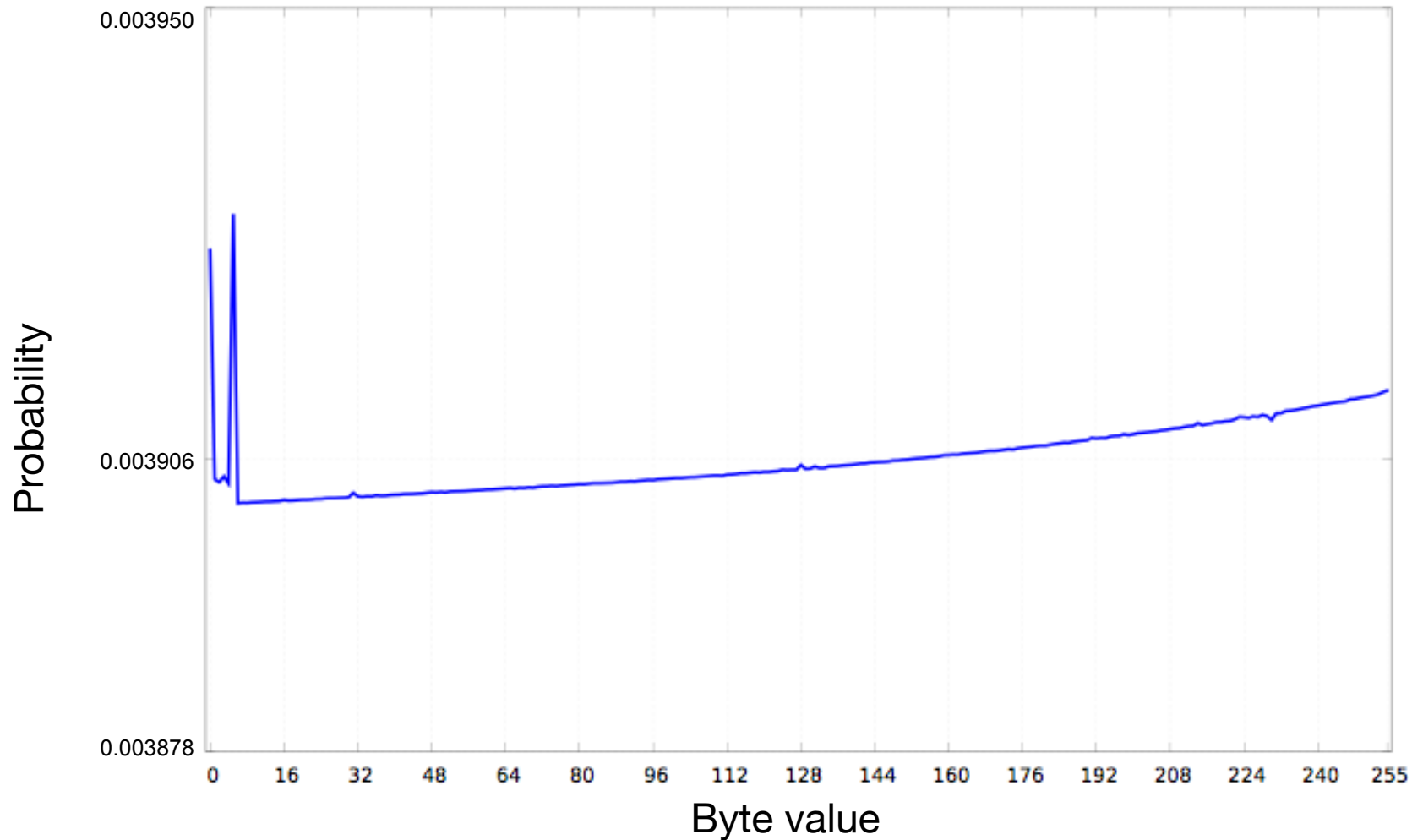
Keystream Distribution at Position 6
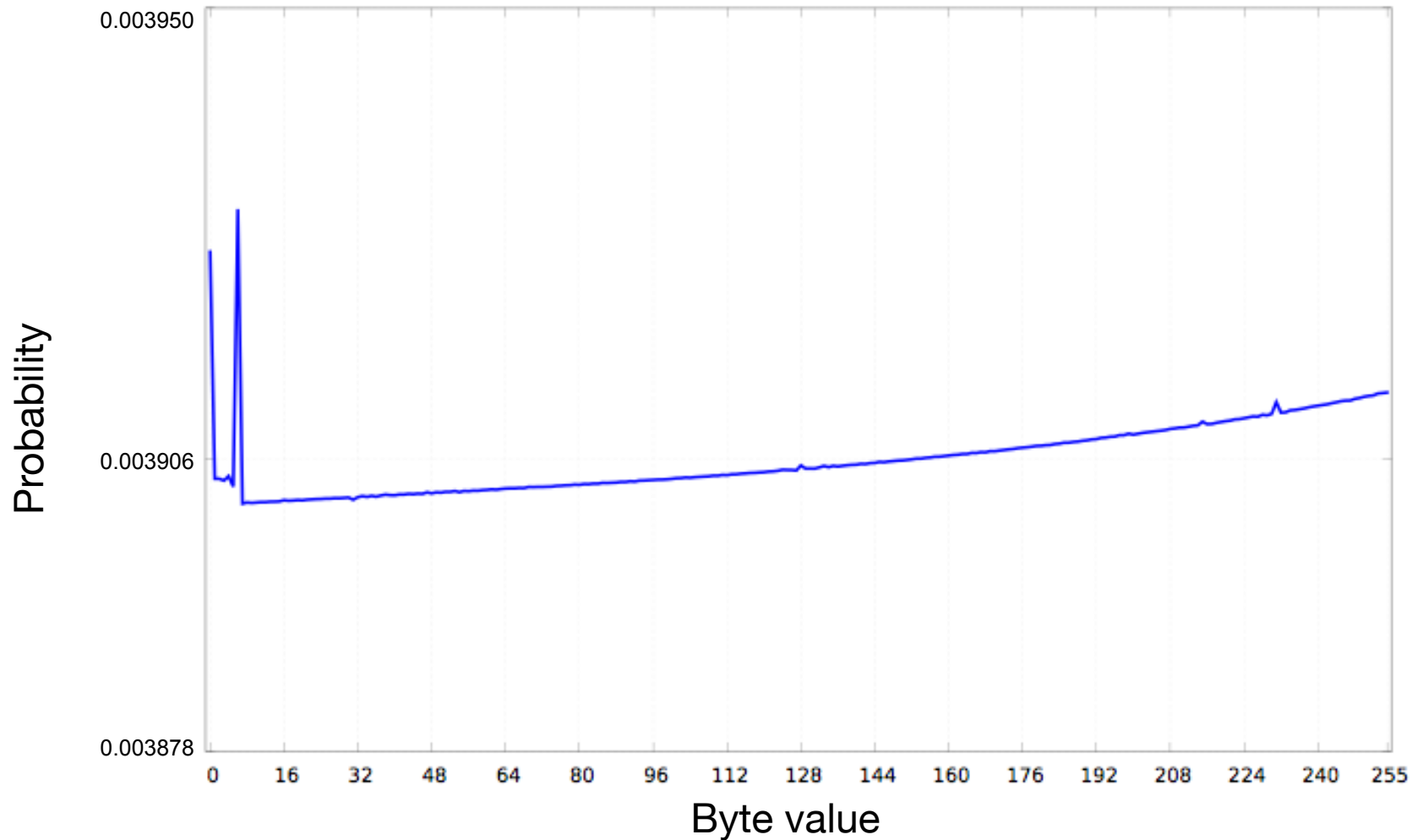
Keystream Distribution at Position 7
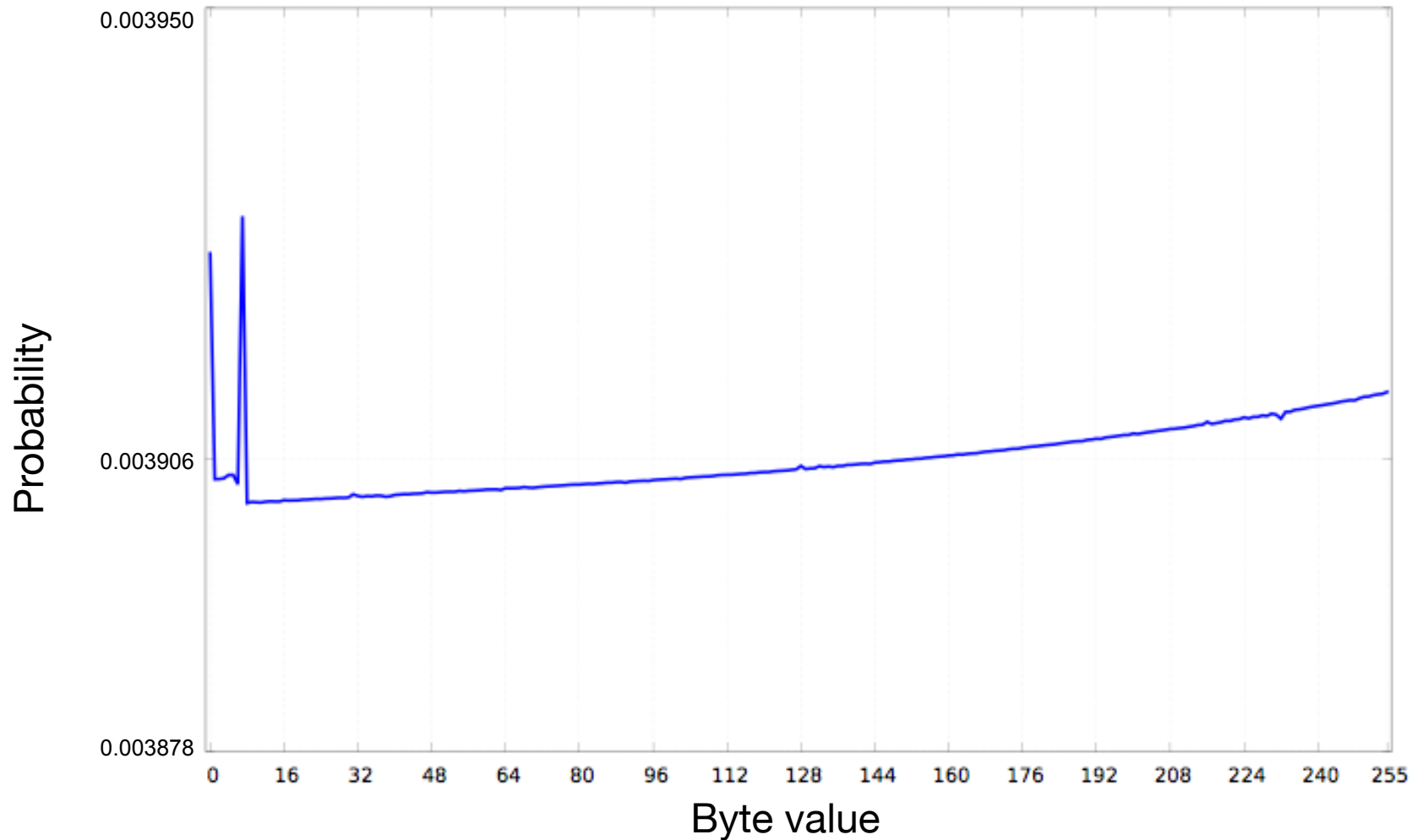
Keystream Distribution at Position 8
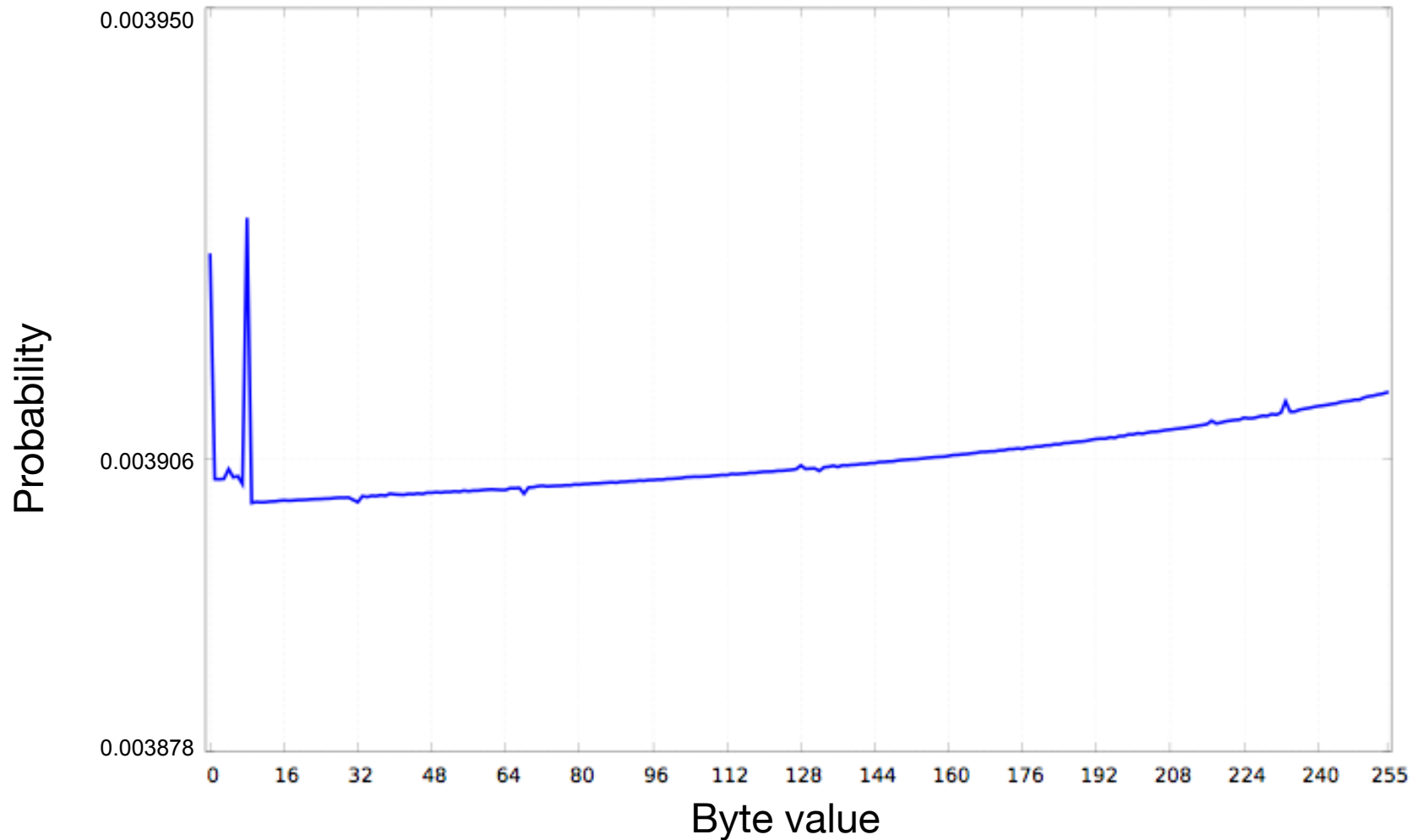
Keystream Distribution at Position 9
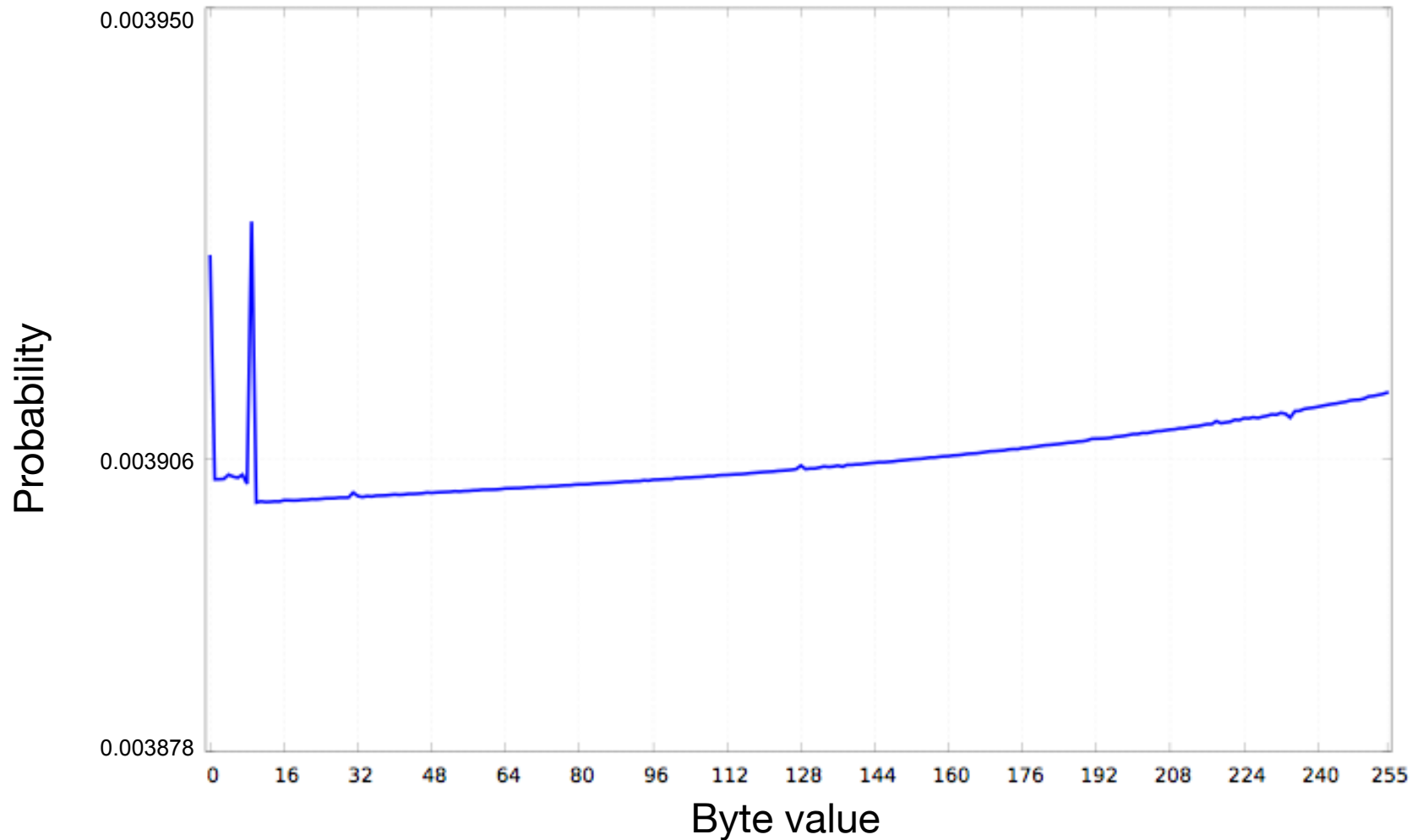
Keystream Distribution at Position 10
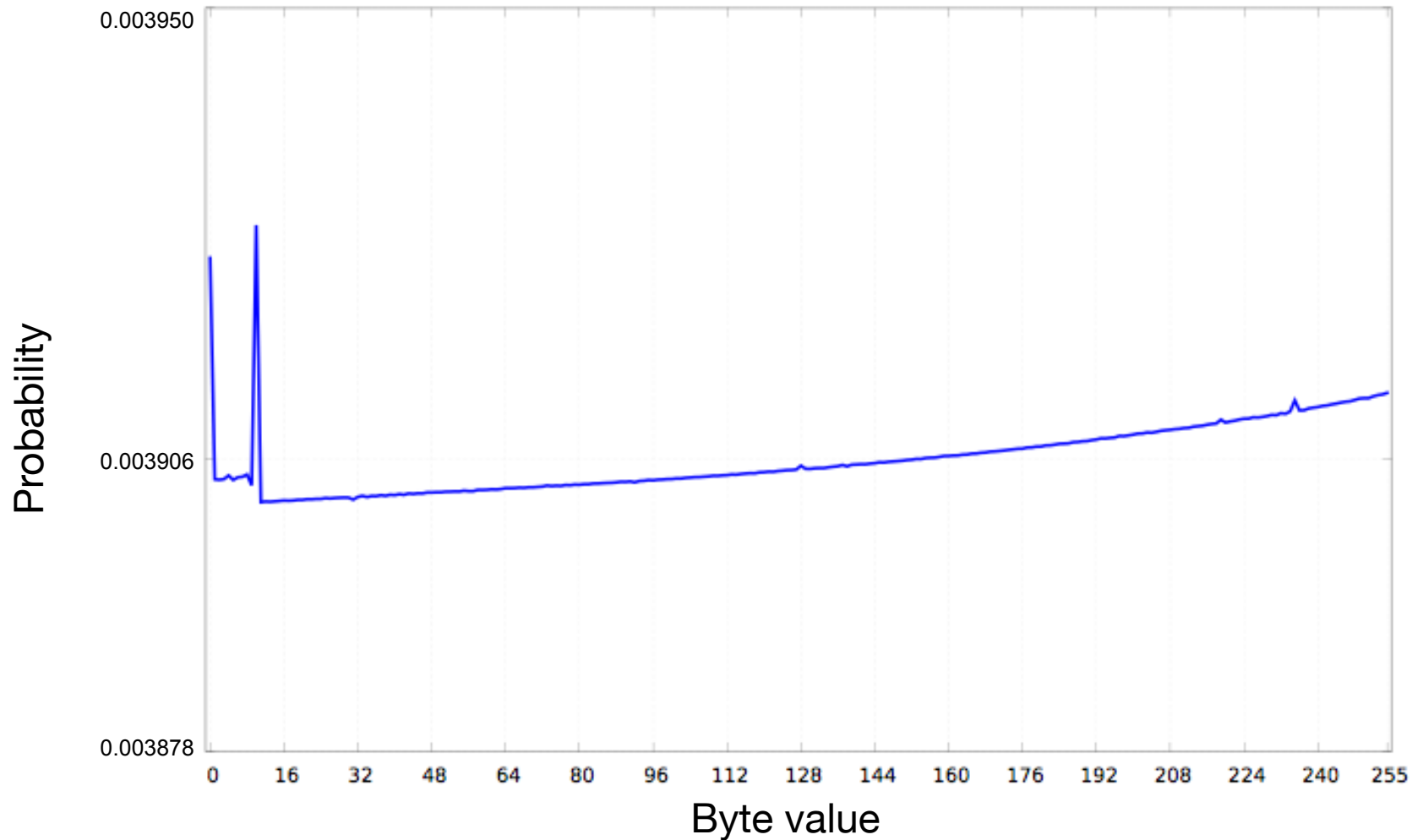
Keystream Distribution at Position 11
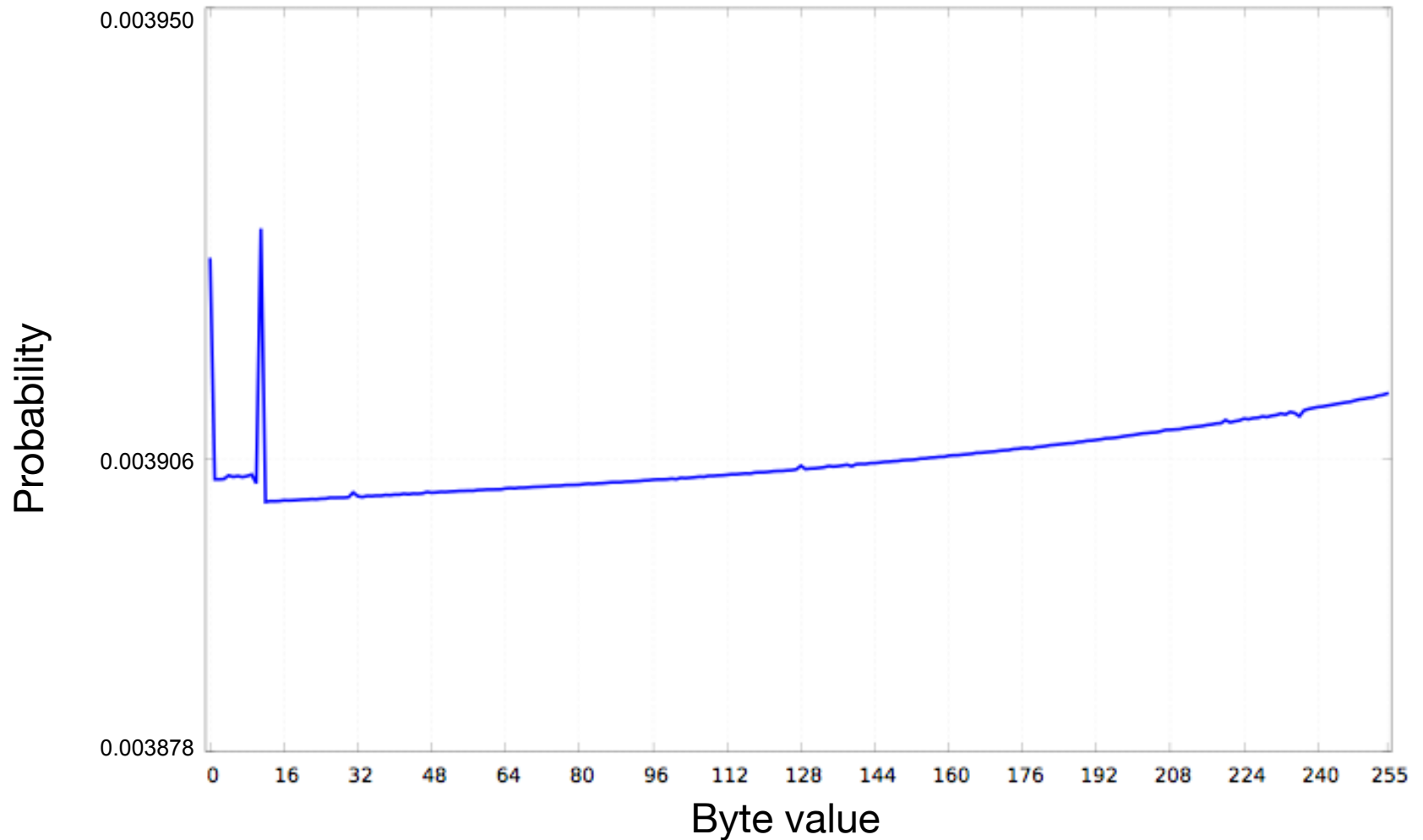
Keystream Distribution at Position 12

Keystream Distribution at Position 13
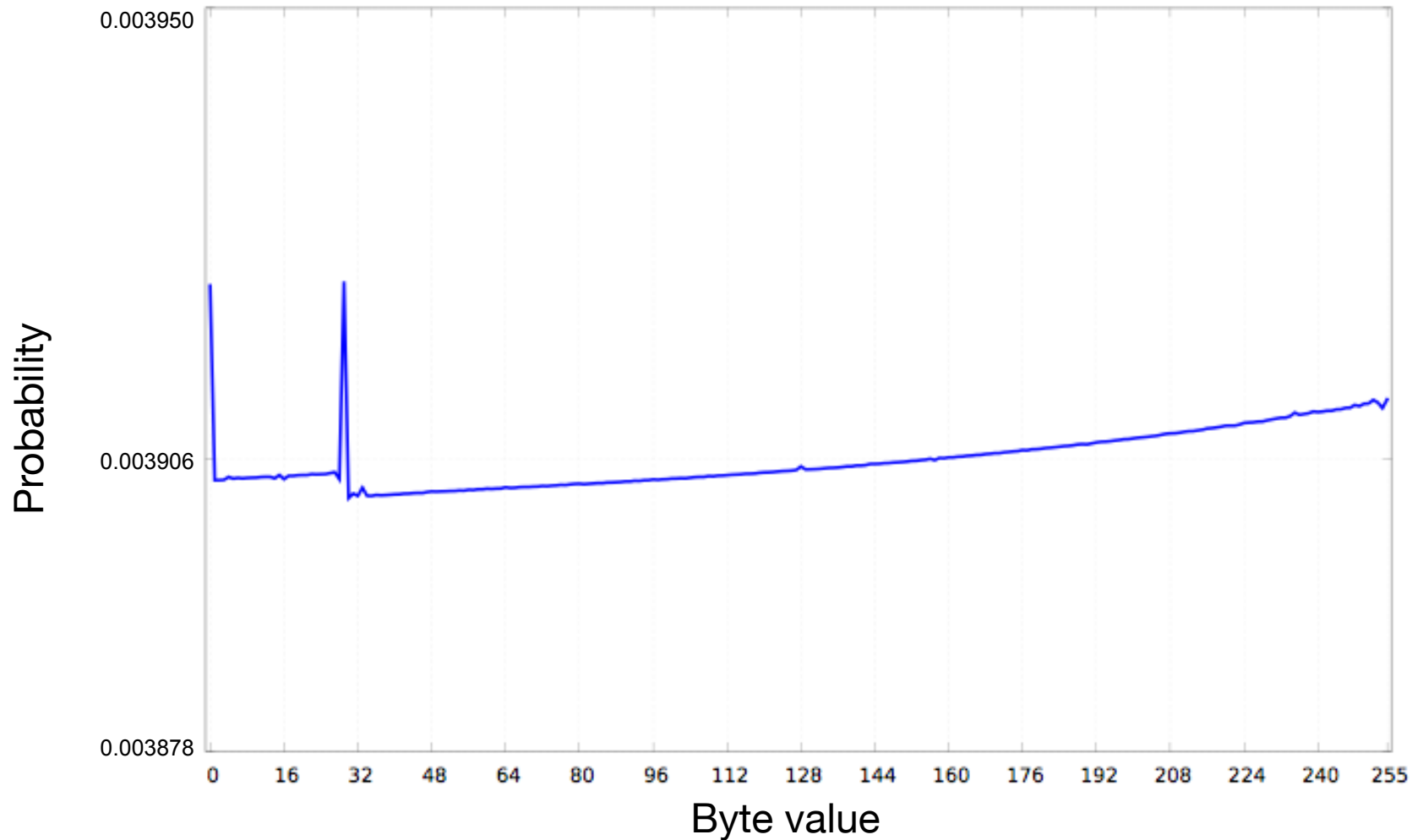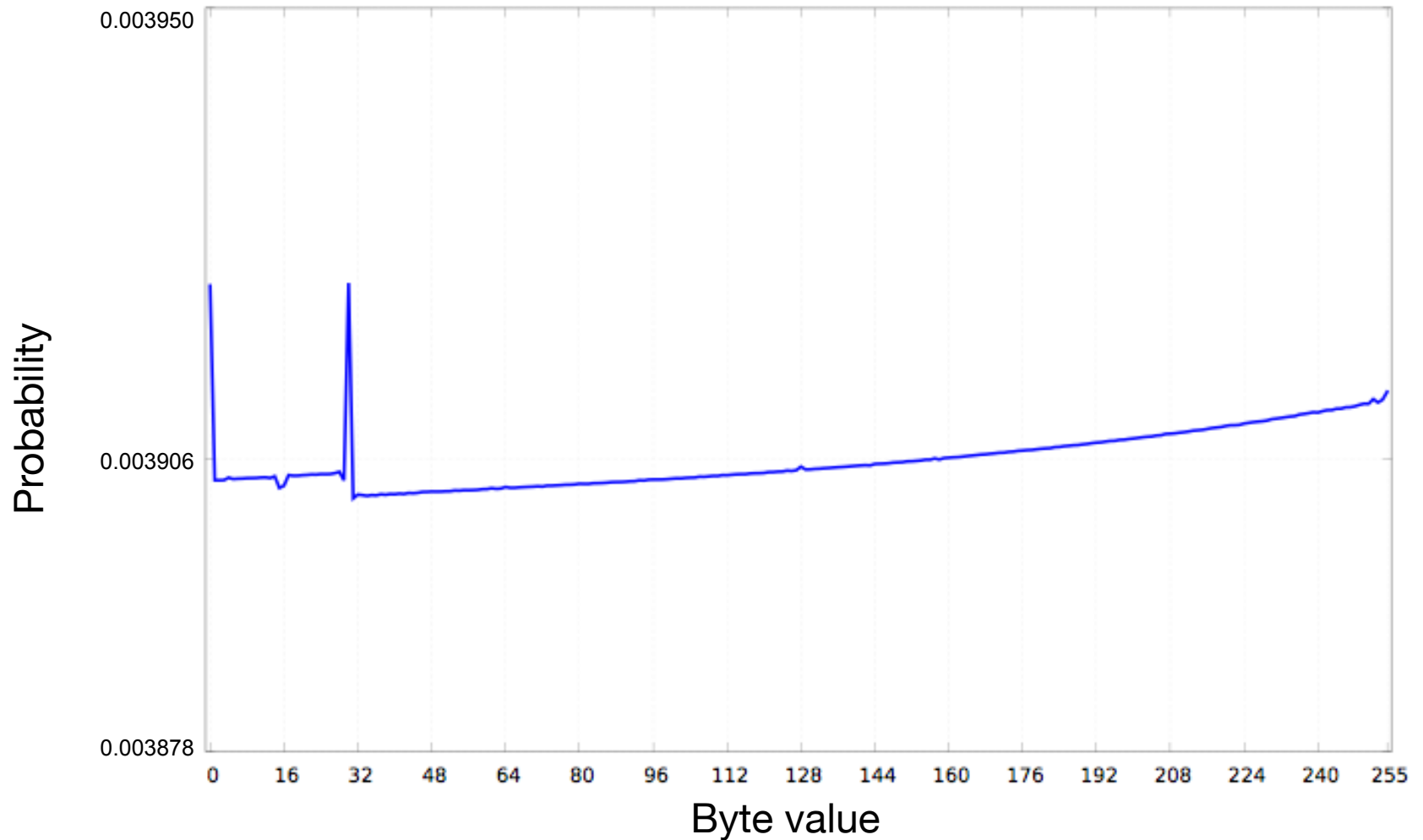

Probability vs. Byte value
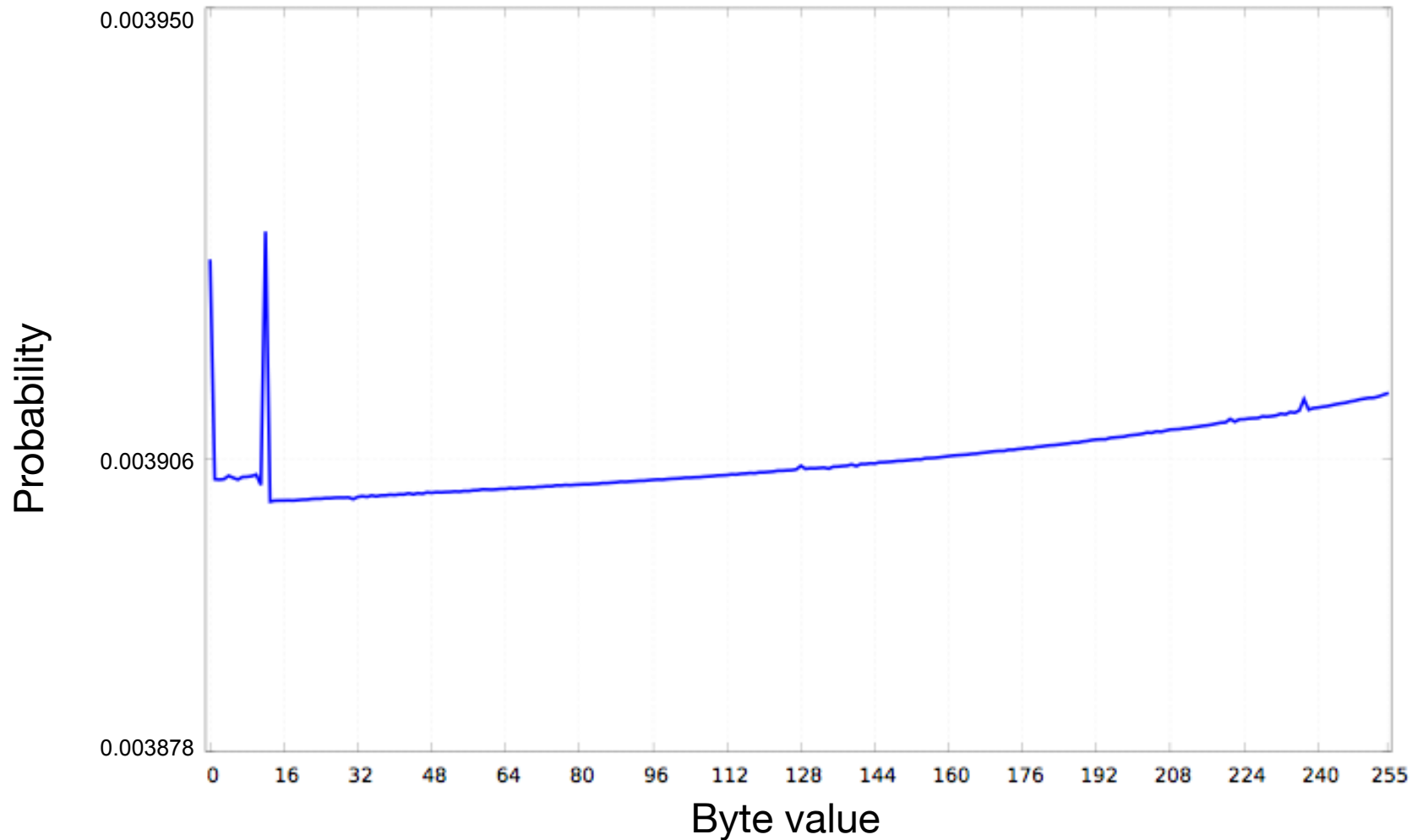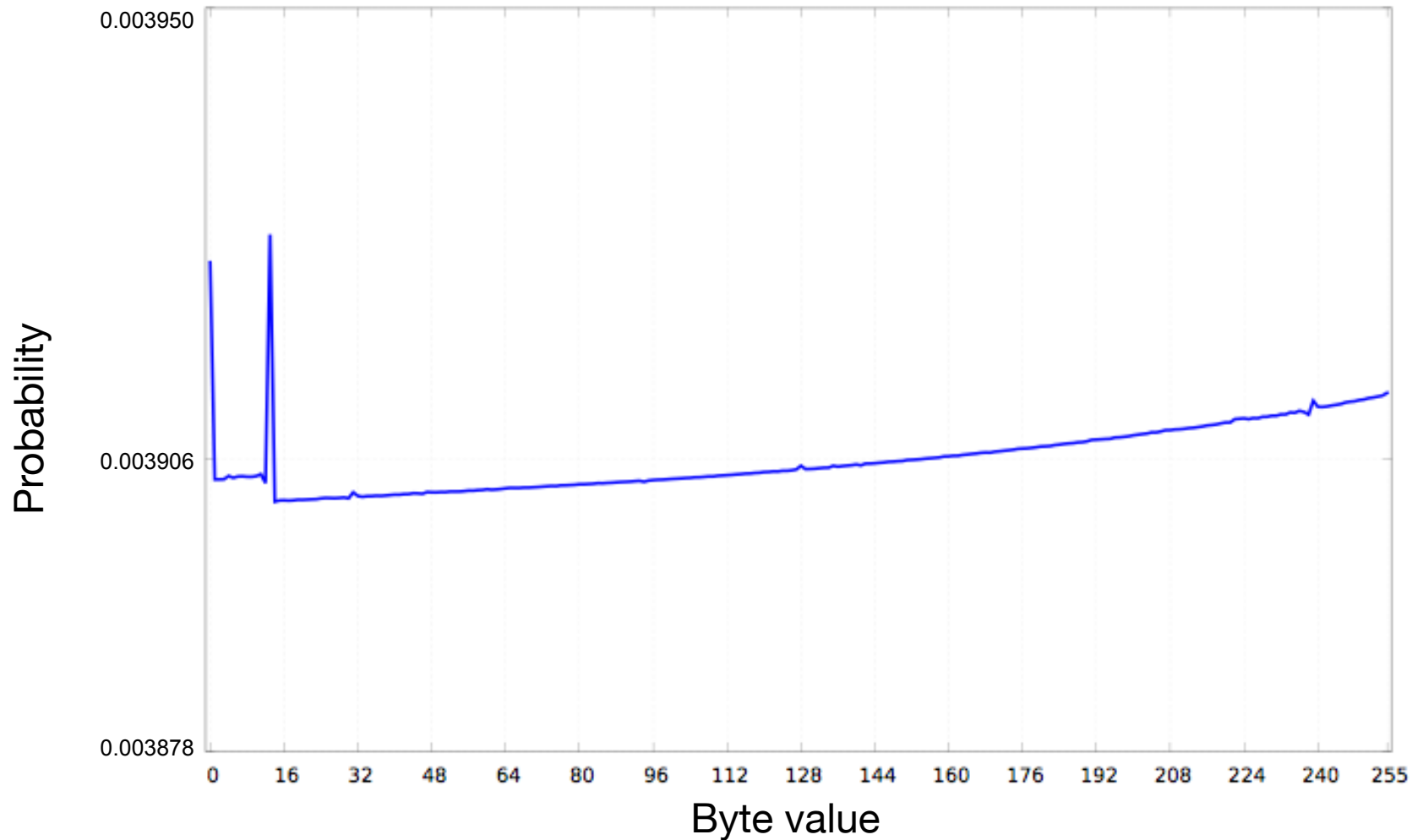
Keystream Distribution at Position 14
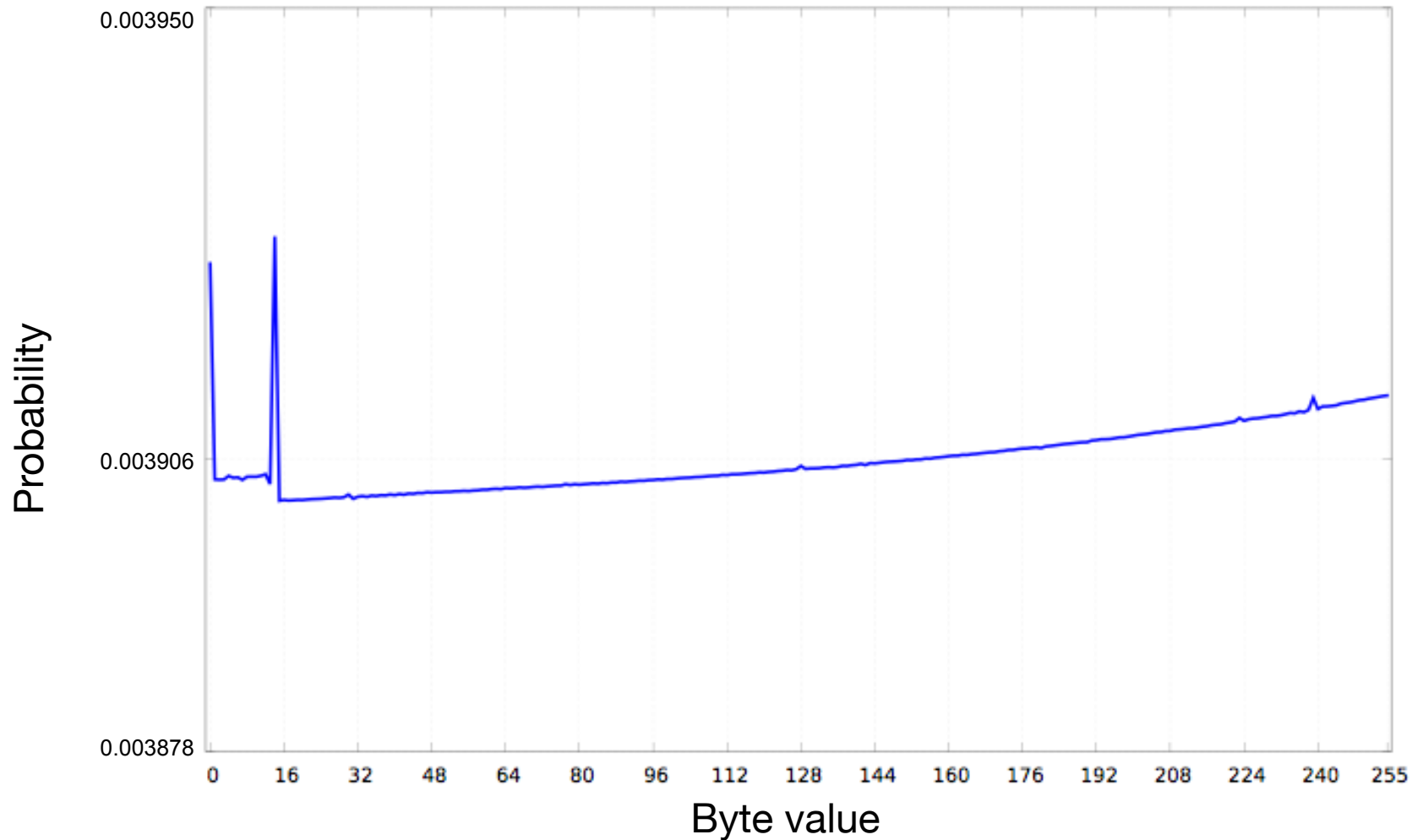
Keystream Distribution at Position 15
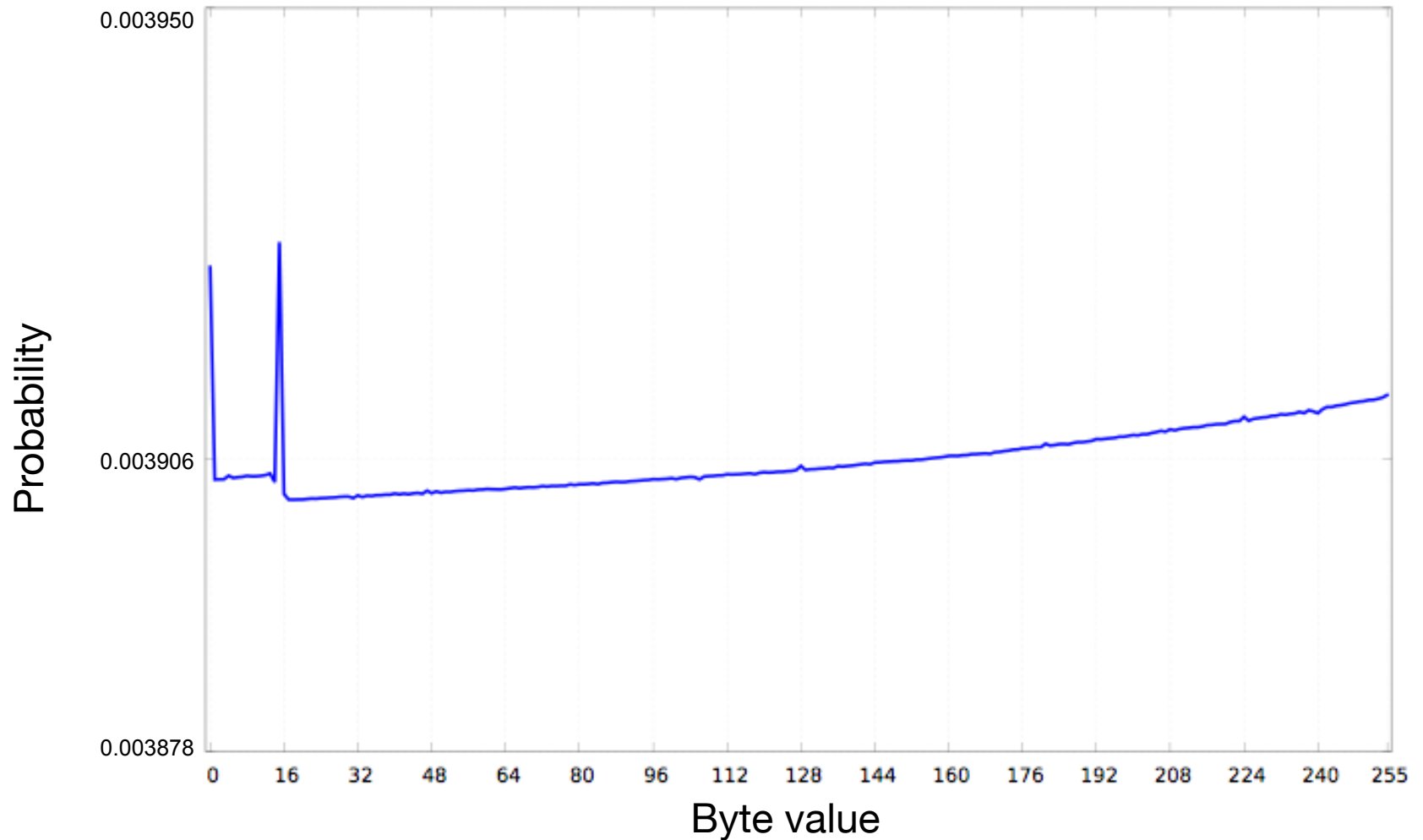
Keystream Distribution at Position 16
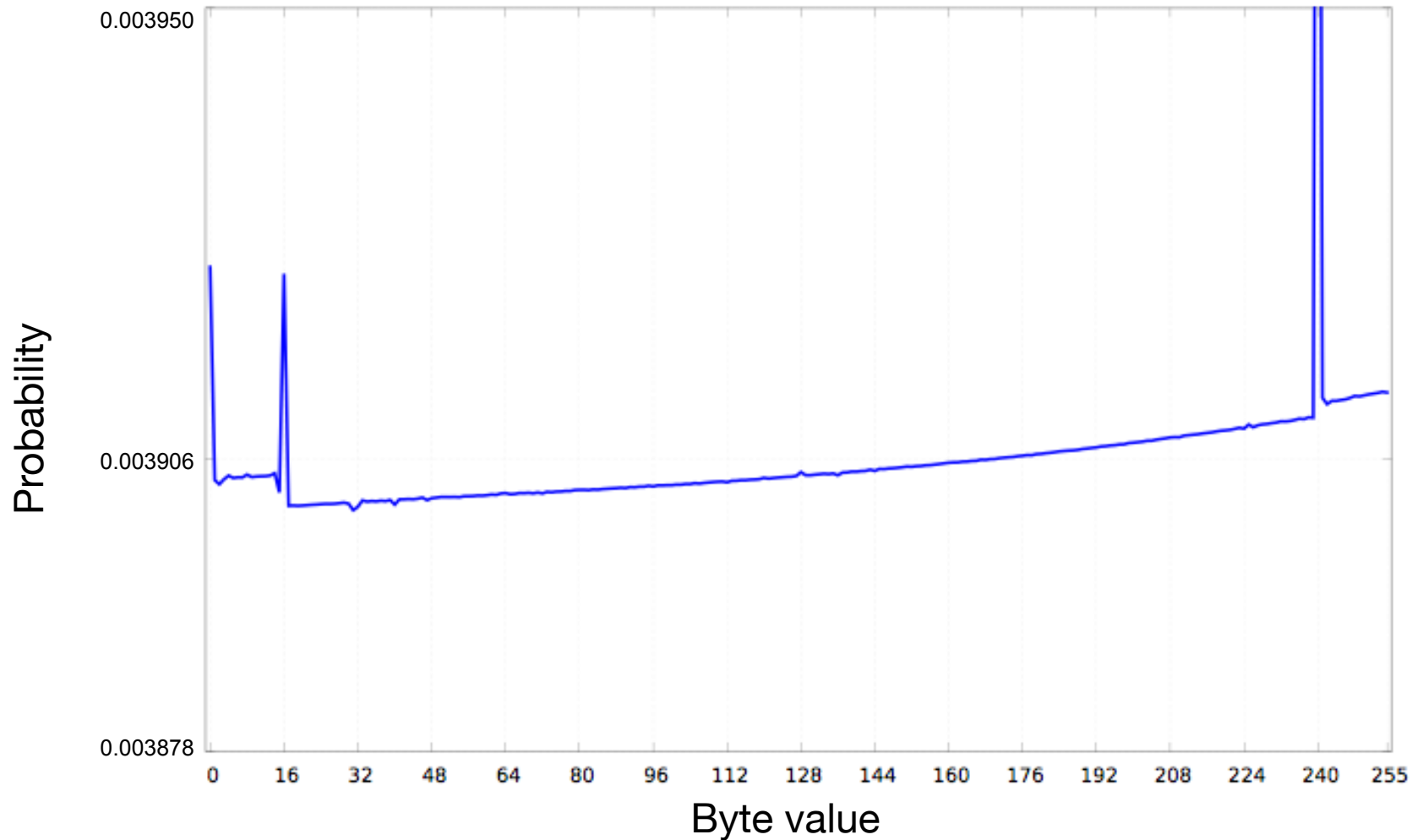
Keystream Distribution at Position 17

Keystream Distribution at Position 18

Probability vs. Byte value

STUDENTS-HUB.com
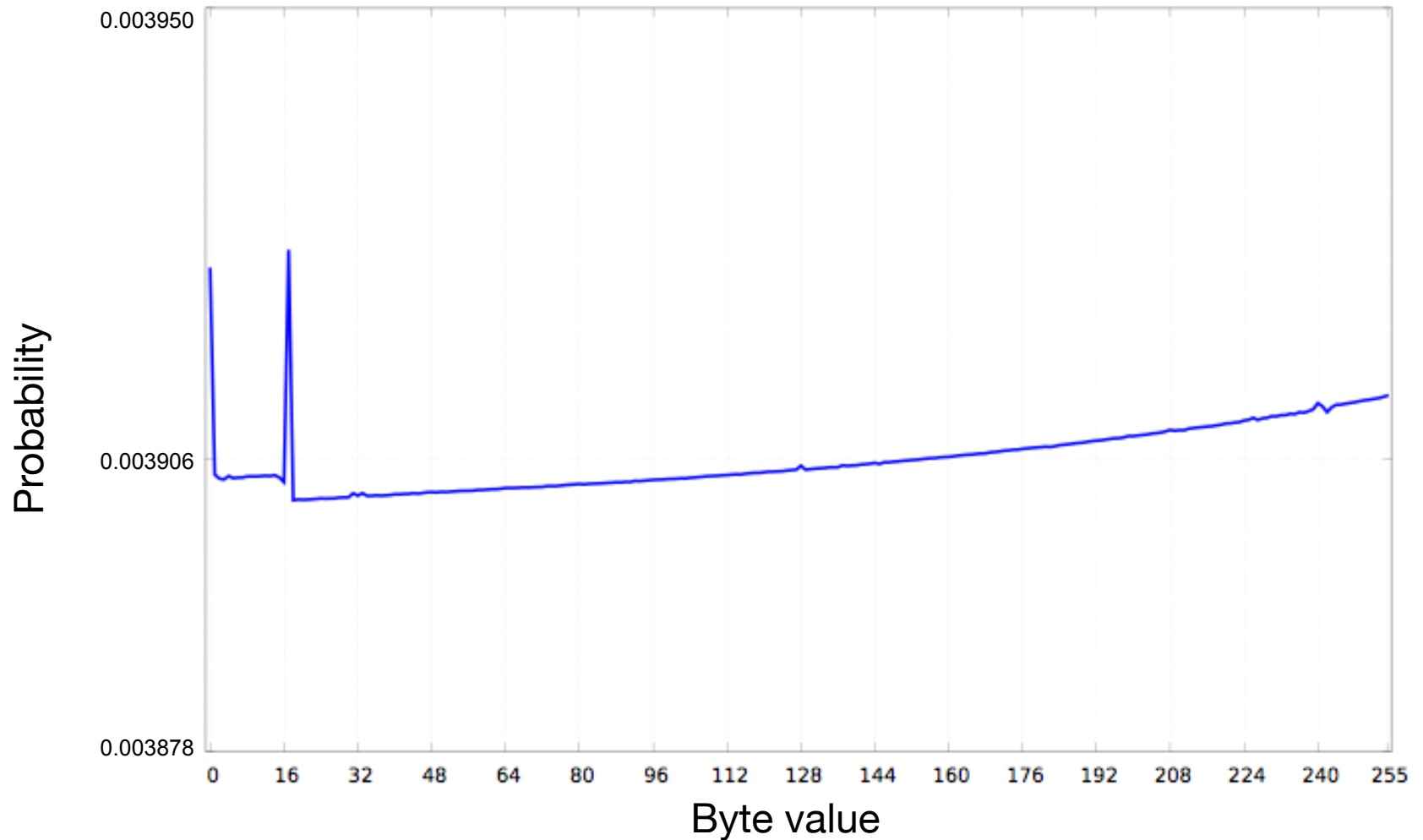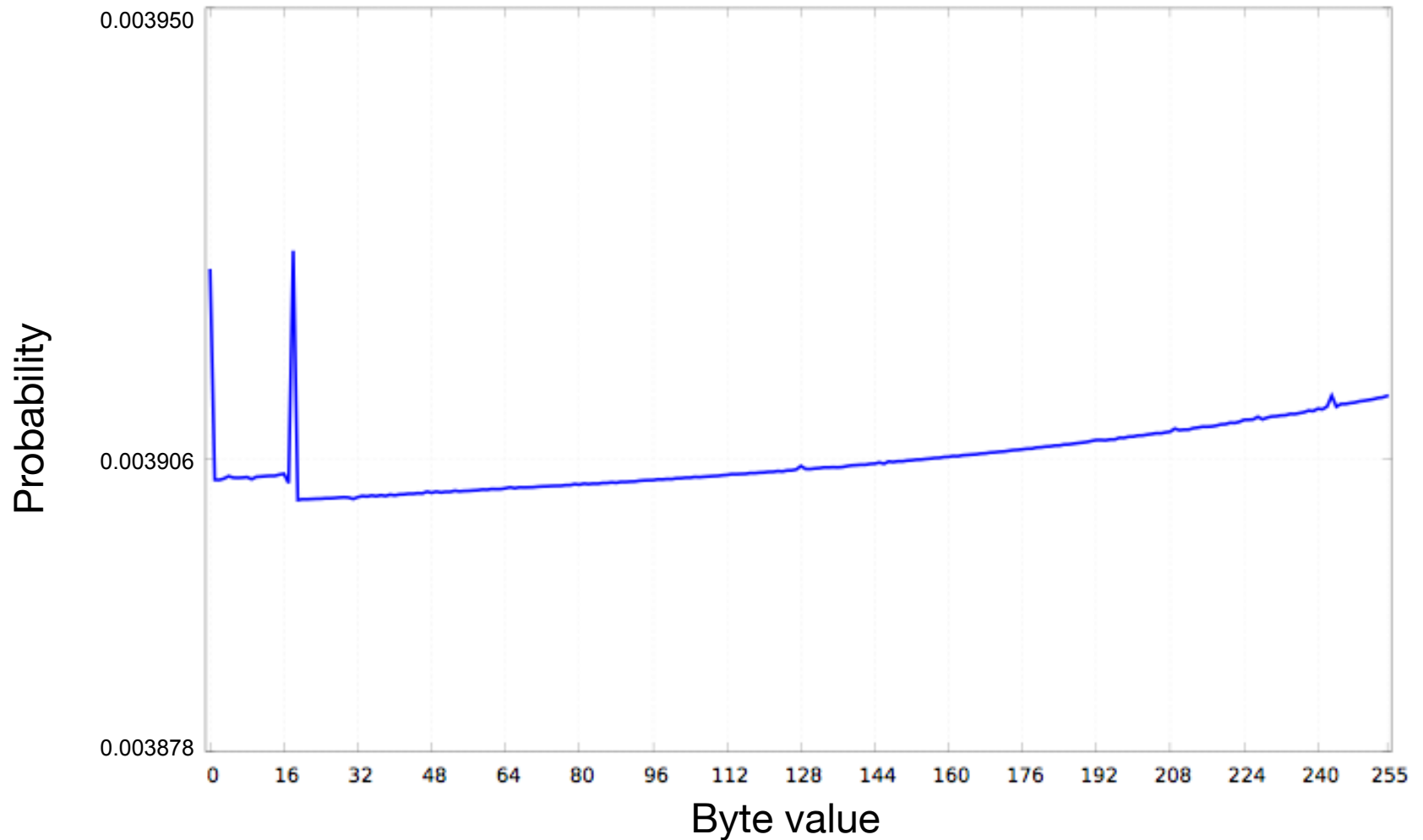
Keystream Distribution at Position 19
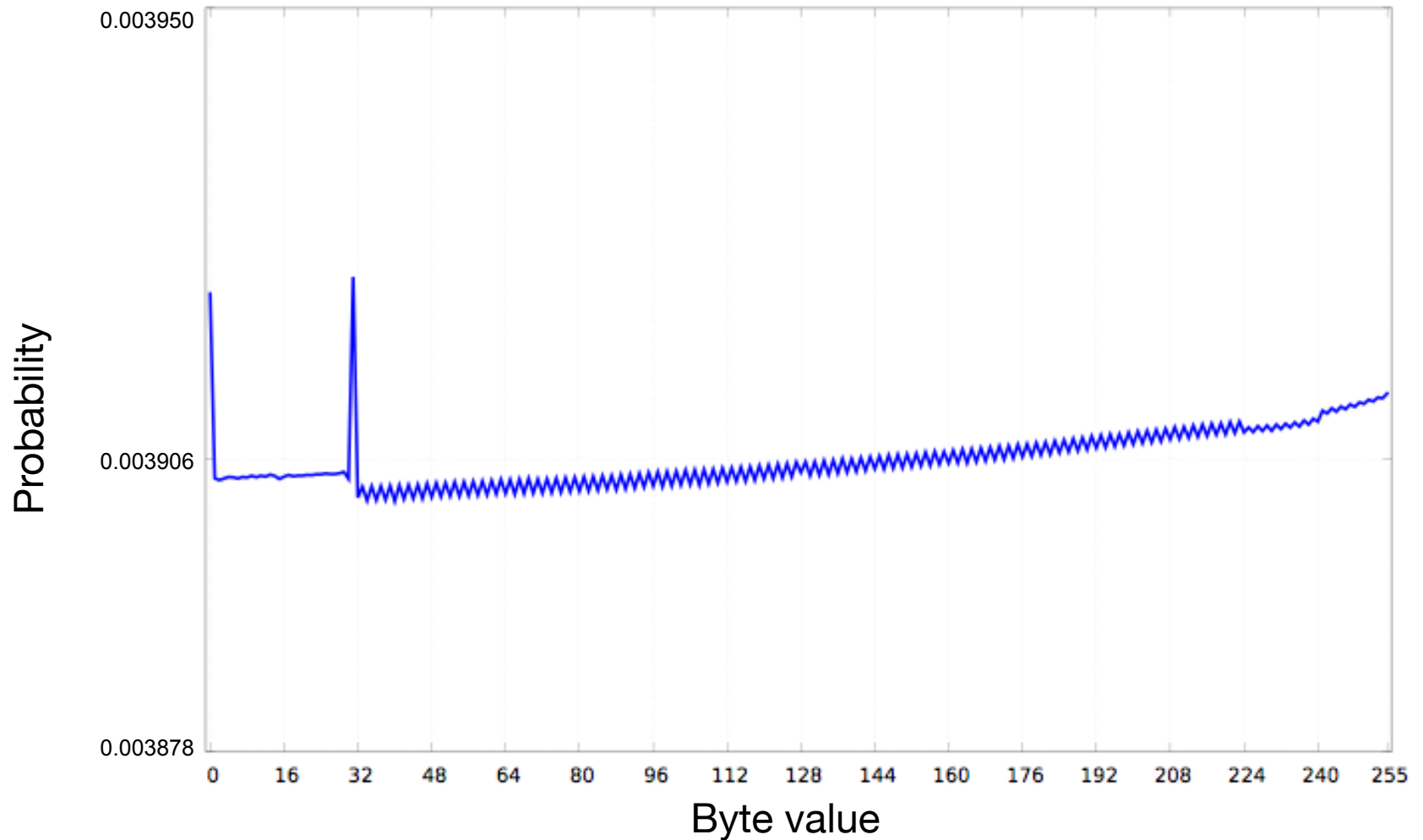
STUDENTS-HUB.com
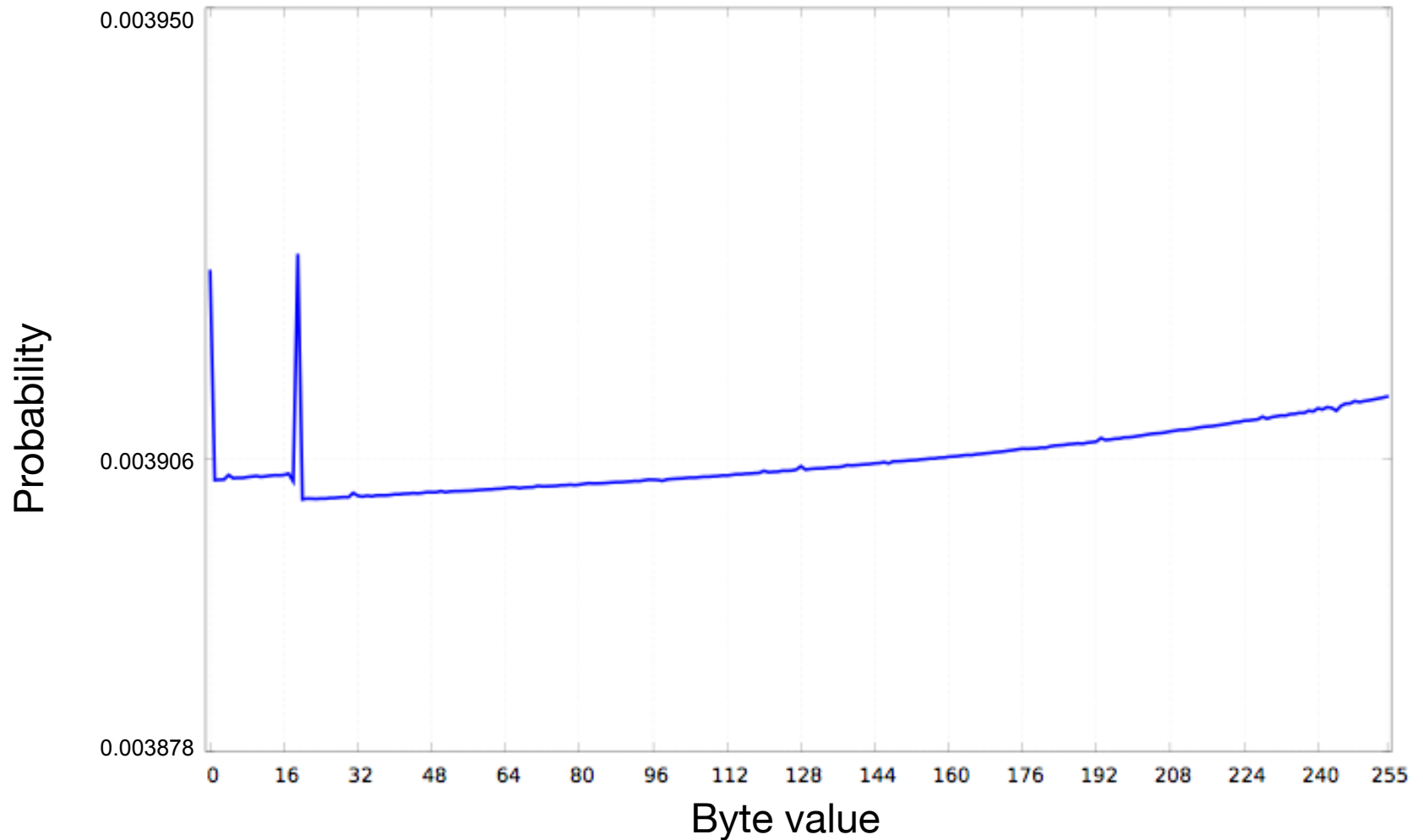
Keystream Distribution at Position 20
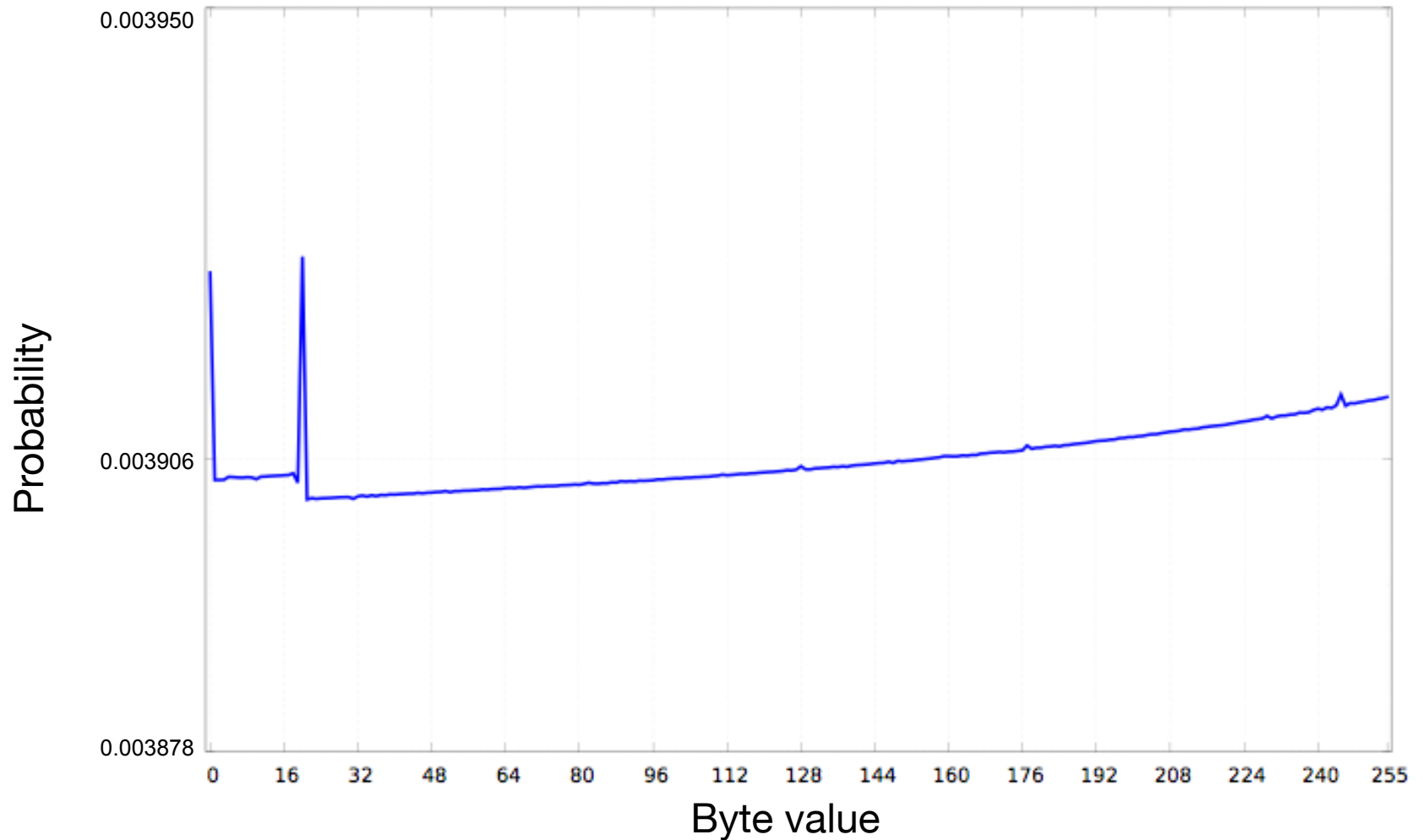
Keystream Distribution at Position 21
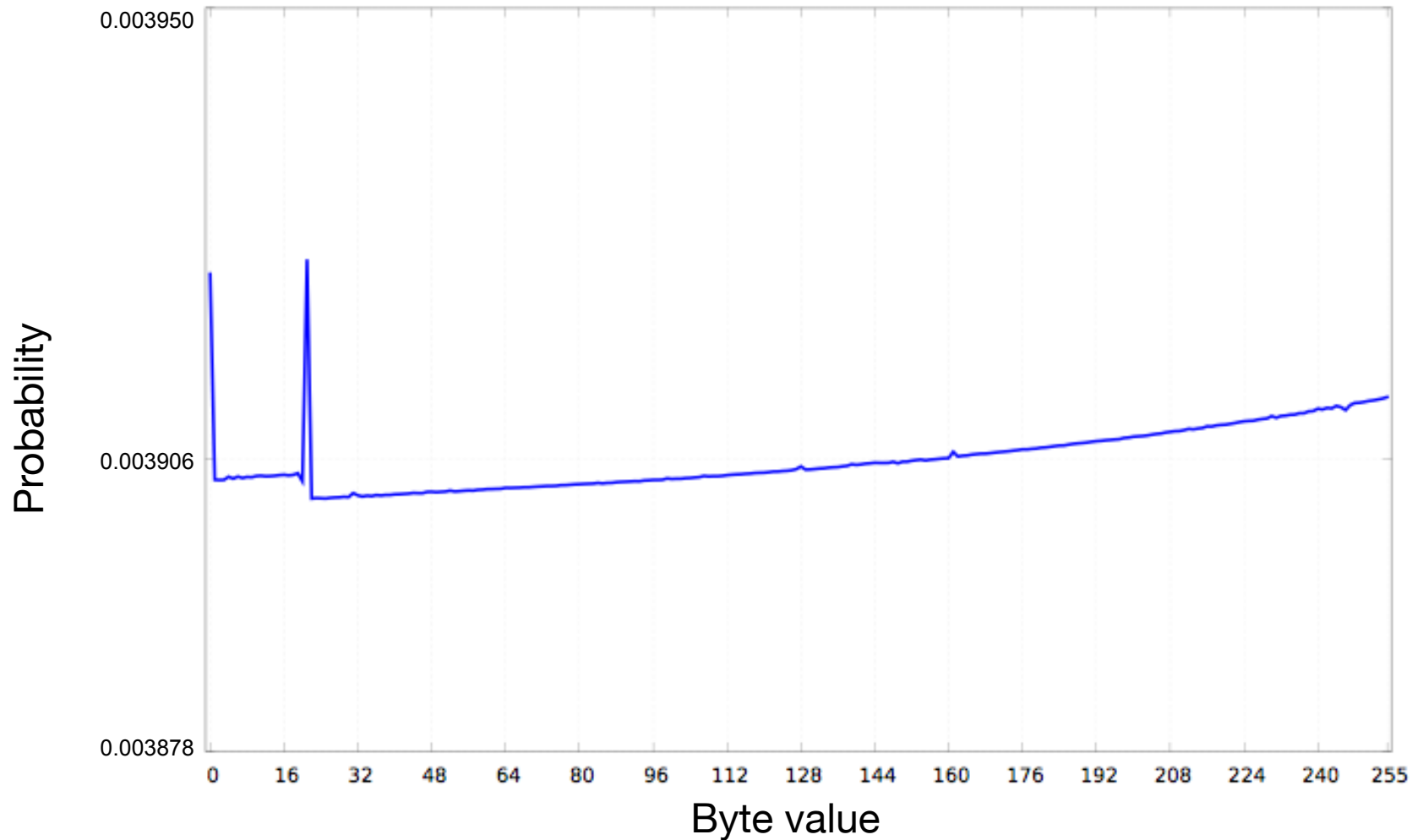
Keystream Distribution at Position 22

Keystream Distribution at Position 23

STUDENTS-HUB.com

# Keystream Distribution at Position 24

Keystream Distribution at Position 25

Keystream Distribution at Position 26

STUDENTS-HUB.com

Keystream Distribution at Position 27

Keystream Distribution at
Position 28

STUDENTS-HUB.com

Keystream Distribution at Position 29

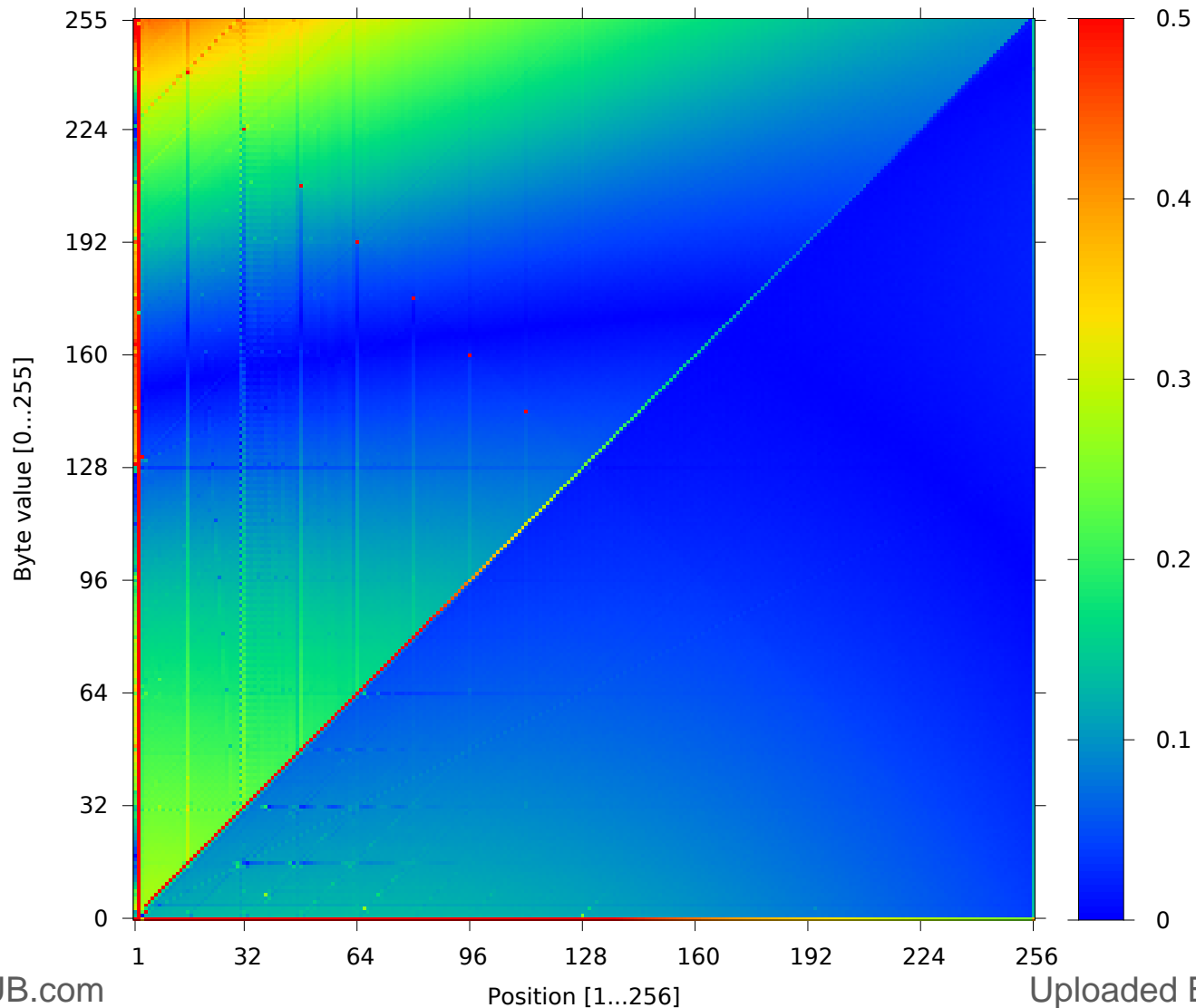# Keystream Distribution at Position 30

Keystream Distribution at Position 31

Keystream Distribution at
# Position 32

All the Biases

180
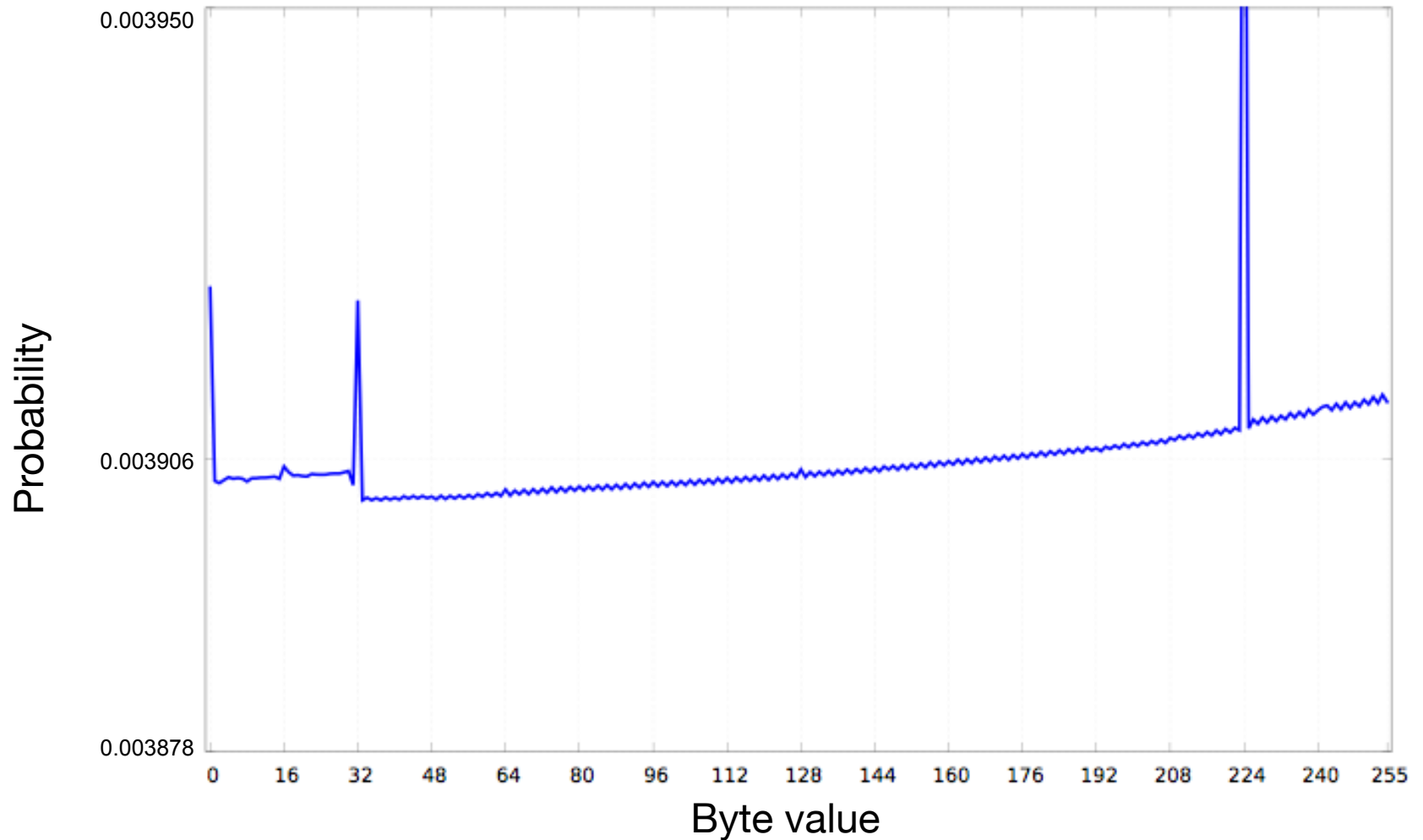
Plaintext Recovery for TLS-RC4

- Pretty picture, but where's the plaintext?

- 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.

  - Reusing the BEAST mechanism once more.
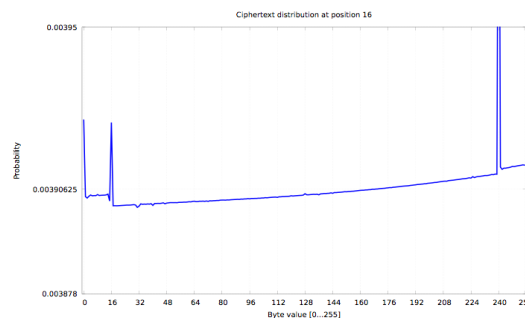
Plaintext Recovery Using Keystream Biases

Encryptions of fixed plaintext
under different keys

Plaintext candidate
byte p

r

C_1

C_2

C_3

$\vdots$

C_n

$p \oplus \square$

$p \oplus \square$

$p \oplus \square$

$\vdots$

$p \oplus \square$
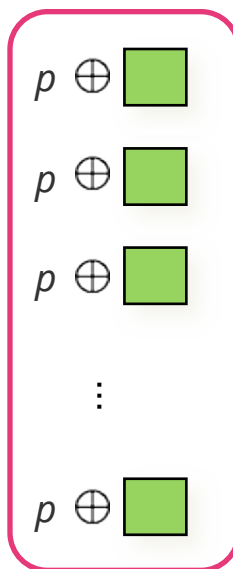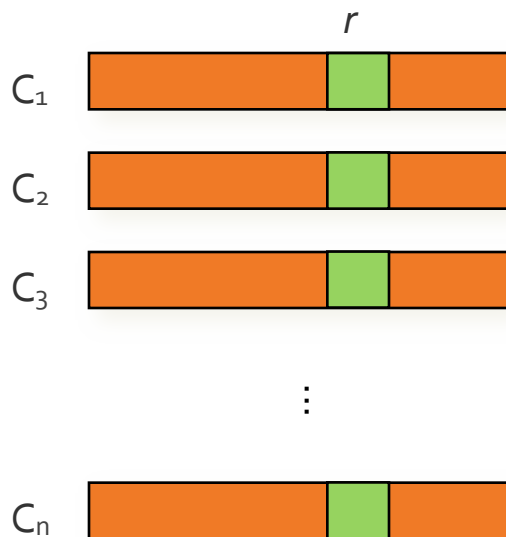
yields induced
distribution on
keystream byte Z_r

combine with known distribution

Likelihood of p being
correct plaintext byte

Recovery algorithm:
Compute most likely plaintext byte

Details of Statistical Analysis

Let c be the n-vector of ciphertext bytes in position r.

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

Bayes theorem:

$$\Pr[P=p \mid C=c] \; = \; \Pr[C=c \mid P=p].\Pr[P=p]/\Pr[C=c]$$

$$= \; \Pr[Z=c \oplus p \mid P=p].\Pr[P=p]/\Pr[C=c].$$
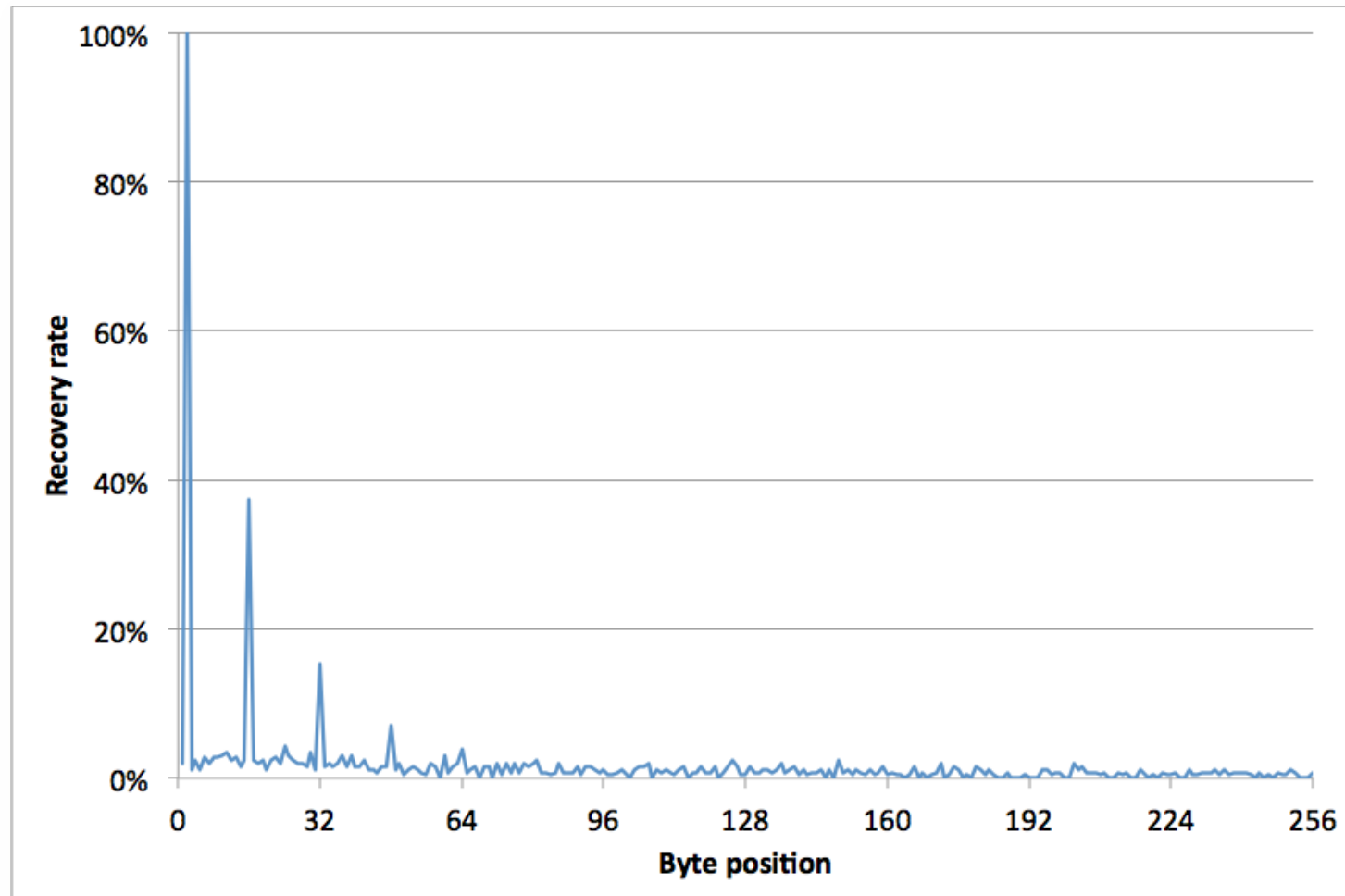
Assume $\Pr[P=p]$ is constant; $\Pr[C=c]$ is independent of the choice of p.

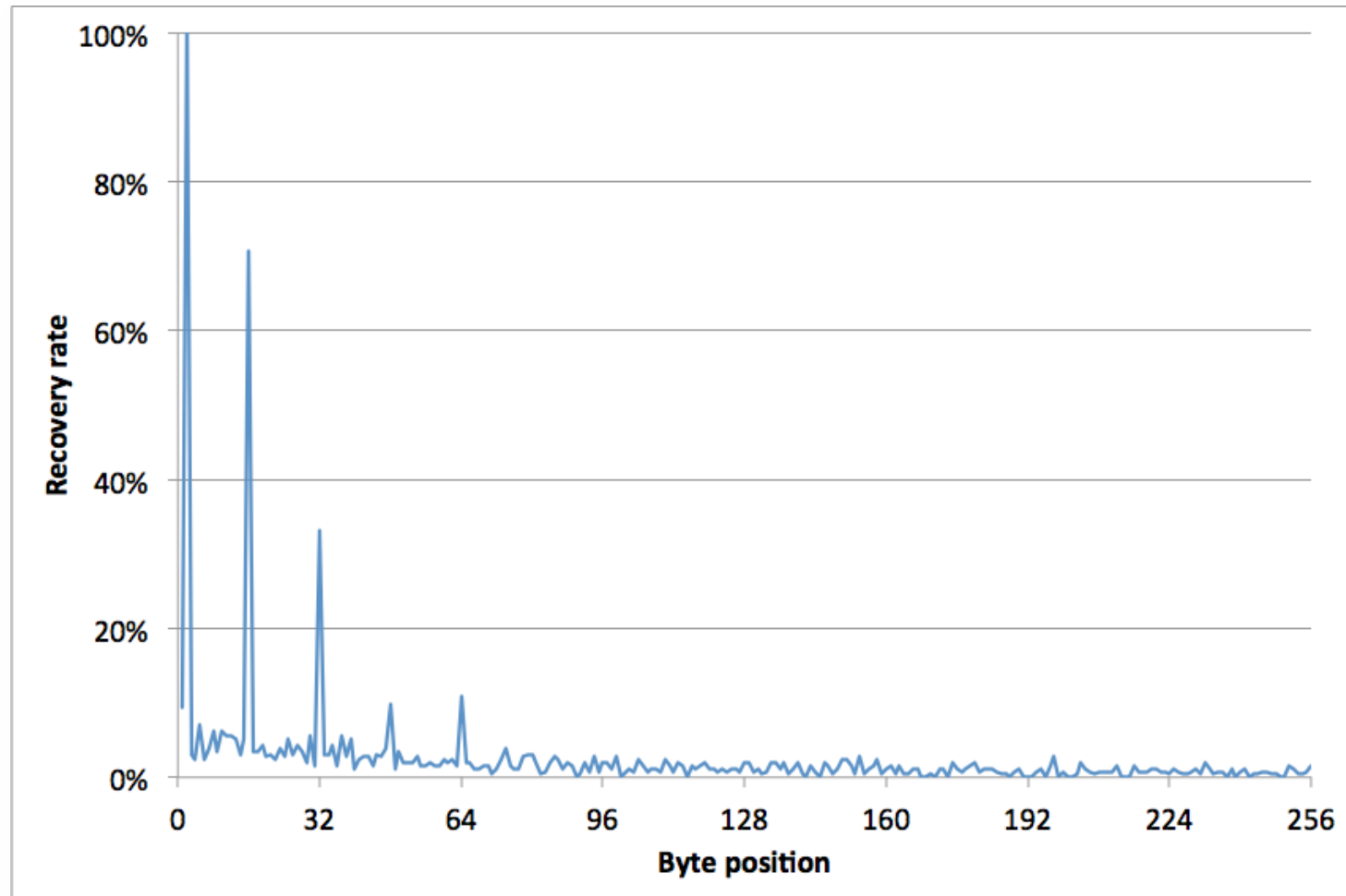To maximise $\Pr[P=p \mid C=c]$ over all choices of p, we simply need to maximise:

$$\Pr[Z=c \oplus p \mid P=p] \quad = \quad q_{00}^{n_{00}} \, q_{01}^{n_{01}} \ldots q_{ff}^{n_{ff}}$$

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

Success Probability 2²⁰ Sessions

STUDENTS-HUB.com

Success Probability 2²¹ Sessions

Success Probability 2²² Sessions

Success Probability 2²³ Sessions

Success Probability 2²⁴ Sessions

Success Probability 2²⁵ Sessions
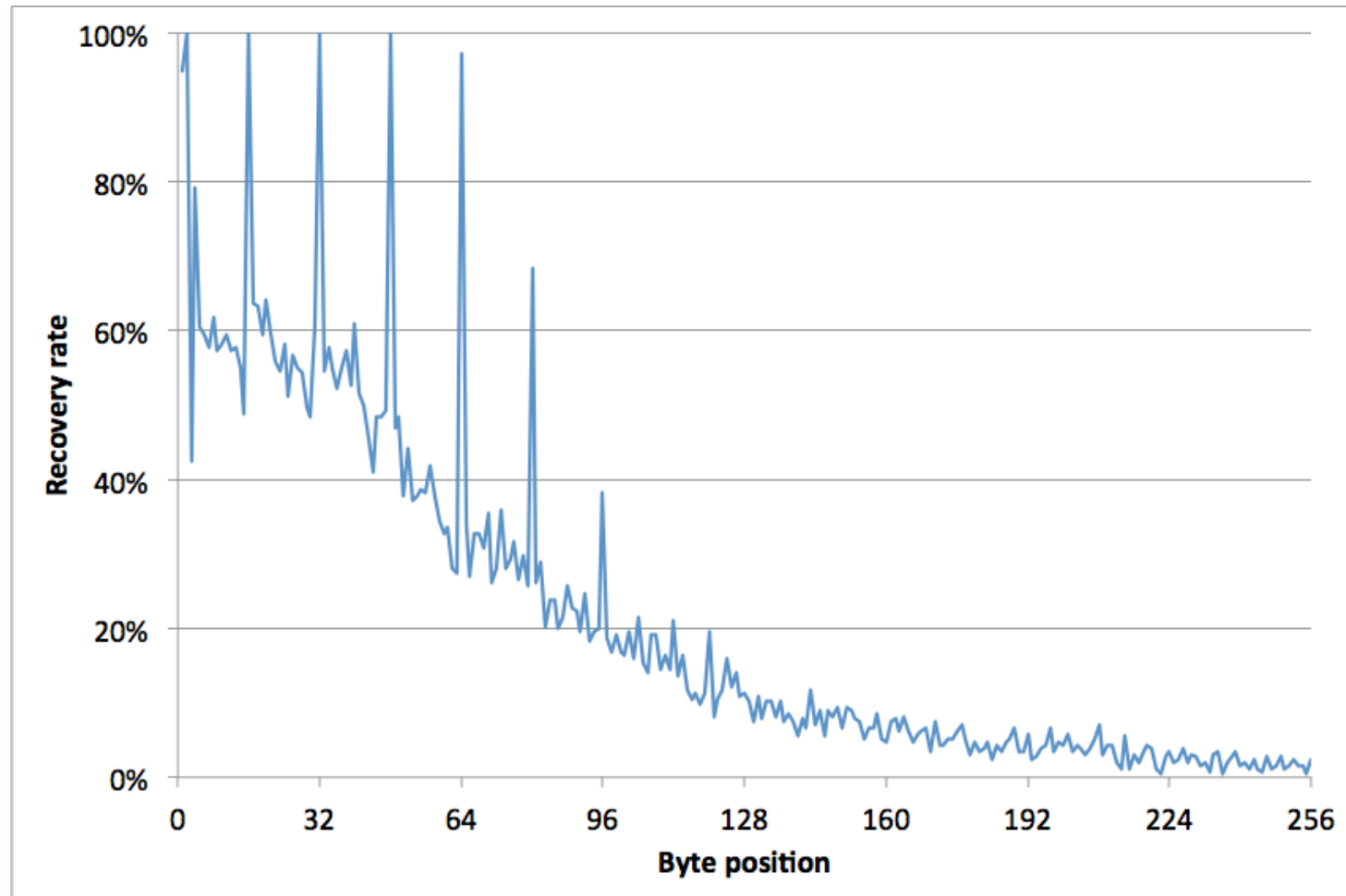
Success Probability 2²⁶ Sessions

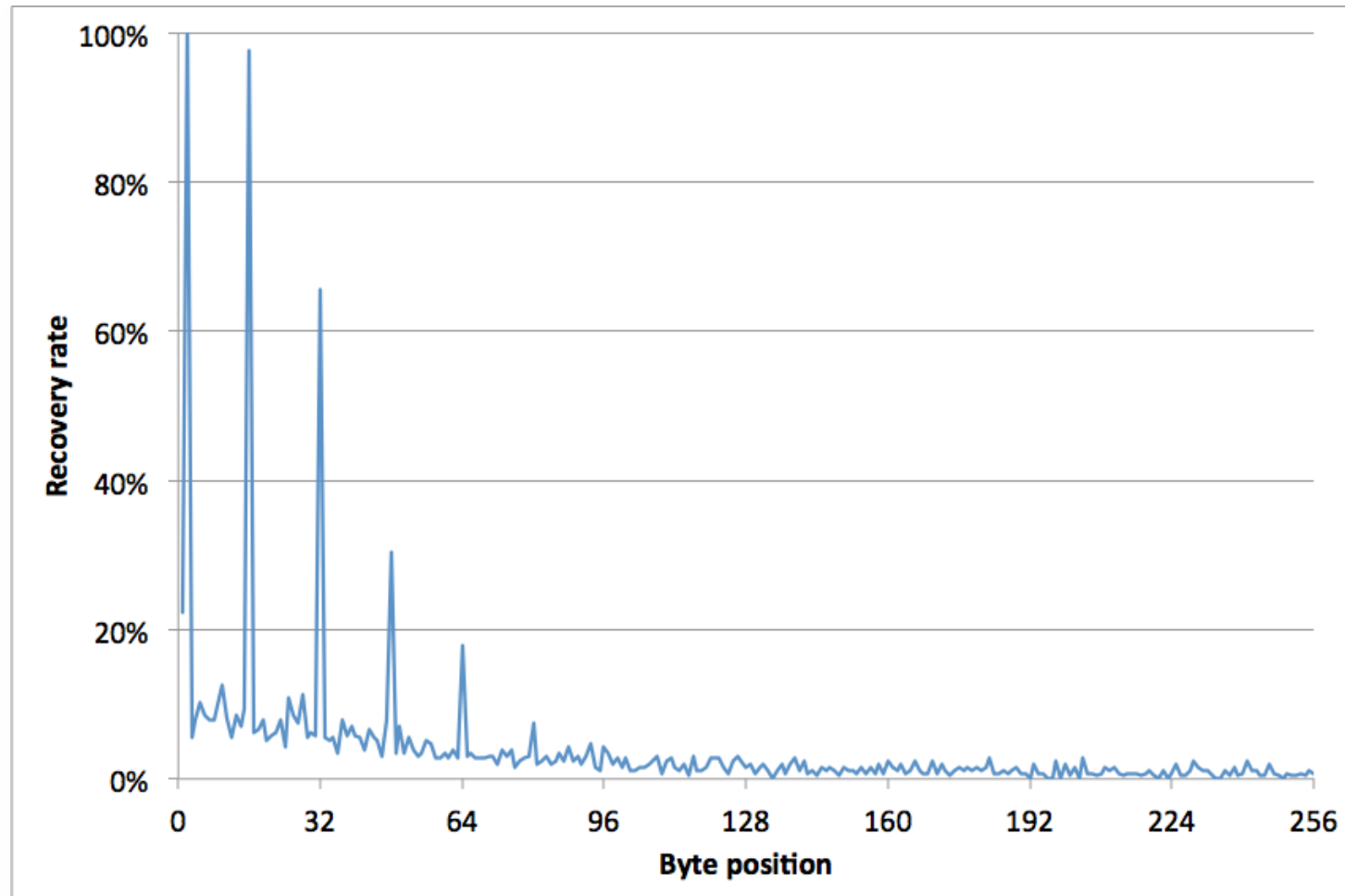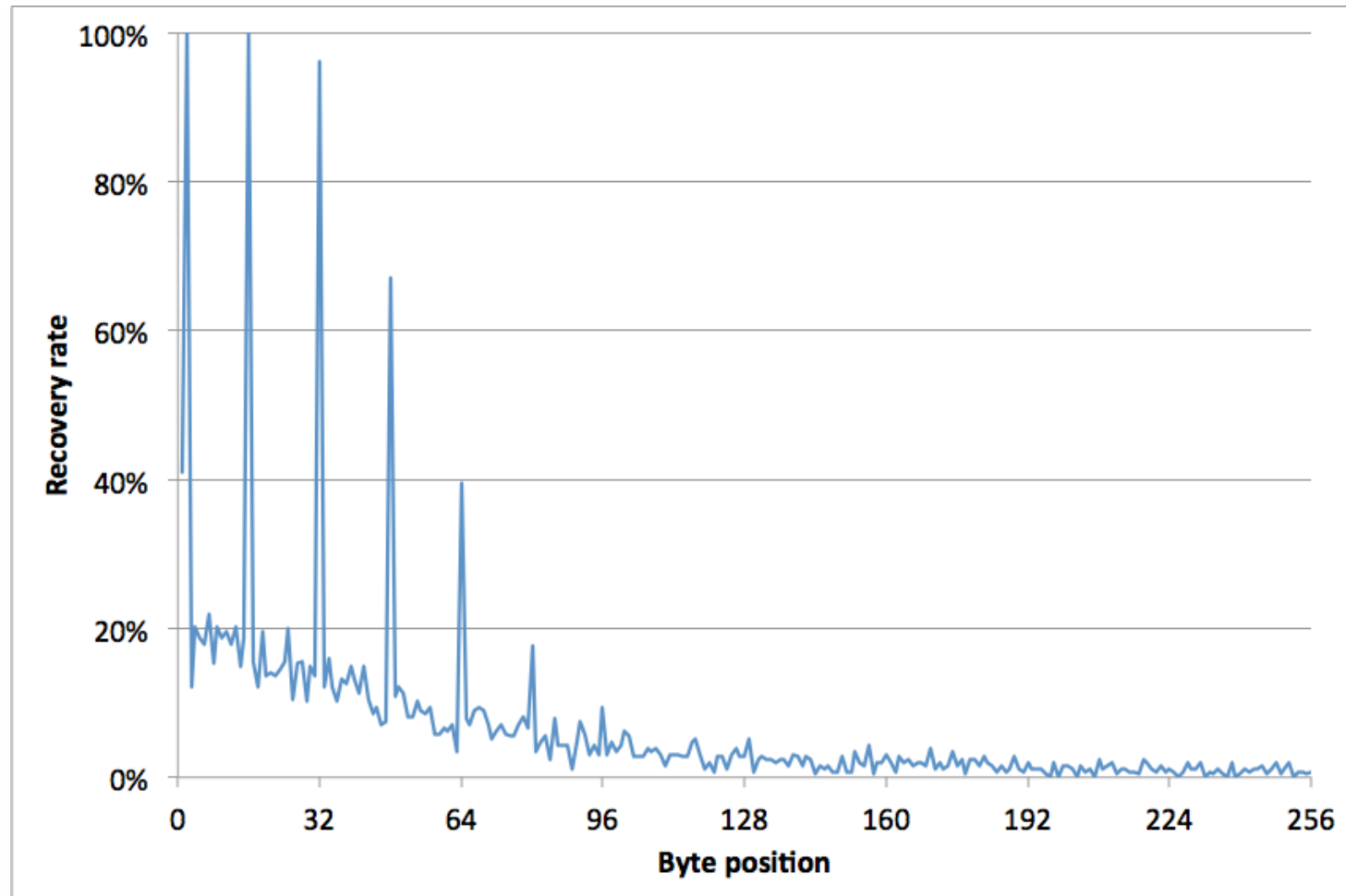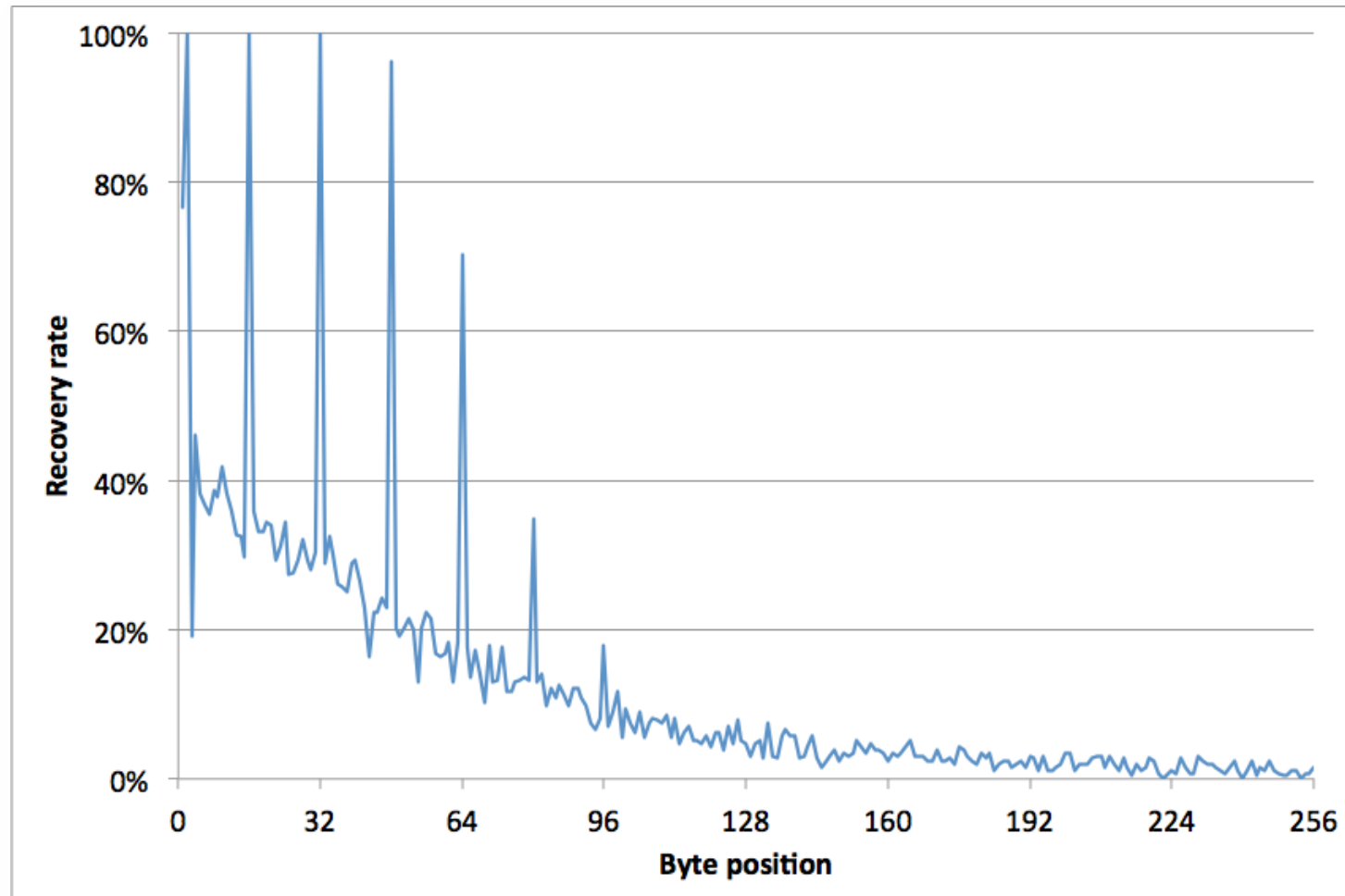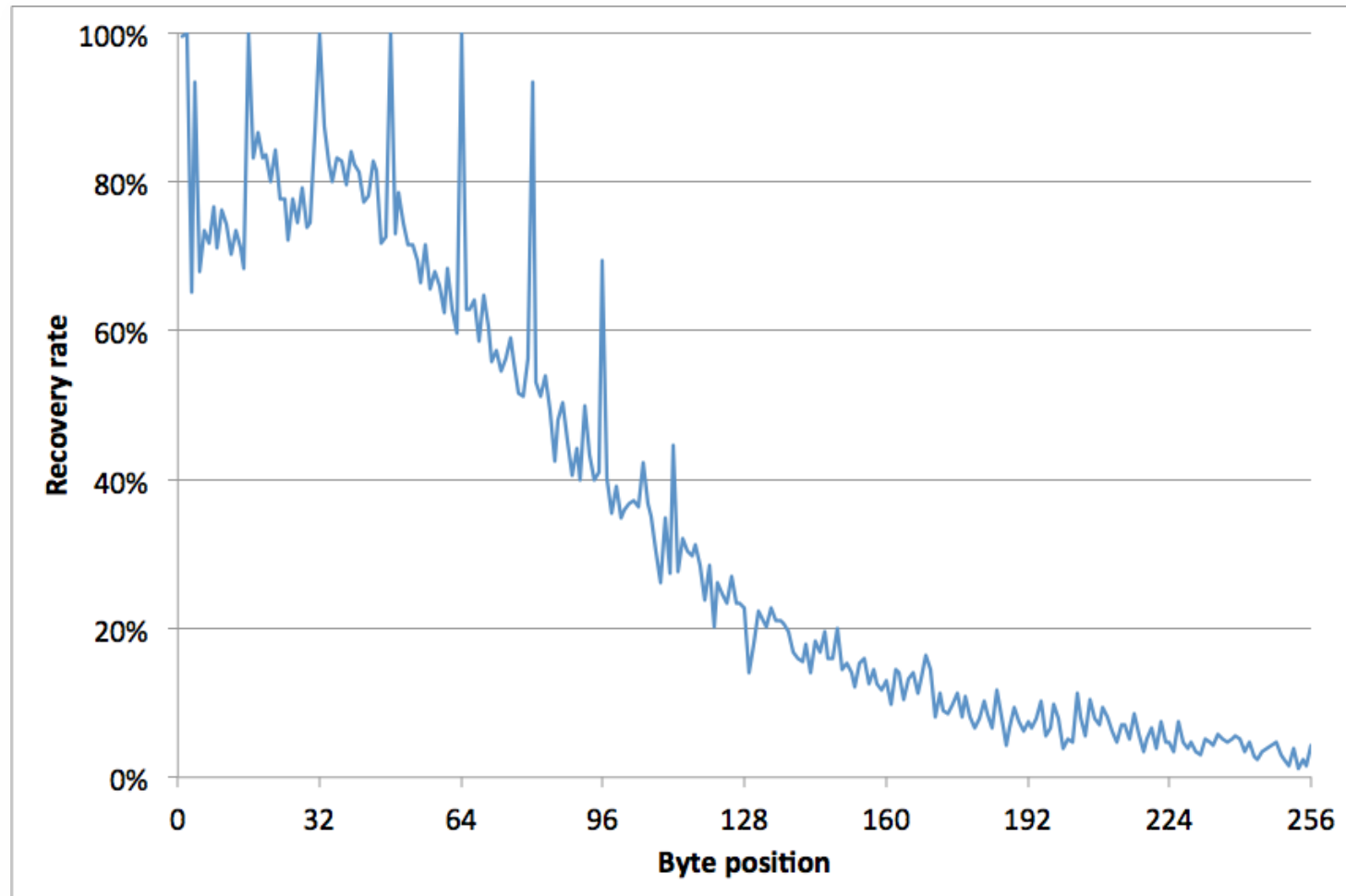# Success Probability 2²⁷ Sessions
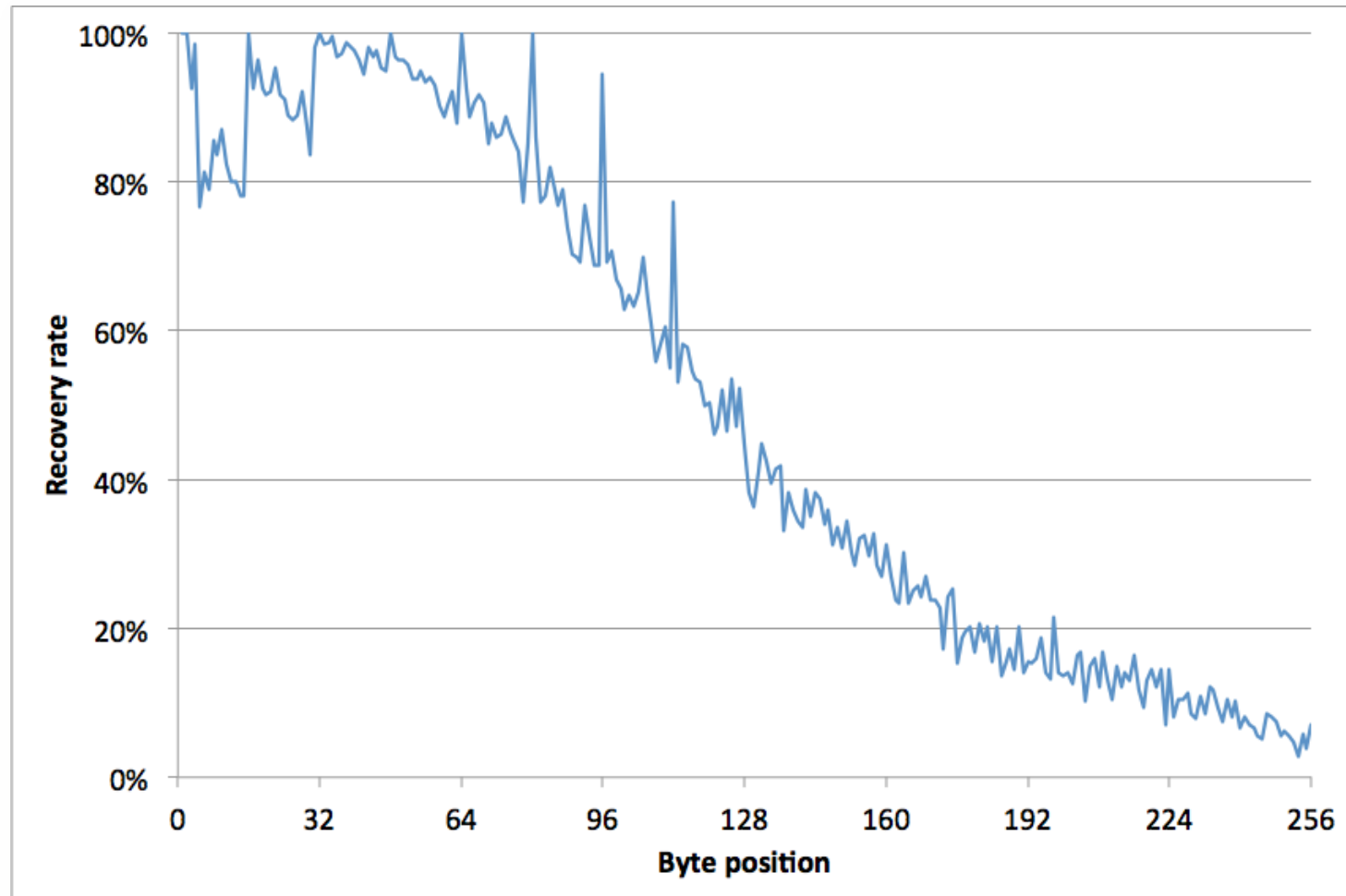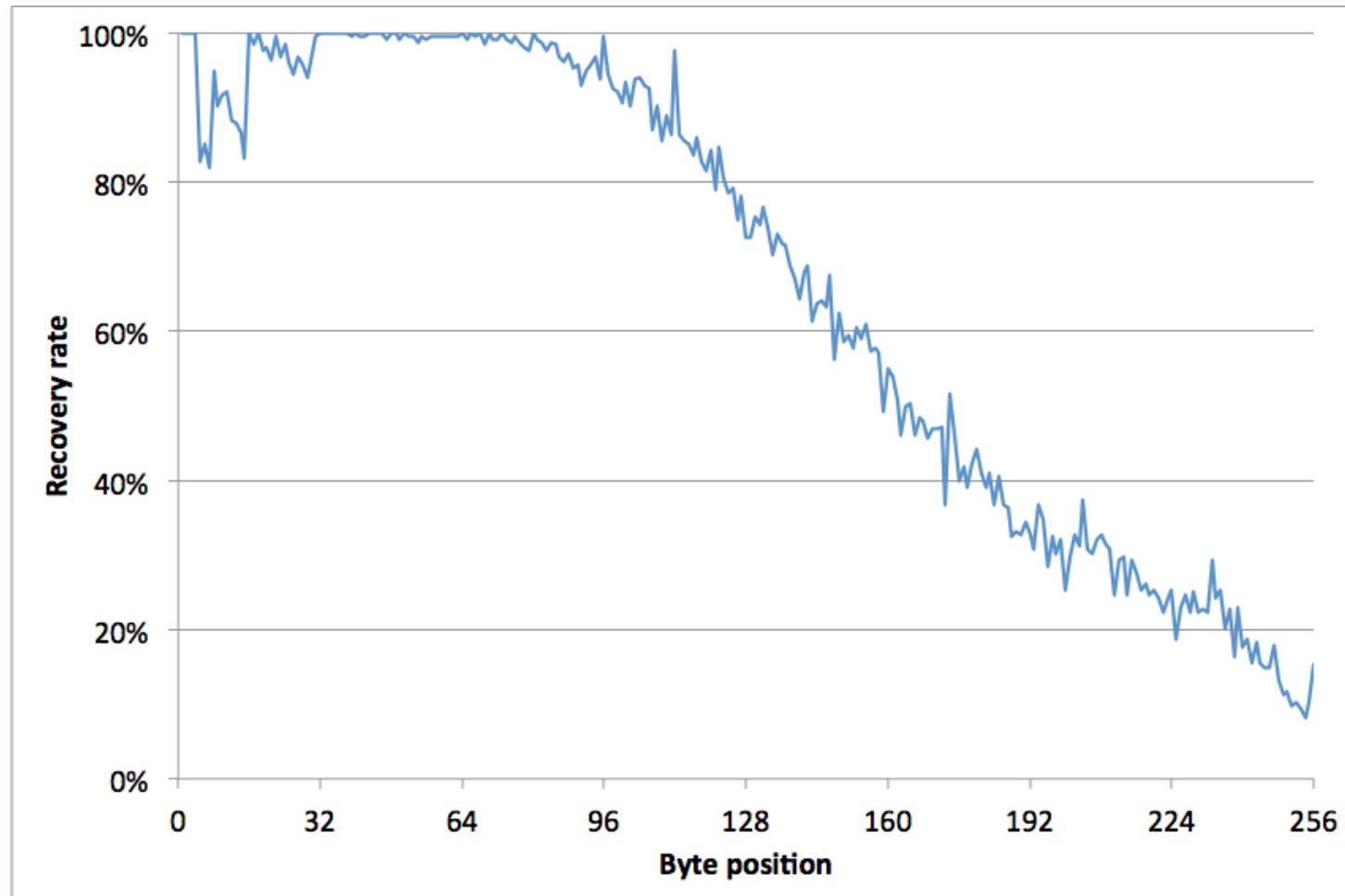
Success Probability 2²⁹ Sessions
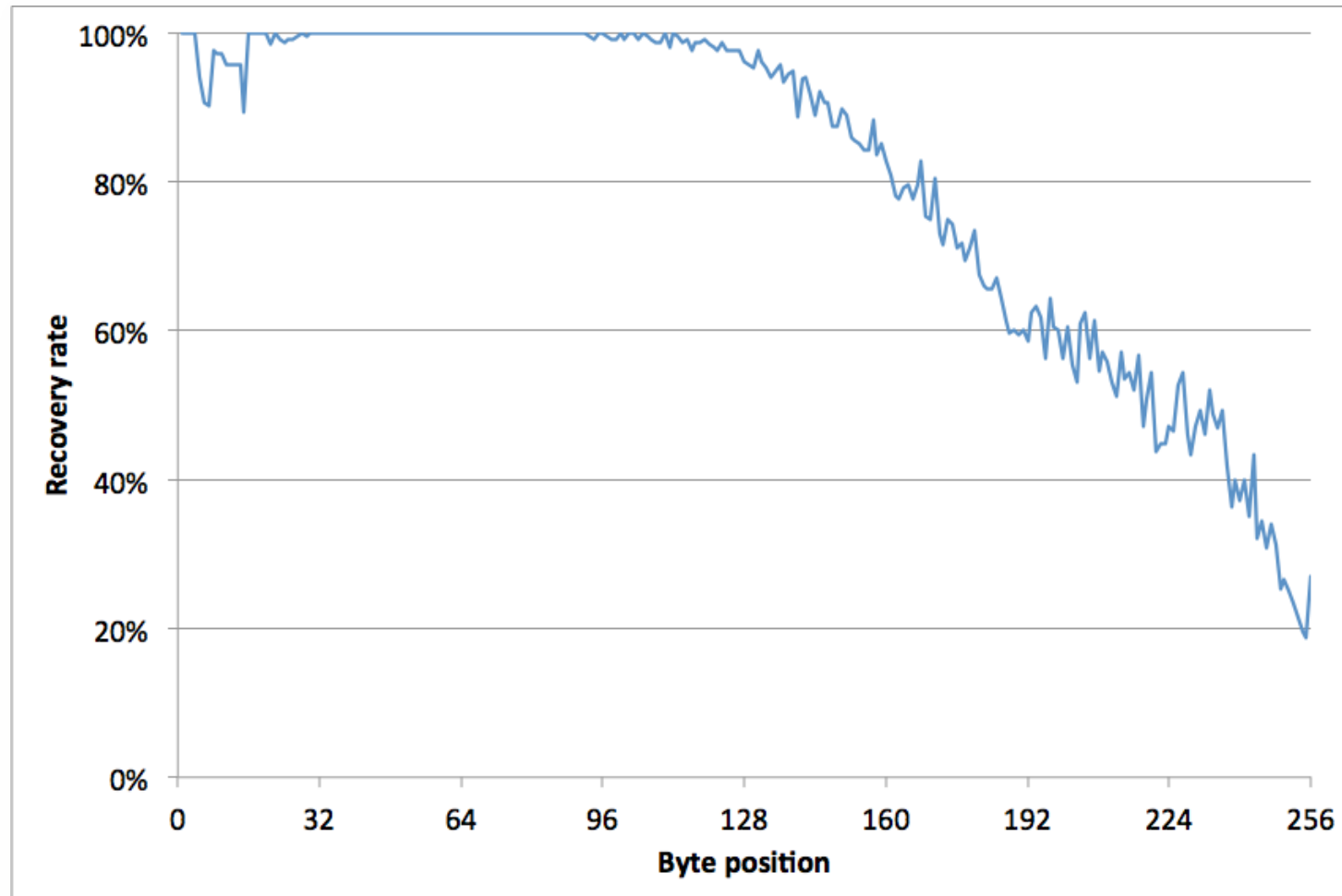
Success Probability 2³⁰ Sessions

Success Probability 2³¹ Sessions
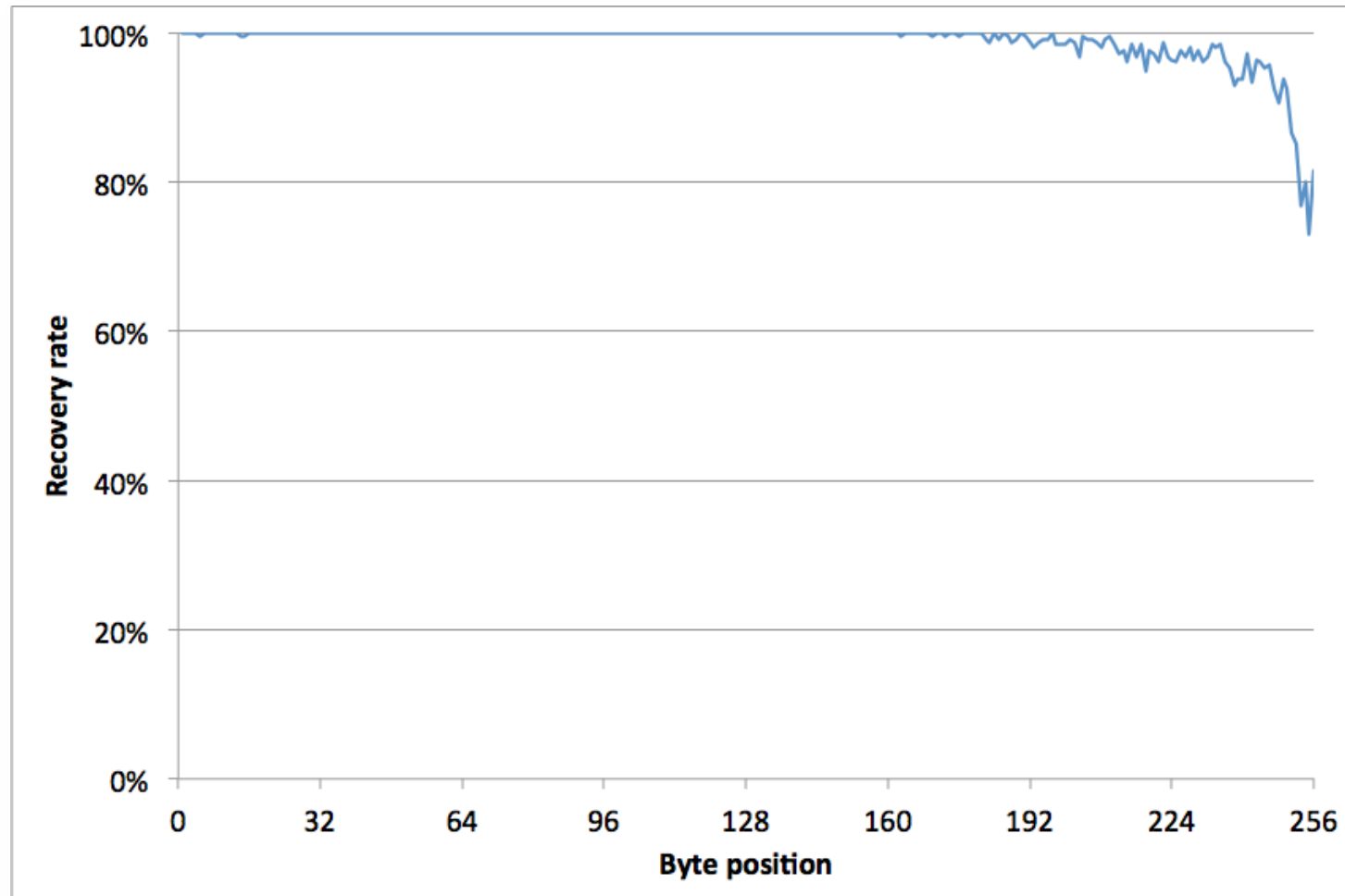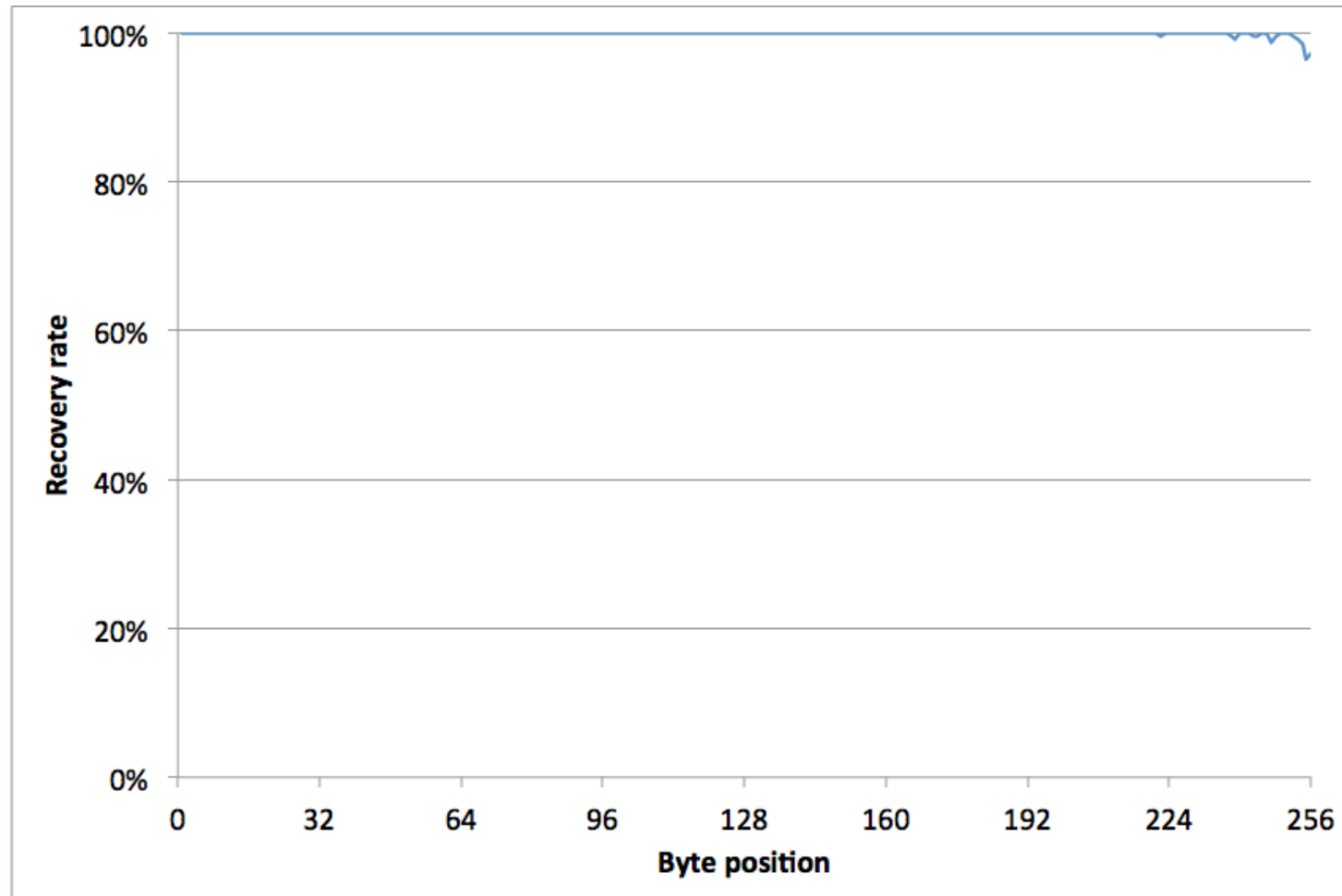
Limitations of Attack

Requires $2^{28} \sim 2^{32}$ TLS 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 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, we construct 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
$(0,0)$	$i = 1$	$2^{-16}(1 + 2^{-7})$
$(0,0)$	$i \neq 1, 255$	$2^{-16}(1 + 2^{-8})$
$(0,1)$	$i \neq 0, 1$	$2^{-16}(1 + 2^{-8})$
$(i+1, 255)$	$i \neq 254$	$2^{-16}(1 + 2^{-8})$
$(255, i+1)$	$i \neq 1, 254$	$2^{-16}(1 + 2^{-8})$
$(255, i+2)$	$i \neq 0, 253, 254, 255$	$2^{-16}(1 + 2^{-8})$
$(255, 0)$	$i = 254$	$2^{-16}(1 + 2^{-8})$
$(255, 1)$	$i = 255$	$2^{-16}(1 + 2^{-8})$
$(255, 2)$	$i = 0, 1$	$2^{-16}(1 + 2^{-8})$
$(129, 129)$	$i = 2$	$2^{-16}(1 + 2^{-8})$
$(255, 255)$	$i \neq 254$	$2^{-16}(1 - 2^{-8})$
$(0, i+1)$	$i \neq 0, 255$	$2^{-16}(1 - 2^{-8})$

A Second Attack

Align plaintext with repeating Fluhrer-McGrew biases by padding.

RC4 Keystream

Plaintext copies

TLS Ciphertexts

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

$P_3 P_4$ ...

$P_2 P_3$

$P_1 P_2$

$\Rightarrow$
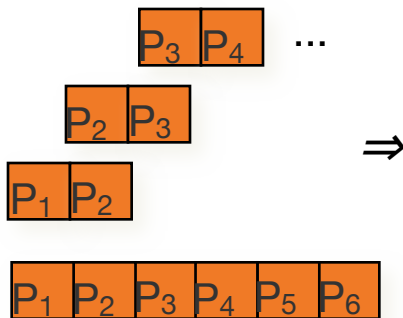
Approximate likelihood for
$P = P_1 P_2 P_3 P_4 P_5 P_6$

$P_1 P_2 P_3 P_4 P_5 P_6$

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

Success Probability
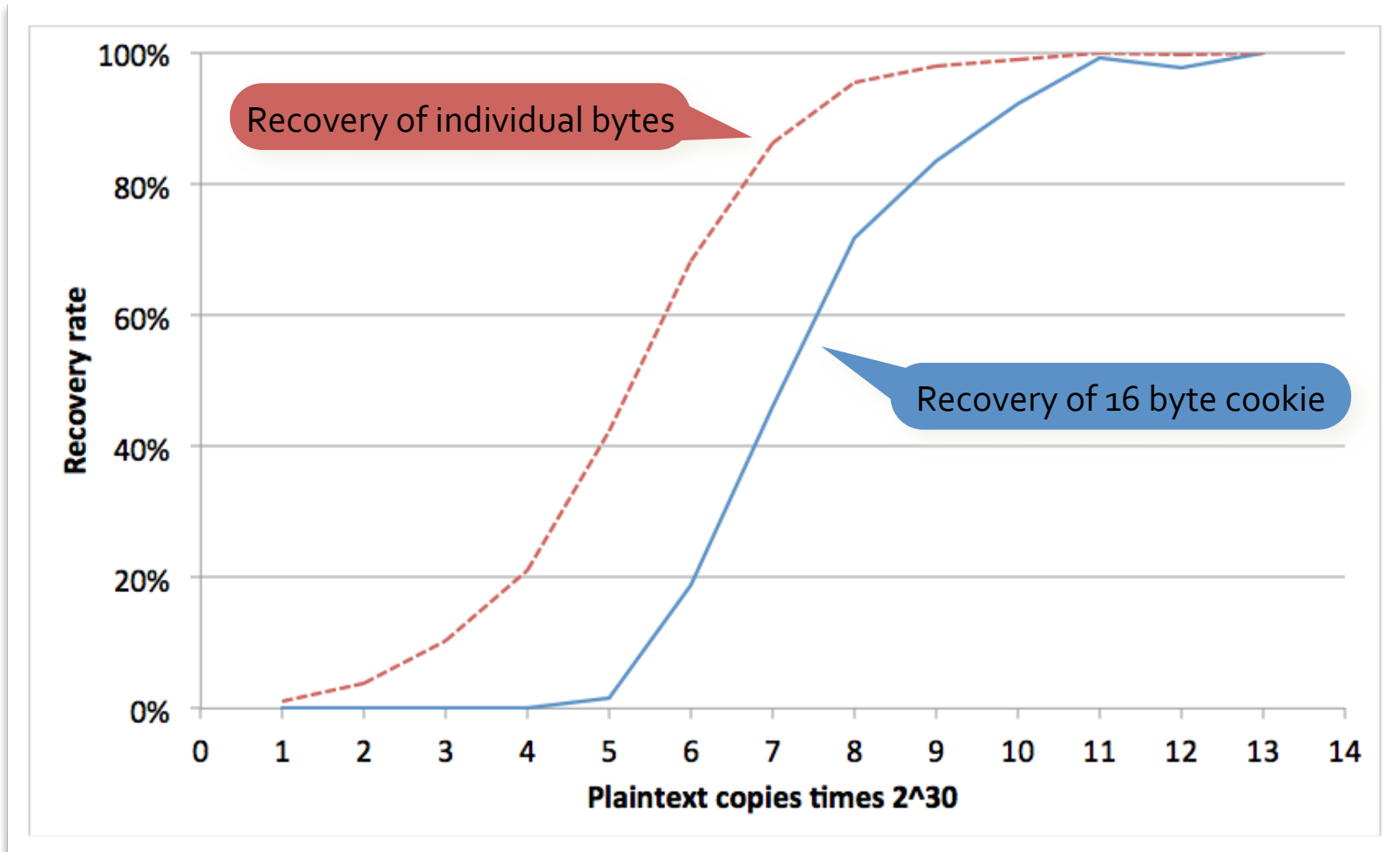
Countermeasures

Possible countermeasures against the attacks

Discard initial keystream bytes.

Fragment initial records at the application layer.

Add random length padding to records.

Limit lifetime of cookies or number of times cookies can be sent.

Stop using RC4 in TLS.

Vendor Responses to RC4 Attacks

Opera: 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.

Full details at www.isg.rhul.ac.uk/tls

Summary of RC4 in TLS

Plaintext recovery attacks against RC4 in TLS are feasible although not truly practical.

2^{28} ~ 2^{32} sessions for reliable recovery of initial bytes.

2^{33} ~ 2^{34} encryptions for reliable recovery of 16 bytes anywhere in plaintext.

The attacks illustrate that RC4 in TLS provides a security level far below the strength suggested by the key size of 128 bits.

Furthermore, attacks only becomes better with time…

Current Status and Future Developments in TLS

Current Status – CBC-mode

Most CBC-mode implementations have been patched against BEAST and Lucky 13 , but reputation damaged by long series of attacks.

Large ecosystem and trickiness of proper patch means that not every implementation is patched to the same degree.

Relative performance also an issue (AES-CBC + HMAC quite slow).
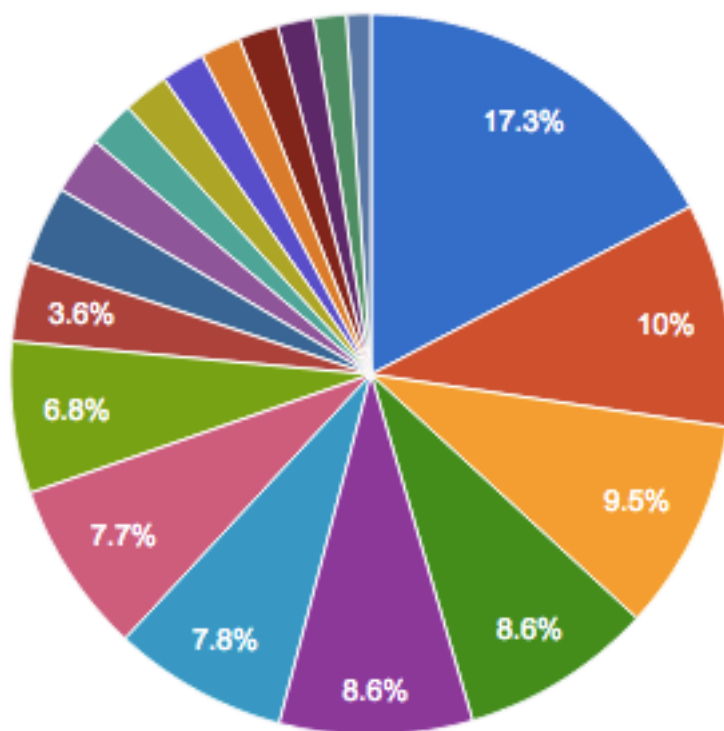
Snapshot from ICSI Certificate Notary Project:

Legend (with arrows marking RC4 entries):
- TLS_RSA_WITH_RC4_128_SHA ←
- TLS_ECDHE_RSA_WITH_AES_128_CBC_S...
- TLS_RSA_WITH_RC4_128_MD5 ←
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_GC...
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_GCM_S...
- TLS_ECDHE_RSA_WITH_RC4_128_SHA ←
- other
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC...
- TLS_RSA_WITH_NULL_SHA
- TLS_ECDHE_ECDSA_WITH_RC4_128_SHA ←
- TLS_ECDHE_RSA_WITH_AES_256_CBC_S...
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_S...
- TLS_ECDHE_RSA_WITH_AES_256_CBC_S...
- TLS_ECDHE_ECDSA_WITH_CHACHA20_P...
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA

>35%

Pie chart percentages: 17.3%, 10%, 9.5%, 8.6%, 8.6%, 7.8%, 7.7%, 6.8%, 3.6%

206

Current Status – RC4

RC4 should be dead, but is not.

Usage has dropped from 50% to 35% in about 1 year.

More work seems to needed to kill it off completely.

AES-GCM and AES-CCM are only available for TLS 1.2.

But TLS 1.2 is now supported in all major browsers.

And by 42.6% of top 200k websites.

So why is only 15.3% of traffic using AES-GCM?

Long tail of old servers and old clients.

Peculiarities of TLS ciphersuite negotiation and version fallback.

Snapshot from ICSI Certificate Notary Project:

Legend:
- TLS_RSA_WITH_RC4_128_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_S...
- TLS_RSA_WITH_RC4_128_MD5
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_GC...
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_GCM_S...
- TLS_ECDHE_RSA_WITH_RC4_128_SHA
- other
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC...
- TLS_RSA_WITH_NULL_SHA   ← 2.6%!
- TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_S...
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_S...
- TLS_ECDHE_RSA_WITH_AES_256_CBC_S...
- TLS_ECDHE_ECDSA_WITH_CHACHA20_P...
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA

Pie chart values: 17.3%, 10%, 9.5%, 8.6%, 8.6%, 7.8%, 7.7%, 6.8%, 3.6%

208

Current Status – AES-GCM

AES-based ciphersuites are generally slow without AES-NI instruction.

AES-GCM is tricky to implement securely.

    Main issue is avoiding leakage of hash key via side-channel attack.

    Also need side-channel resistant implementation of AES.

AES-GCM is relatively fast

    Especially with AES-NI and PCLMULQDQ instructions (Intel and AMD).

    2.53 to 1.03 cycles per byte, depending on processor.

        http://2013.diac.cr.yp.to/slides/gueron.pdf

    Roughly twice as fast as AES-CBC + HMAC-SHA-*

    But OCB would be even faster!

Current Developments

Fresh algorithms are under active consideration in IETF TLS WG.

Important for environments where AES is not available in hardware.

Momentum behind ChaCha20 stream cipher plus Poly1305 MAC.

See https://tools.ietf.org/html/draft-irtf-cfrg-chacha20-poly1305-01

Combination is a good AEAD scheme assuming ChaCha20 is ideal [P14].

But additional review needed of ChaCha20.

Reform of MEE to EtM to make CBC-mode easier to implement securely.

RFC 7366 (Gutmann) recently published .

Deployment via TLS extension, unclear how widely adopted it will become.

Current Developments

TLS 1.3 is now under active development in TLS WG.

Reducing latency in Handshake.

Simplification of key exchange and authentication methods in Handshake.

Server name/identity hiding/handshake encryption for improved privacy (?)

Reform of symmetric algorithms.

Active review of drafts needed by users and cryptographers.

Closing Remarks

Once a bad cryptographic choice is out there in implementations, it's very hard to undo.

Old versions of TLS hang around for a long time.

There is no TLS product recall programme!

Slow uptake of TLS 1.1, 1.2.

TLS has come under sustained pressure from attacks.

BEAST, Lucky 13 and RC4 attacks are providing incentives to move to TLS 1.2.

Attacks are "semi-practical" but we ignore such attacks at our peril.

Good vendor response to Lucky 13, less so to RC4 attack.

Closing Remarks

Good algorithm design is hard.

But so is good protocol design.

Attacks are usually obvious in retrospect.

> But so is a lot of crypto theory!

Finding attacks is high-risk, high-reward.

Keep in mind the value of attacks on paper versus attacks in practice.

> Implemented attacks are needed to convince practitioners.
>
> But an on-paper attack is often the harbinger of a practical attack.

Closing Remarks

There are many possible research directions in this area.

- We didn't even talk about TLS Handshake security!

- Or Heartbleed, or the CCS attack on OpenSSL, or Frankencerts…

TLS has provided a particularly rich seam of attacks.

- Maybe more attacks still to find?

If you enjoyed these lecture, then you might also enjoy…

Real World Cryptography 2015
London, UK, 7-9 January 2015
http://www.realworldcrypto.com/rwc2015
#realworldcrypto

Speakers to include:

Elena Andreeva (K.U. Leuven)

Dan Bogdanov (Cybernetica)

Sasha Boldyreva (Georgia Tech)

Claudia Diaz (K.U. Leuven)

Roger Dingledine (Tor project)

Ian Goldberg (U. Waterloo)

Arvind Mani (LinkedIn)

Luther Martin (Voltage Security)

Elisabeth Oswald (U. Bristol)

Scott Renfro (Facebook)

Ahmad Sadeghi (TU Darmstadt)

Elaine Shi (UMD)

Brian Sniffen (Akamai)

Nick Sullivan (CloudFlare)

Selective TLS Attack Literature

[V02] Vaudenay, Eurocrypt 2002

[M02] Moeller, http://www.openssl.org/~bodo/tls-cbc.txt, 2002

[CHVV03] Canvel *et al.*, Crypto 2003

[RD09] Ray and Dispensa, TLS renegotiation attack, 2009.

[PRS11] Paterson *et al.*, Asiacrypt 2013

[DR11] Duong and Rizzo, "Here come the XOR Ninjas", manuscript 2011.

[AP12] N.J. AlFardan and K.G. Paterson, NDSS 2012.

[DR12] Duong and Rizzo, CRIME, various webpages + Ekoparty 2012.

[MVVP12] Mavrogiannopoulos *et al.*,CCS 2012.

[SP12] Smyth and Pironti, USENIX WOOT 2013.

[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.

[BDFPS14] Bhargavan *et al.,* IEEE S&P, 2014.