Bar-Ilan Winter School
Lecture 6
Attacks and security notions for the TLS secure channel

Kenny Paterson @kennyog
Based on joint work with Martin Albrecht, Jean Paul Degabriele and Torben Hansen

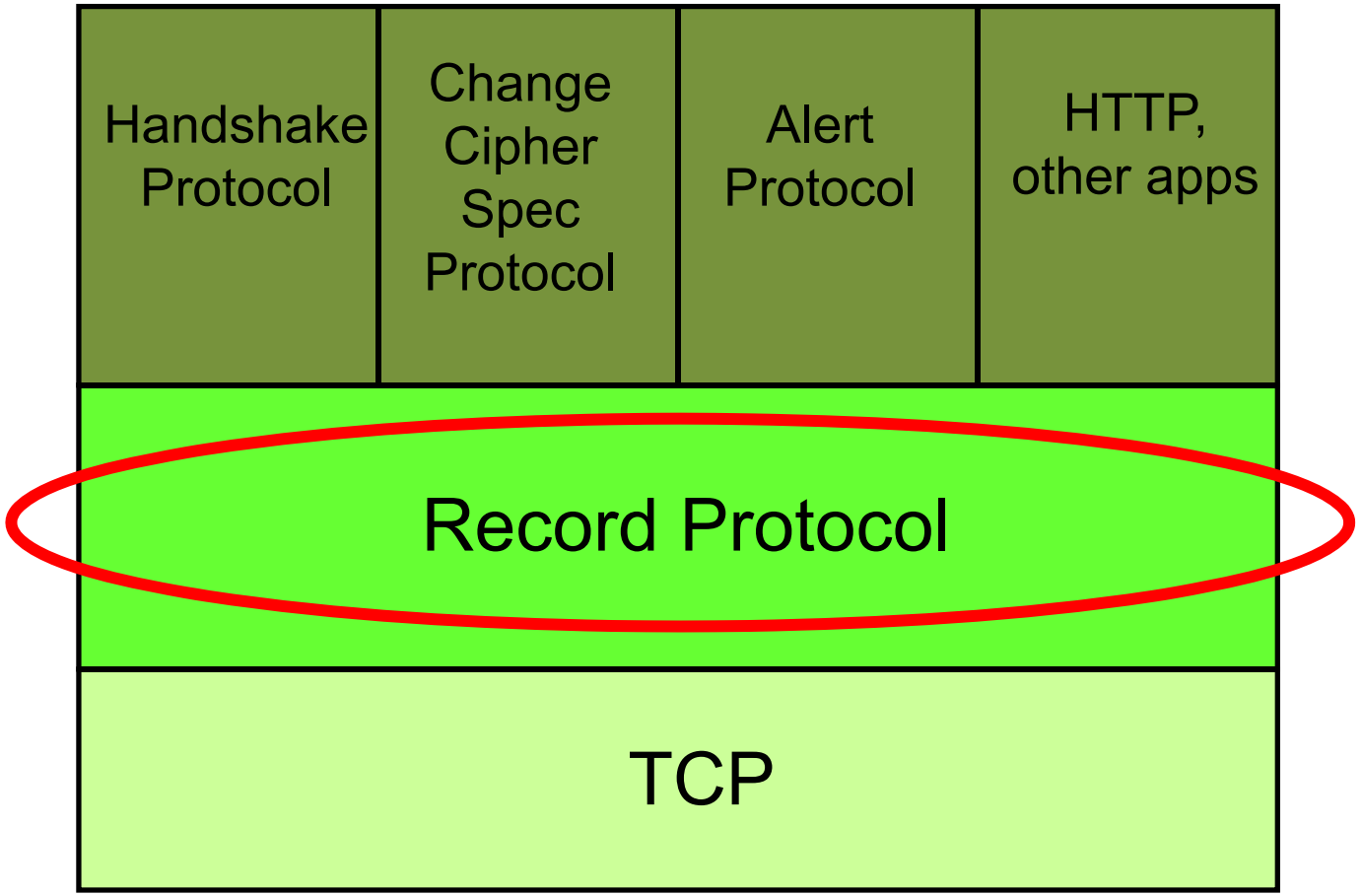
ROYAL HOLLOWAY UNIVERSITY OF LONDON

Overview

1. The TLS Record Protocol
2. An unfortunate sequence of attacks on the TLS Record Protocol
3. Security modelling for streaming secure channels
4. Concluding remarks


The TLS Record Protocol

TLS Protocol Architecture



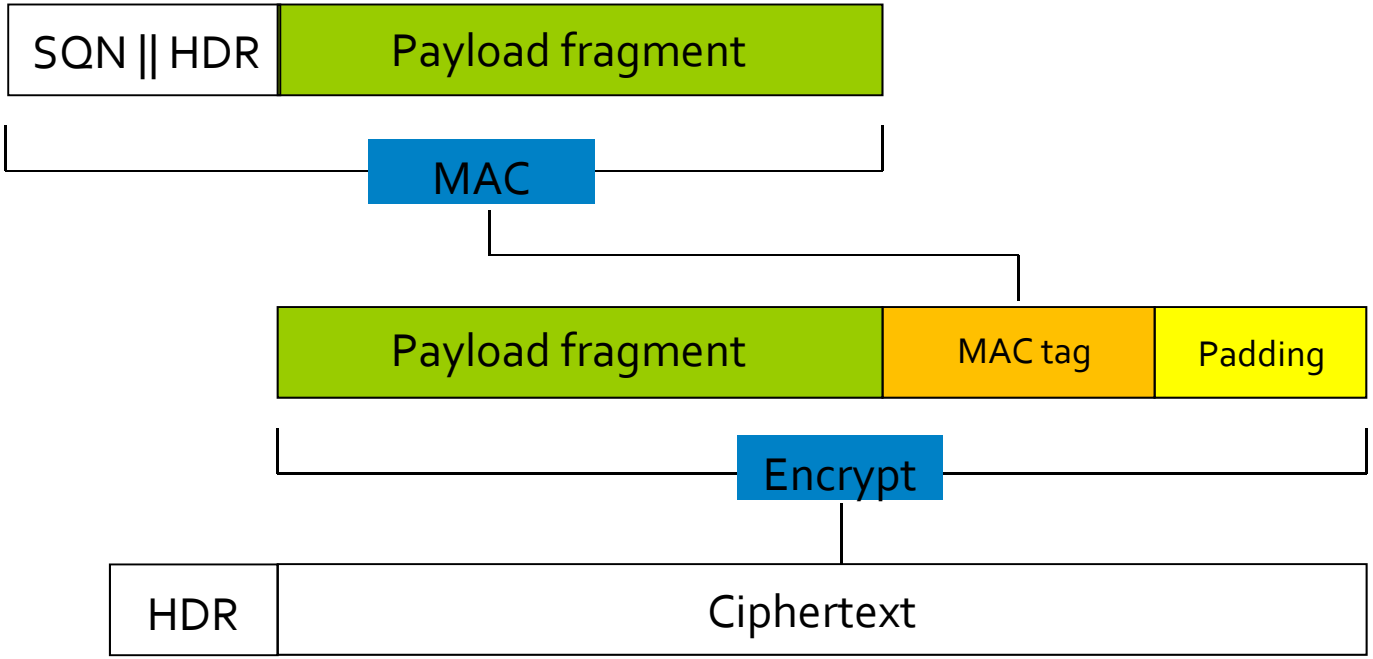
TLS Record Protocol

TLS Record Protocol provides:

- Data origin authentication, integrity using a MAC.
- Confidentiality using a symmetric encryption algorithm.
- Anti-replay using sequence numbers protected by the MAC.
- (Optional compression.)

- TLS presents a *stream-oriented API* to applications.
- So TLS may fragment into smaller records or coalesce into larger records any data supplied by the calling application.
- Hence if the calling application wants to deliver "atomic" messages, then it needs to add its own message delimiters.

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”

Operation of TLS Record Protocol

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

Operation of TLS Record Protocol

In-bound processing steps reverses these steps:

1. Receive message, of length specified in HDR.
2. Decrypt.
3. Remove padding.
4. Check MAC.
5. (Decompress payload.)
6. Pass payload to upper layer

(NB: no *de*fragmentation; TLS just 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: error message sent and connection is terminated.

AEAD and TLS Record Protocol

Dedicated *Authenticated Encryption with Associated Data* (AEAD) algorithms were added in TLS 1.2, along with the MEE construction.

- AEAD: single algorithm providing both confidentiality and integrity/data origin (authentication)
- Need not conform to MEE template.
- General AEAD interface specified in RFC 5116.
- Nonce construction not fully specified; natural choice is to use TLS SQN or random values.
- AES-GCM specified in RFC 5288.
- AES-CCM specified in RFC 6655.
- ChaCha20-Poly1305 specified in RFC 7539 and RFC 7905.

TLS Record Protocol Design Decisions

- Stream-oriented.
    - Application layer is responsible for demarcating message boundaries if desired.
    - Fragmentation done by Record Protocol when sending, but defragmentation not done when receiving.
- Most errors are fatal.
    - TLS runs over TCP, which is assumed to provide reliable transport.
    - Hence any error arising during in-bound processing should be treated as an attack.
    - Session terminated with error message, keys thrown away.
    - So DoS attacks are trivial to mount.
    - No retransmission of lost messages by TLS itself.

TLS Record Protocol Design Decisions

- Implicit sequence numbers.
  - 8-byte SQN included in MAC calculation, but not sent on the wire as part of Record Protocol messages.
  - Sender and receiver are assumed to maintain local copies of SQN, incrementing for each message sent/received.
  - Any replay, re-ordering or dropping of messages should be detected through MAC verification failure at receiver.
  - MAC verification failure is a fatal error.
- No attempt to hide message/fragment lengths.
  - Leads to fingerprinting attacks (e.g. Pironti-Strub-Bhargavan, INRIA research report 8067, 2012).
  - Made worse by switch to AES-GCM.
  - Can be partially addressed by use of variable length padding in CBC mode.

TLS Record Protocol Design Decisions

- Use of compression was known in theory to be dangerous.
  - Kelsey, FSE'04.
- Choice of MEE is *not* fully-supported by theory.
  - MtE known to be *not* generically secure (Bellare-Namprempre, Asiacrypt'01).
  - Krawczyk (Crypto'01) provides support for MtE when CBC-mode is used or when stream cipher is used.
  - But the analysis assumes:
    - Random per message IV, no padding, block-size = MAC tag size for CBC mode.
    - Stream cipher has outputs that are indistinguishable from random.
  - More recent analysis of Namprempre-Rogaway-Shrimpton- (EC'14) says MtE provides AE if "E" is "tidy".
    - TLS's choice of "E" is not tidy!

TLS Record Protocol Design Decisions

- The fact is that suitable theory did not exist at the time TLS was designed.
    - Essentially, we need stateful AEAD security.
- Consensus then was that "MtE" is better than "EtM".
    - "Authenticate what you mean to say, not an encrypted version of it." – the Horton principle.
    - "Maybe our MAC algorithms are weak, so we should protect the MAC value by encrypting it."
- Today, we have better theory, but it's been hard to get it deployed.
    - Because it had to displace what's already been massively deployed.
    - Change has been driven by attacks!

Adoption of AEAD in TLS

89.2% of the Alexa top 135k websites now support TLS 1.2 and hence AEAD.

(Up from 82.6% one year ago, 69.5% two years ago, 42.6% three years ago, 17% four years ago and 5% five years ago.)

(source: ssl pulse, Dec. 2017)

TLS 1.2 support in browsers:


Chrome: since release 30.


Firefox: since release 28.

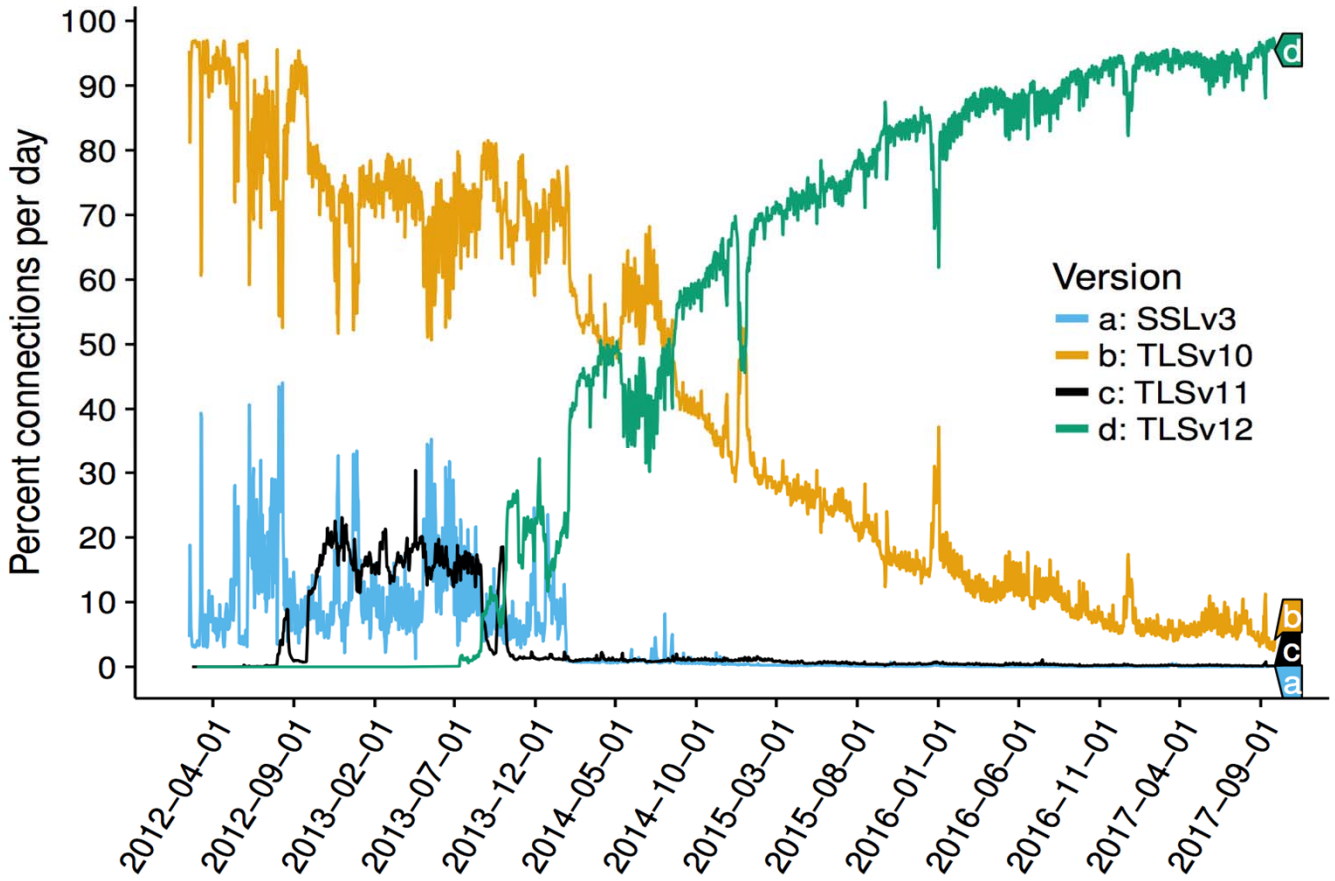

IE: since IE11.


Safari: since iOS5 and OS X 10.9.

(source: wikipedia, Nov. 2013)

Stronger, modern AEAD designs are increasingly being used.

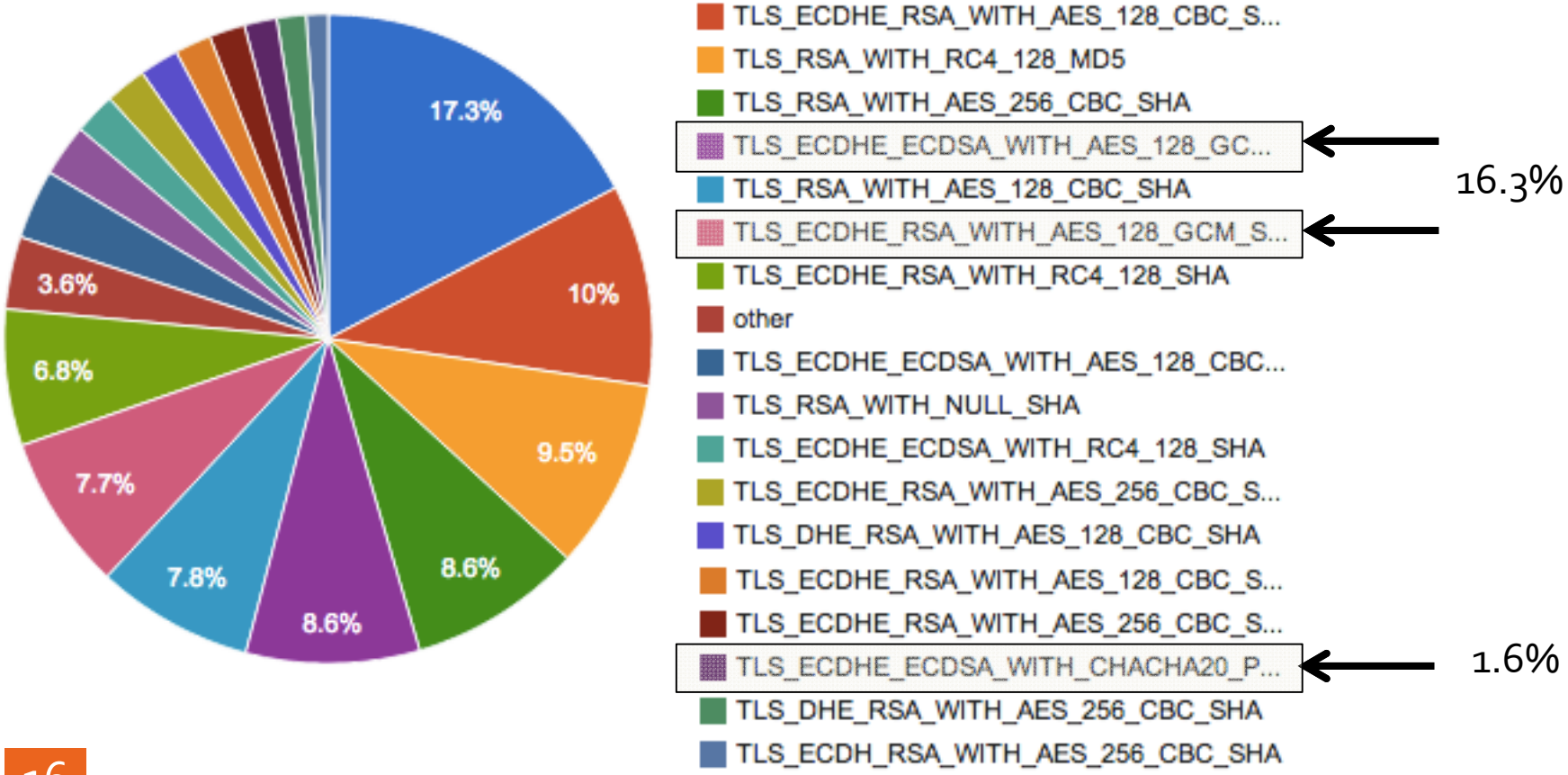
SSL/TLS Versions in Use on the Internet



Source: Amann et al. "Mission Accomplished? HTTPS Security after DigiNotar", IMC 2017.

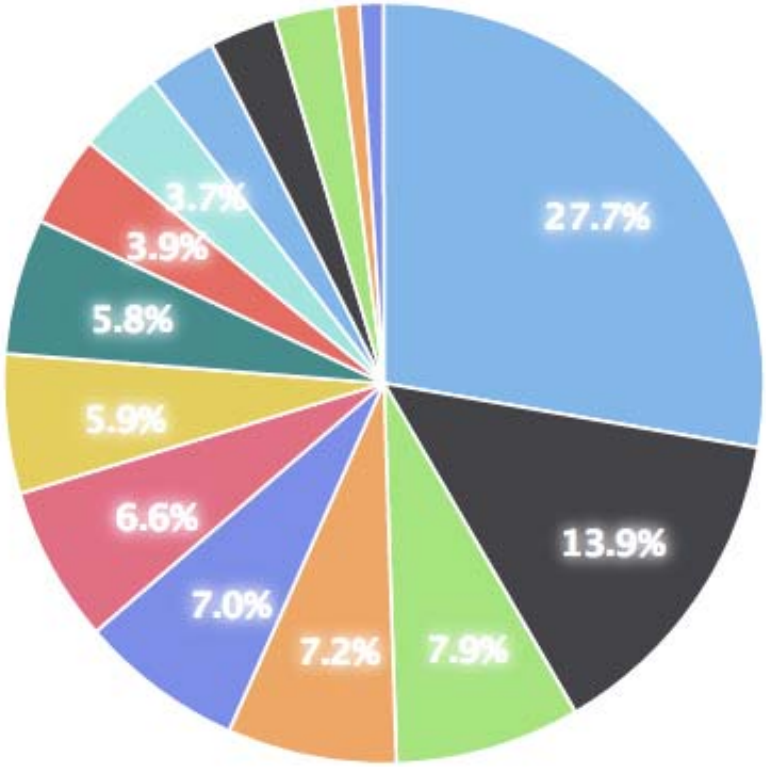
AEAD Usage in TLS: September 2014

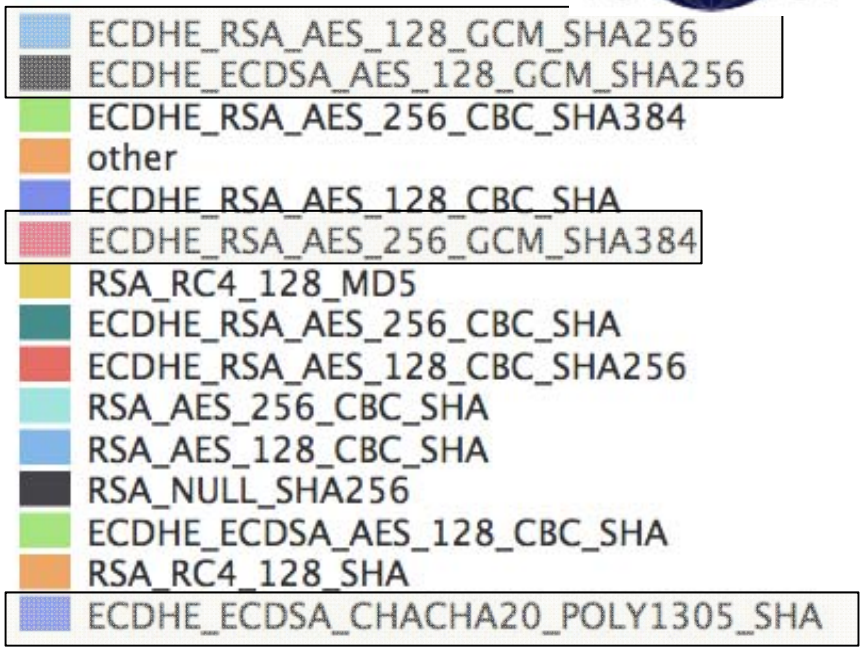
Snapshot from ICSI Certificate Notary Project



AEAD Usage in TLS: December 2015

Snapshot from ICSI Certificate Notary Project:

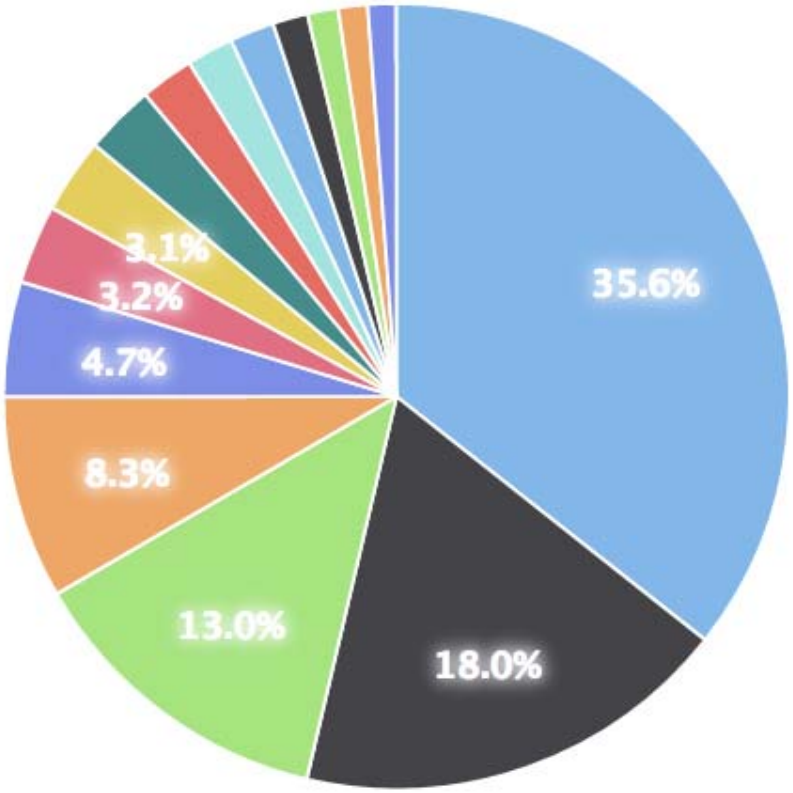




Total of AES-GCM just below 50%

AEAD Usage in TLS: December 2016

Snapshot from ICSI Certificate Notary Project:

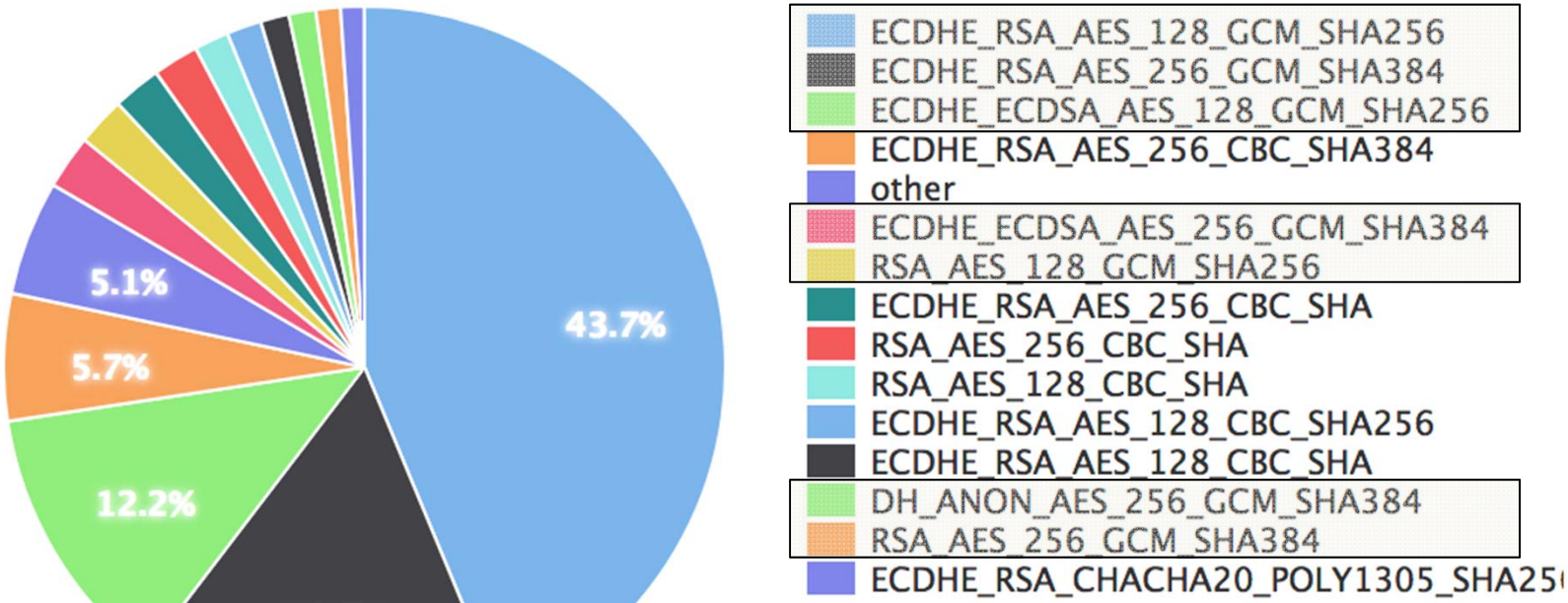


ECDHE_RSA_AES_128_GCM_SHA256
ECDHE_RSA_AES_256_GCM_SHA384
ECDHE_ECDSA_AES_128_GCM_SHA256
ECDHE_RSA_AES_256_CBC_SHA384
other
ECDHE_RSA_AES_128_CBC_SHA
ECDHE_RSA_AES_256_CBC_SHA
RSA_AES_256_CBC_SHA
RSA_AES_128_CBC_SHA
ECDHE_RSA_AES_128_CBC_SHA256
RSA_AES_128_GCM_SHA256
ECDHE_ECDSA_CHACHA20_POLY1305_SHA
ECDHE_ECDSA_AES_128_CBC_SHA
ECDHE_ECDSA_AES_256_GCM_SHA384
RSA_AES_256_GCM_SHA384

Total of AES-GCM above 66%

AEAD Usage in TLS: February 2018

Snapshot from ICSI Certificate Notary Project



Nearly 80% is now AEAD (AES-GCM), 14% is CBC.
1.1% ChaCha20-Poly1305 (other sources report 10%).


TLS Record Protocol Security Issues

Overview of TLS Record Protocol attacks

- **BEAST** (2011)– exploits TLS 1.0's use of predictable IVs.
- **CRIME (and BREACH, TIME)** (2012) – exploits TLS and applications support for compression.
- **Padding oracle attack** (2002, 2003) – exploits TLS 1.0's use of distinguishable error messages for padding and MAC failures.
- **Lucky 13** (2013) – padding oracle attacks are still possible, even after application of recommended countermeasures; MEE with CBC is hard to implement without side channels.
- **POODLE** (2014) – a special kind of padding oracle attack for SSL3.0, based on error messages rather than timing; SSL3.0-killer.
- **RC4 attacks** (2013-2015) – RC4 is not such a good stream cipher after all.
- **Sweet 32** – small block-size block ciphers in CBC mode start to leak plaintext after 2^{32} – 2^{35} blocks.


BEAST

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.

BEAST – Impact

The BEAST was a major headache for TLS vendors.

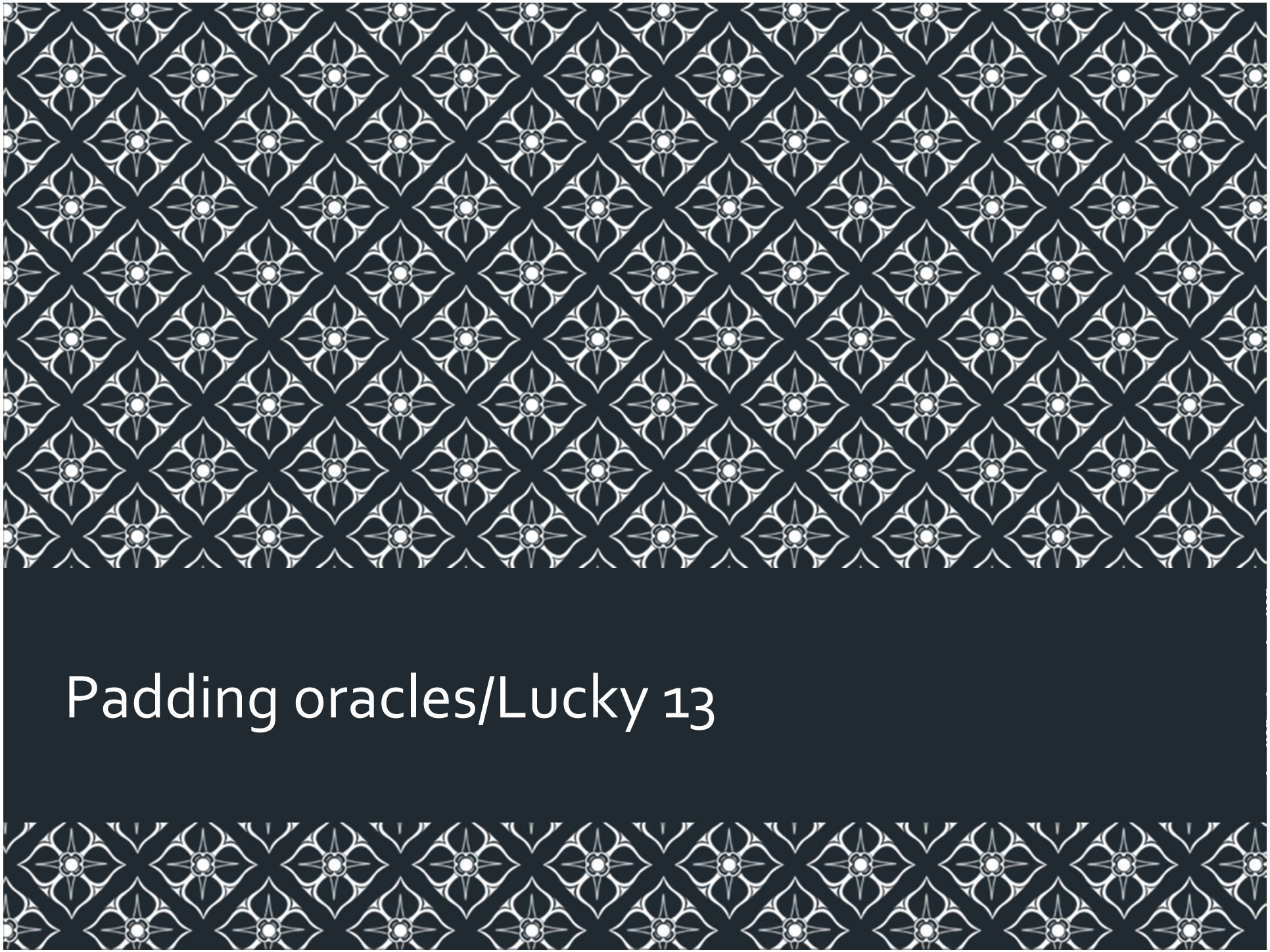
- Perceived to be a realistic attack.
- Most client implementations were "stuck" at TLS 1.0.

Best solution: switch to using TLS 1.1 or 1.2.

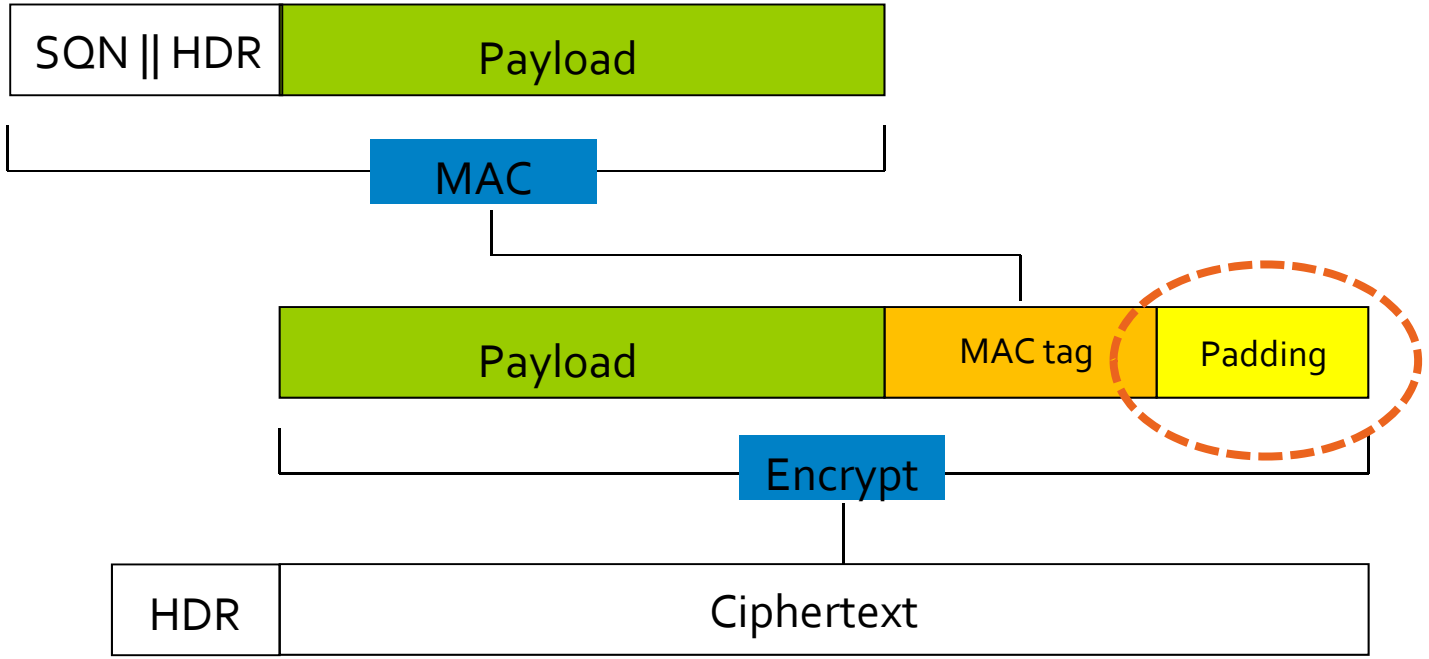
- Uses random IVs, so attack prevented.
- But needs server-side support too.

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.


Padding oracles/Lucky 13

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”

Padding Check in TLS

- 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:

00;

01, 01;

02, 02, 02;

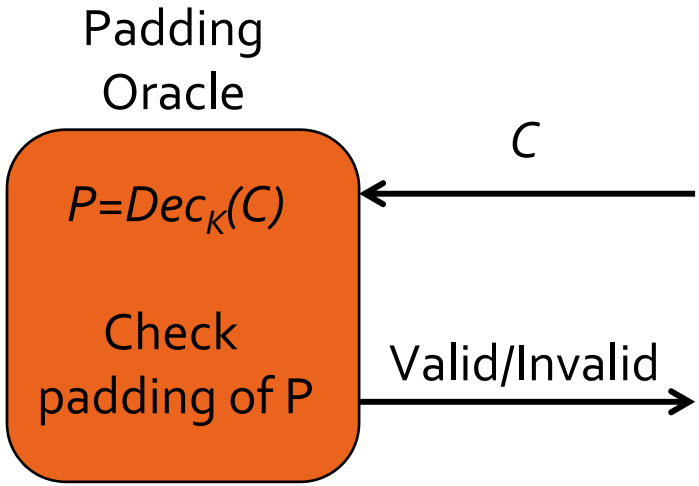
....

FF, FF,.....FF;

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

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

Reminder: Padding Oracles



For CBC mode and for certain padding schemes, a padding oracle can be used to build a decryption oracle!

28

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.

- A 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.
  - Modern viewpoint: use BEAST-style Javascript in the browser 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.

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 Oracle Countermeasures, Revisited

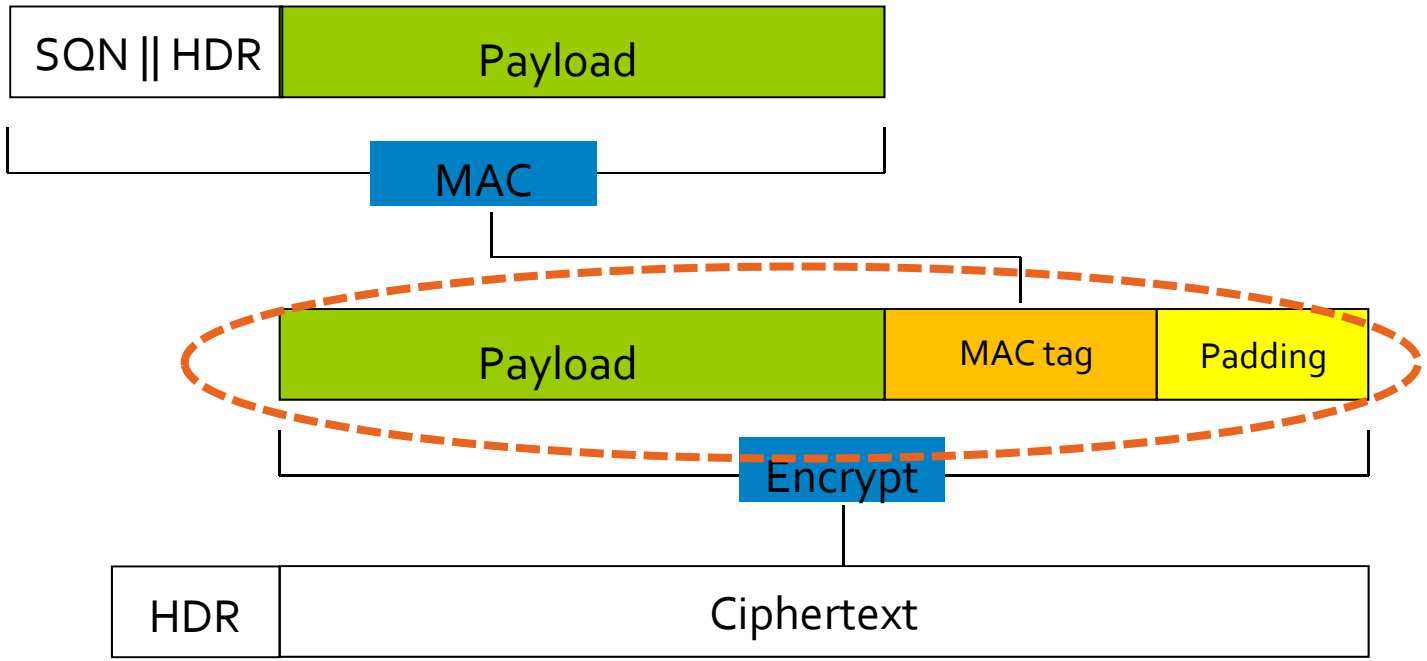
From the TLS 1.1 and 1.2 specifications:

…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.

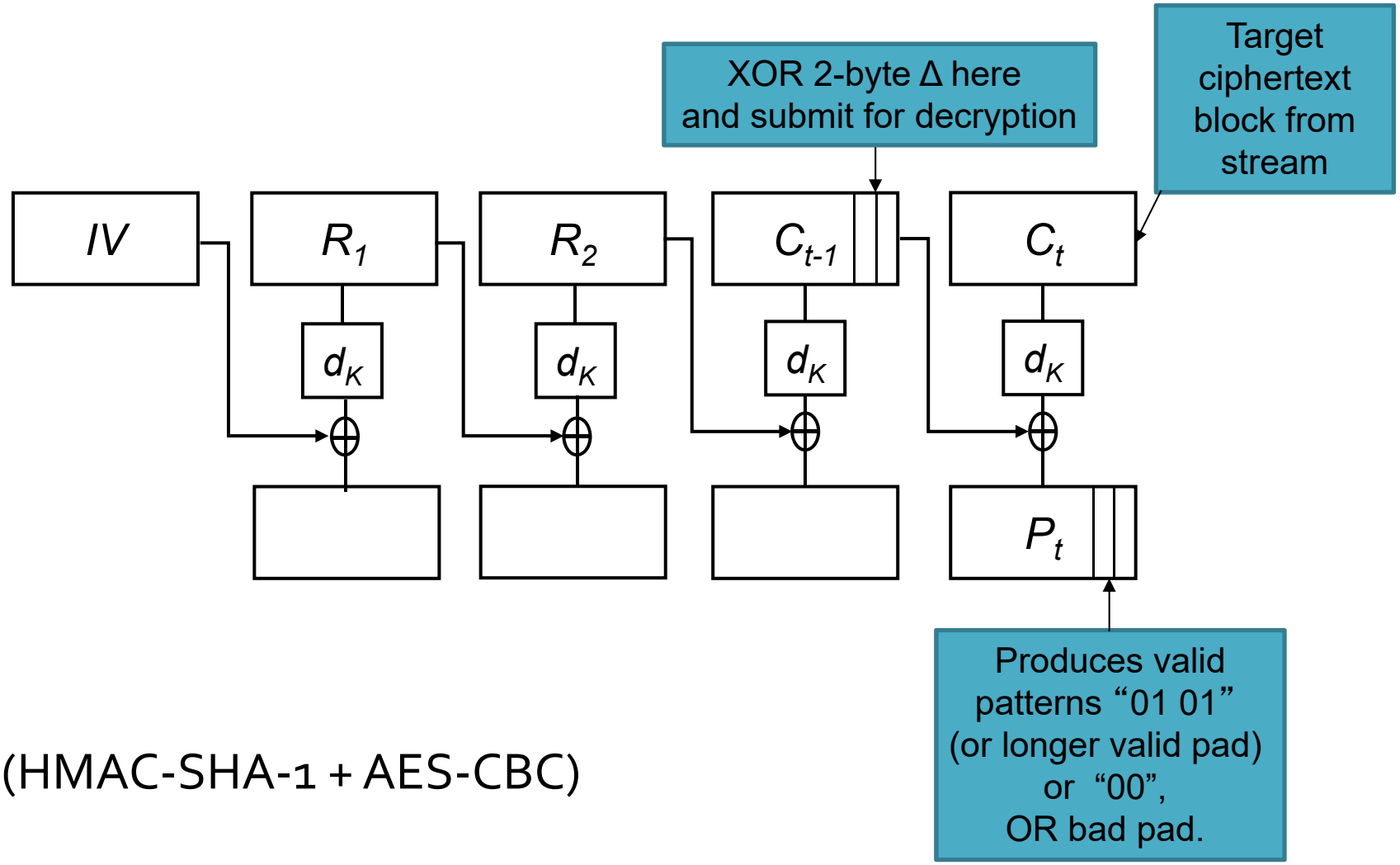
Ensuring Uniform Errors

… 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.

Lucky 13: Main Idea

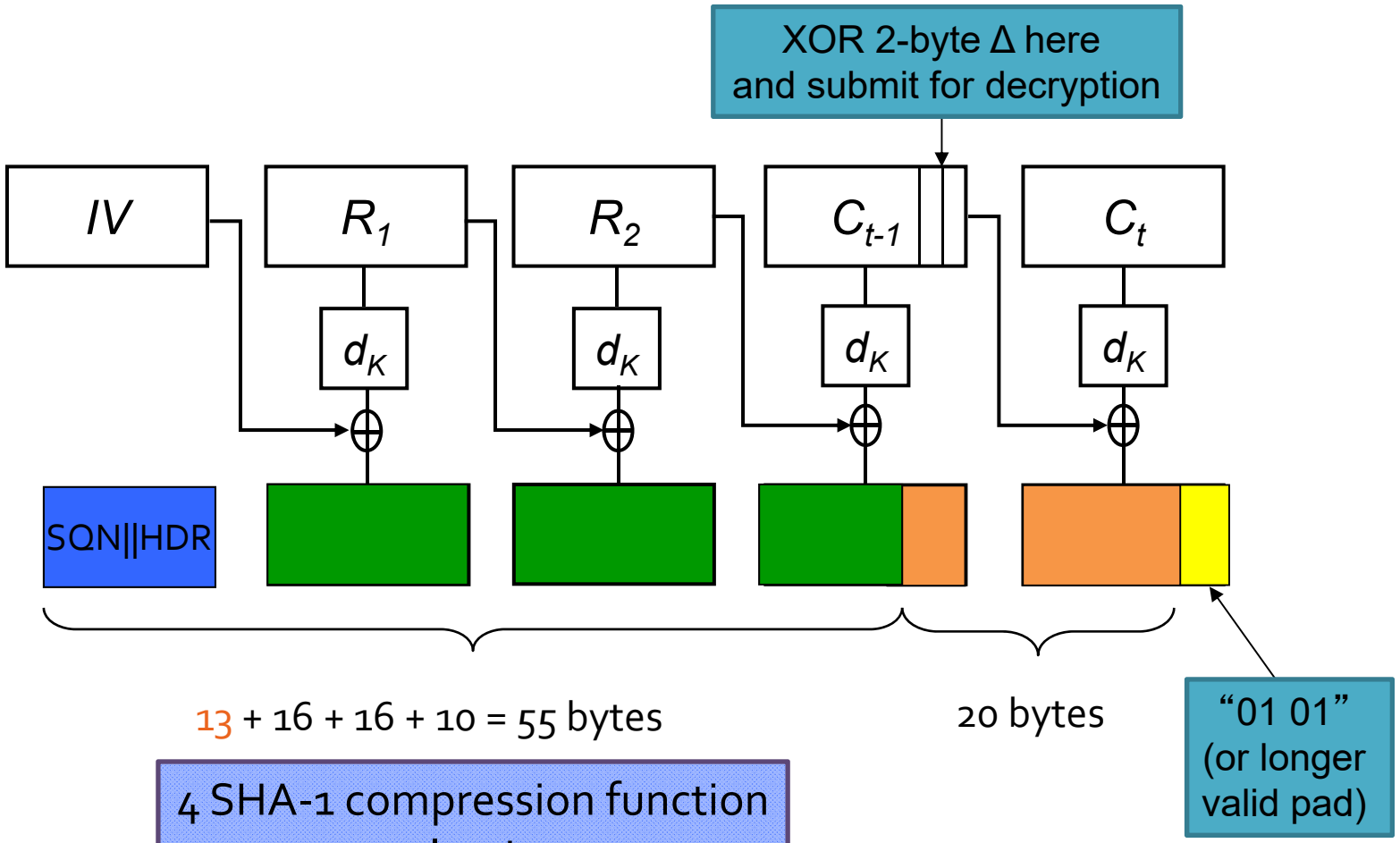
- TLS decryption removes padding and MAC tag to extract PAYLOAD.
- HMAC computed on SQN || HDR || PAYLOAD.
- HMAC computation involves adding ≥9 bytes of padding and iteration of hash compression function, e.g. MD5, SHA-1, SHA-256.
- Running time of HMAC depends on *L*, the exact byte length of SQN || 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;
    - ….

Lucky 13 – Plaintext Recovery

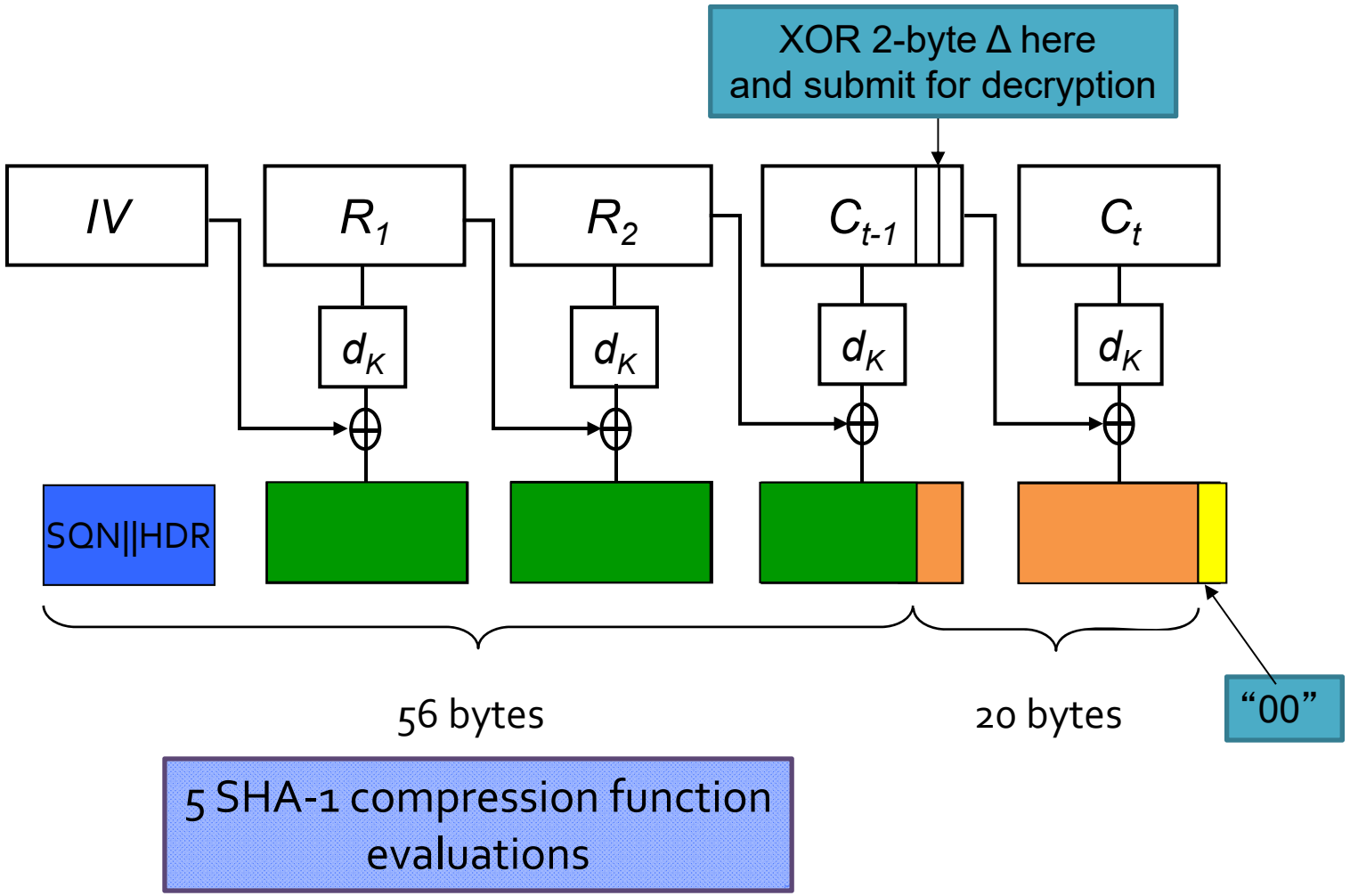


(HMAC-SHA-1 + AES-CBC)

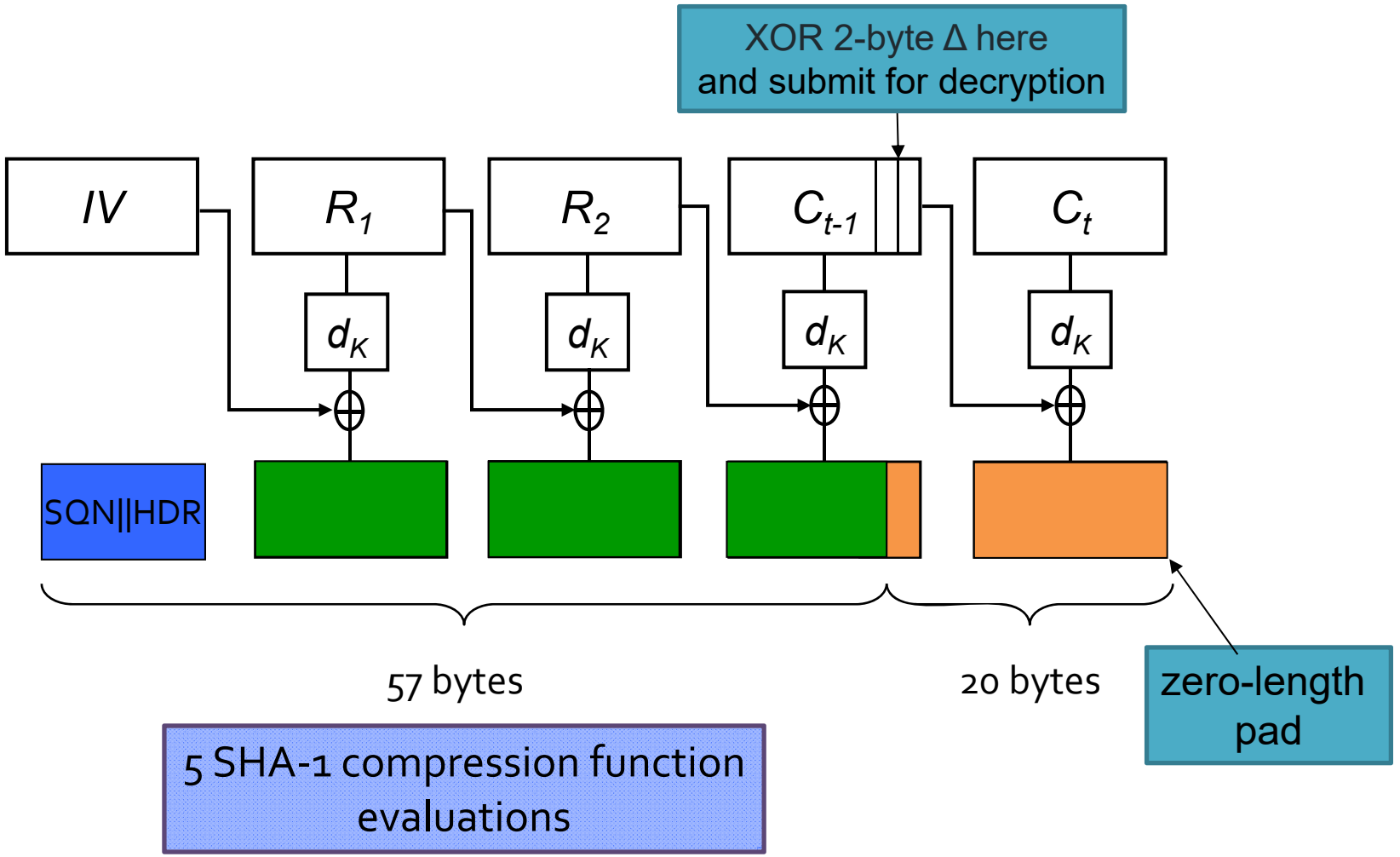
Case 1: "01 01" (or longer valid pad)



Case 2: "00"


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
56 bytes
20 bytes
"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 500 clock cycles, well below 1µs on a typical processor.
- But 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: $P_t \oplus (\ldots\ldots\Delta_1\Delta_0) = (\ldots.0101)$.
- Attack then extends to all bytes as in a standard padding oracle attack.

Constant Time Decryption for MEE

- Proper constant-time, constant-memory access implementation of MEE decryption 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.
- Fix required around 500 lines of new code.

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)

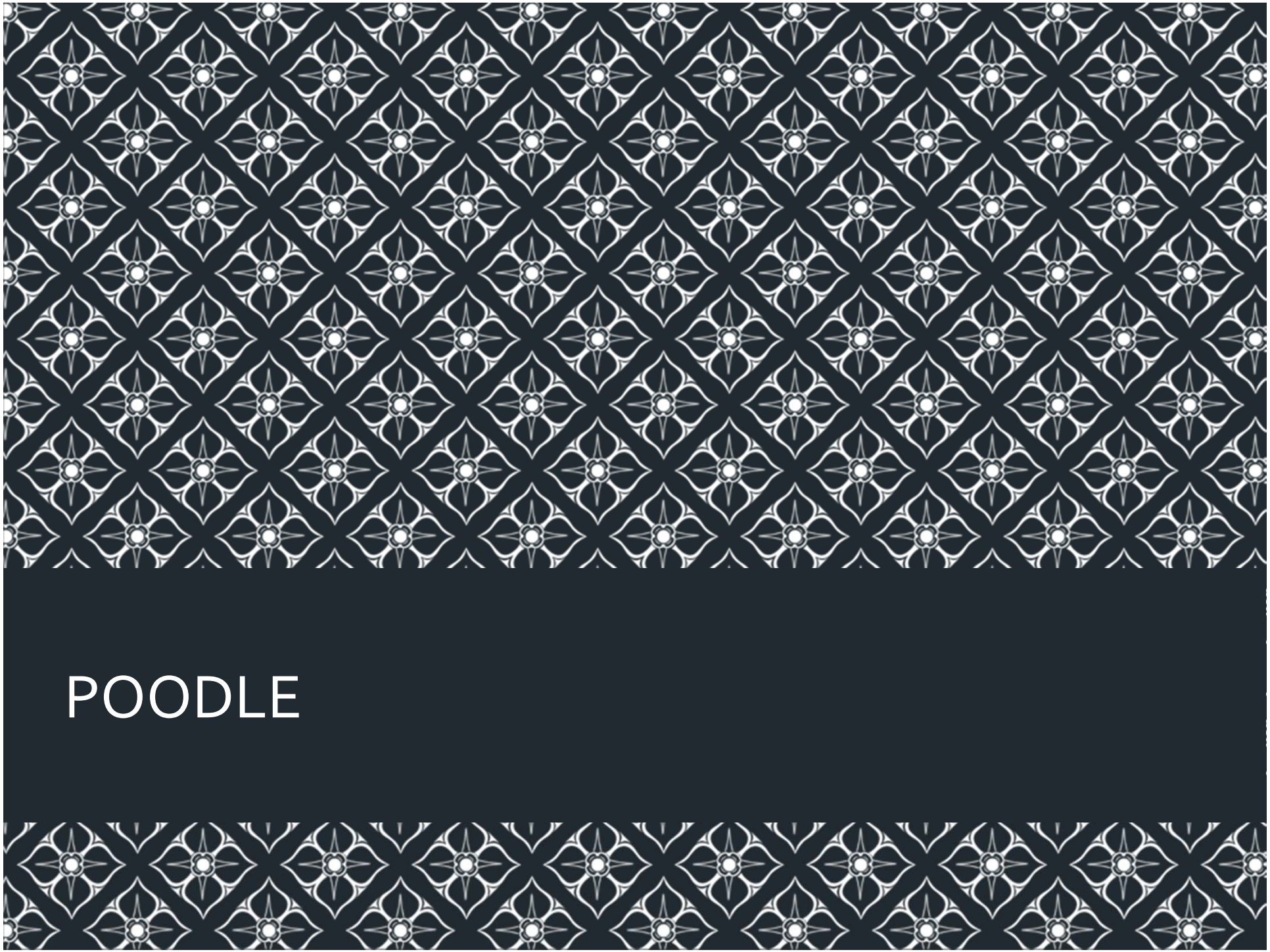
Lucky 13 – Lessons

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.


POODLE

Exploiting Weak Padding Checks – Moeller Attack

Recall SSL/TLS decryption sequence:

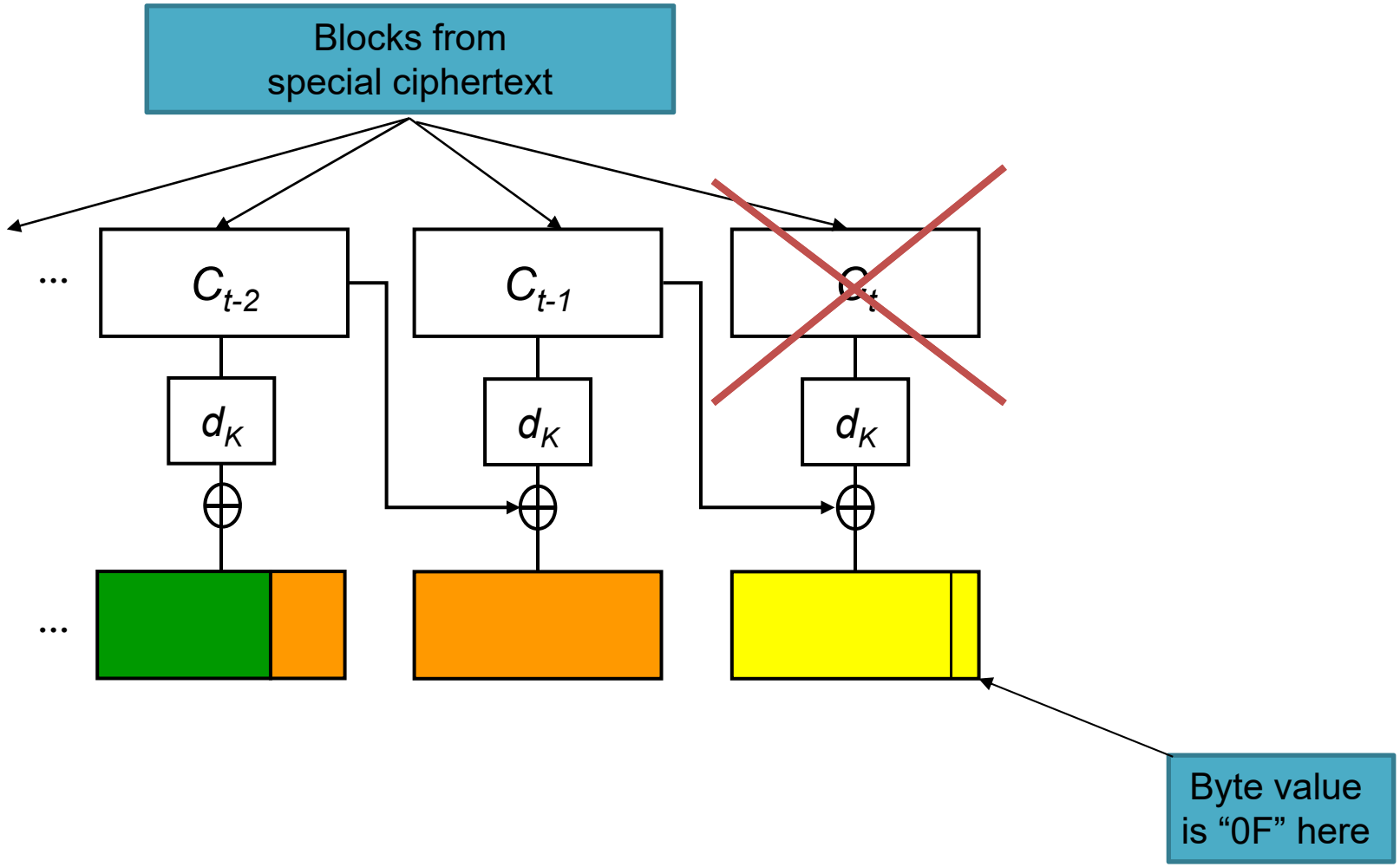
CBC mode decrypt, remove padding, check MAC.

[Moeller, 2002 & 2004]: failure to check padding format leads to a simple attack recovering the last byte of plaintext from any block.

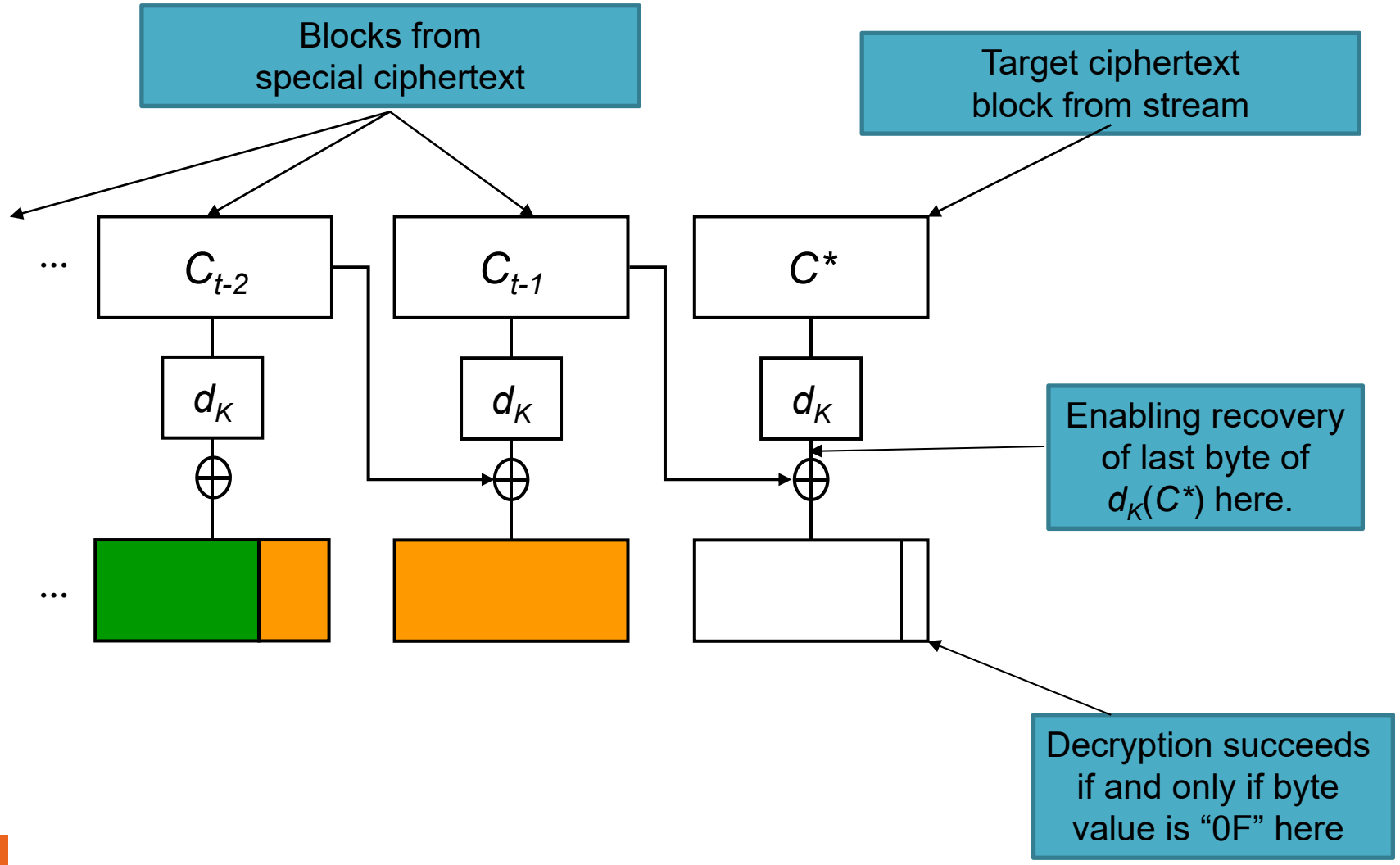
Assumptions:

- 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.

Moeller Attack



Moeller Attack



Moeller Attack

- Decryption succeeds if and only if:

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

- 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:

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

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

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....

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

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.

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.

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

RELATED STORIES

OpenVPN open to pre-auth Bash

POODLE = BEAST techniques + Moeller Attack

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

- In SSLv3, CBC mode encryption uses random padding; only the last byte is used to remove padding.
- In other words, you can't check the padding format!

Repeat:

1. Use Javascript in the browser to pad HTTP GET requests (as in BEAST), ensuring that the 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?

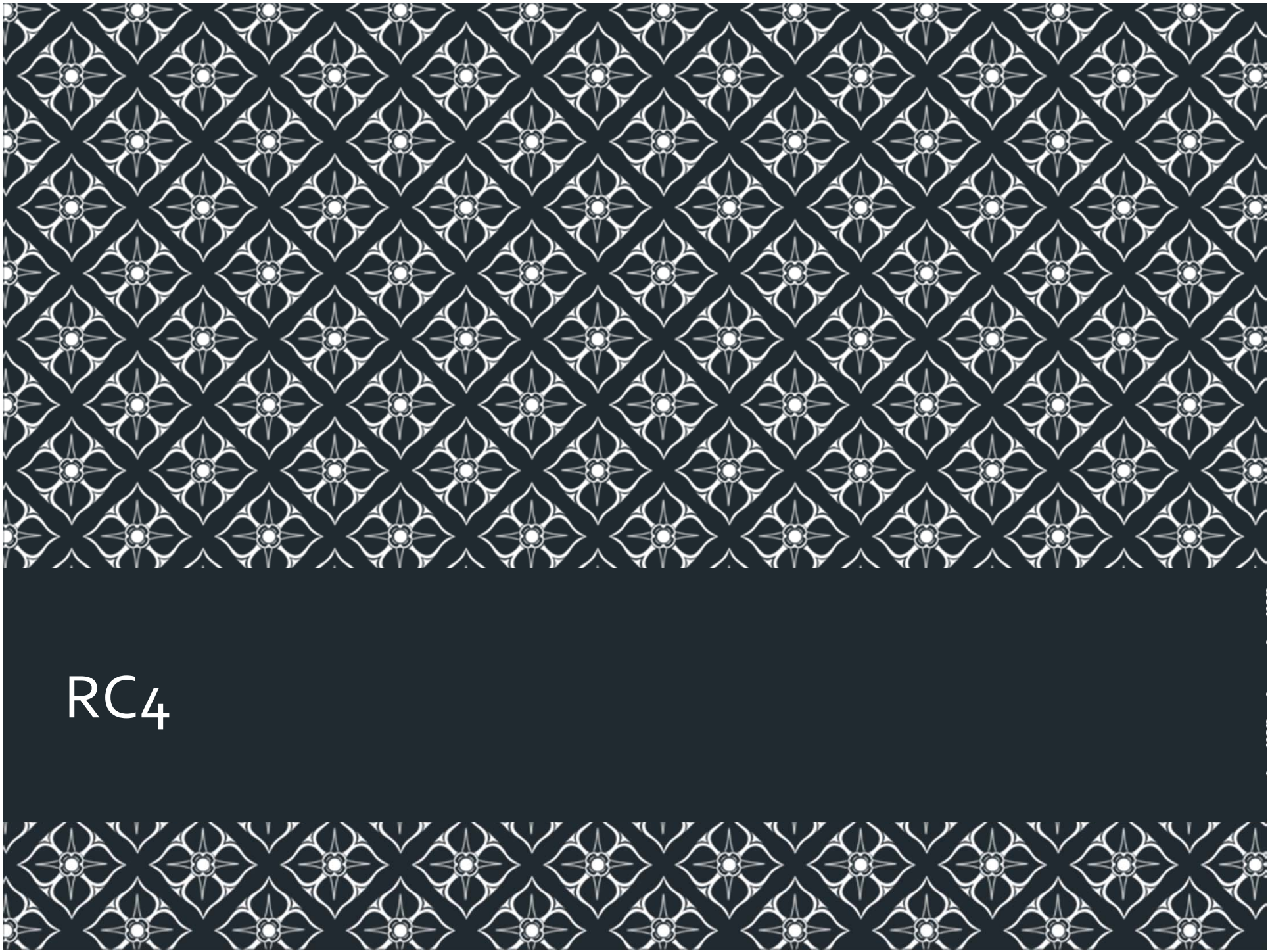
A patch that does not work: upgrade decryption to do full padding check and Lucky 13 protection.

- 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!

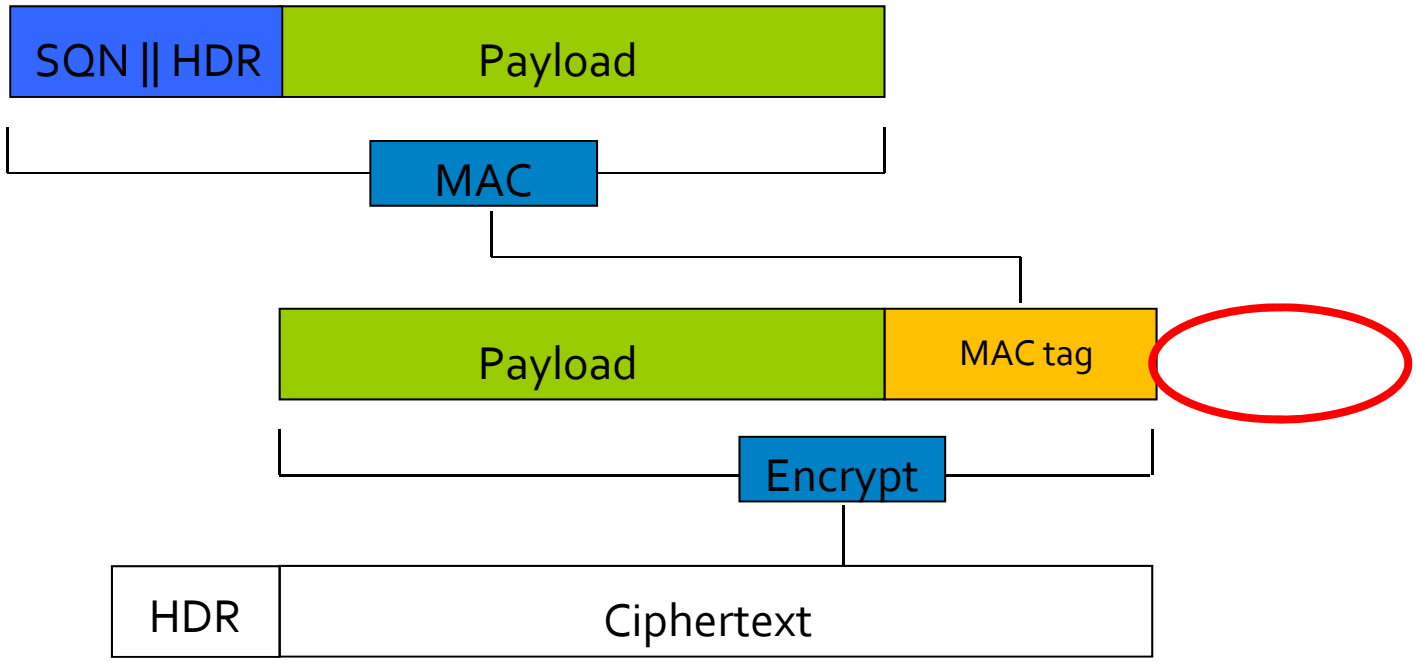
POODLE and SSL/TLS Fallback

The attack is made worse because of the active version downgrade attack, aka SSL/TLS falbback.

- An active MITM attacker could always force client and server to downgrade to SSLv3.
- Because the SSL/TLS version negotiation process is not stateful and was not cryptographically protected across fallbacks.
- Various countermeasures now exist, but the safest option was to stop supporting SSLv3.

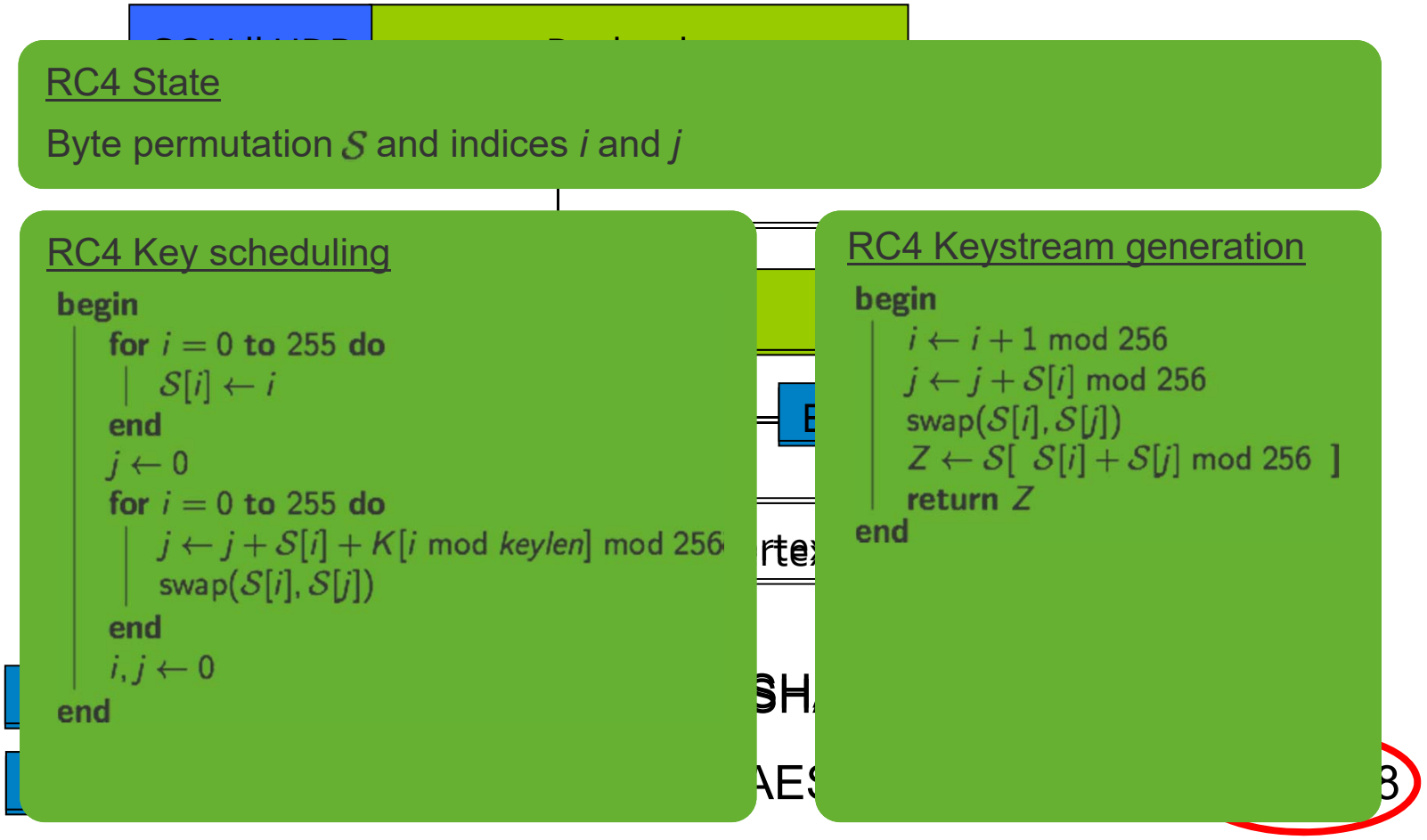

RC4

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 cipher suites 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 cipher suites

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 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'"

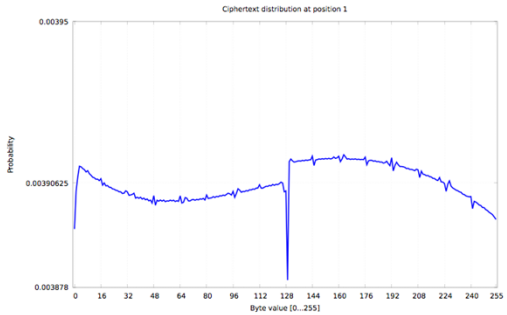
"Google uses it, so it must be OK for my site".

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

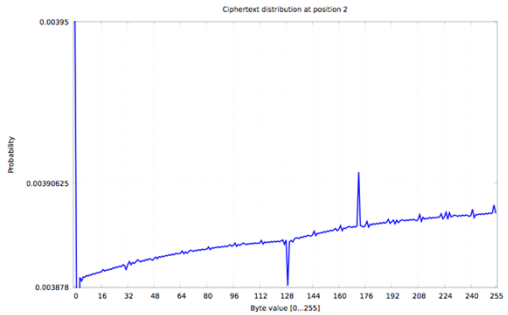
Complete Keystream Byte Distributions

Approach in [ABPPS13]:

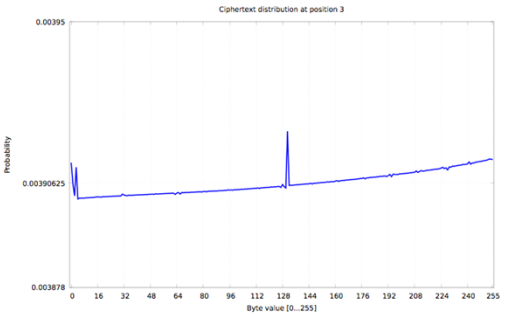
Based on the output from 2^{45} random independent 128-bit RC4 keys, we estimated the keystream byte distributions for the first 256 bytes



Z_1



Z_2



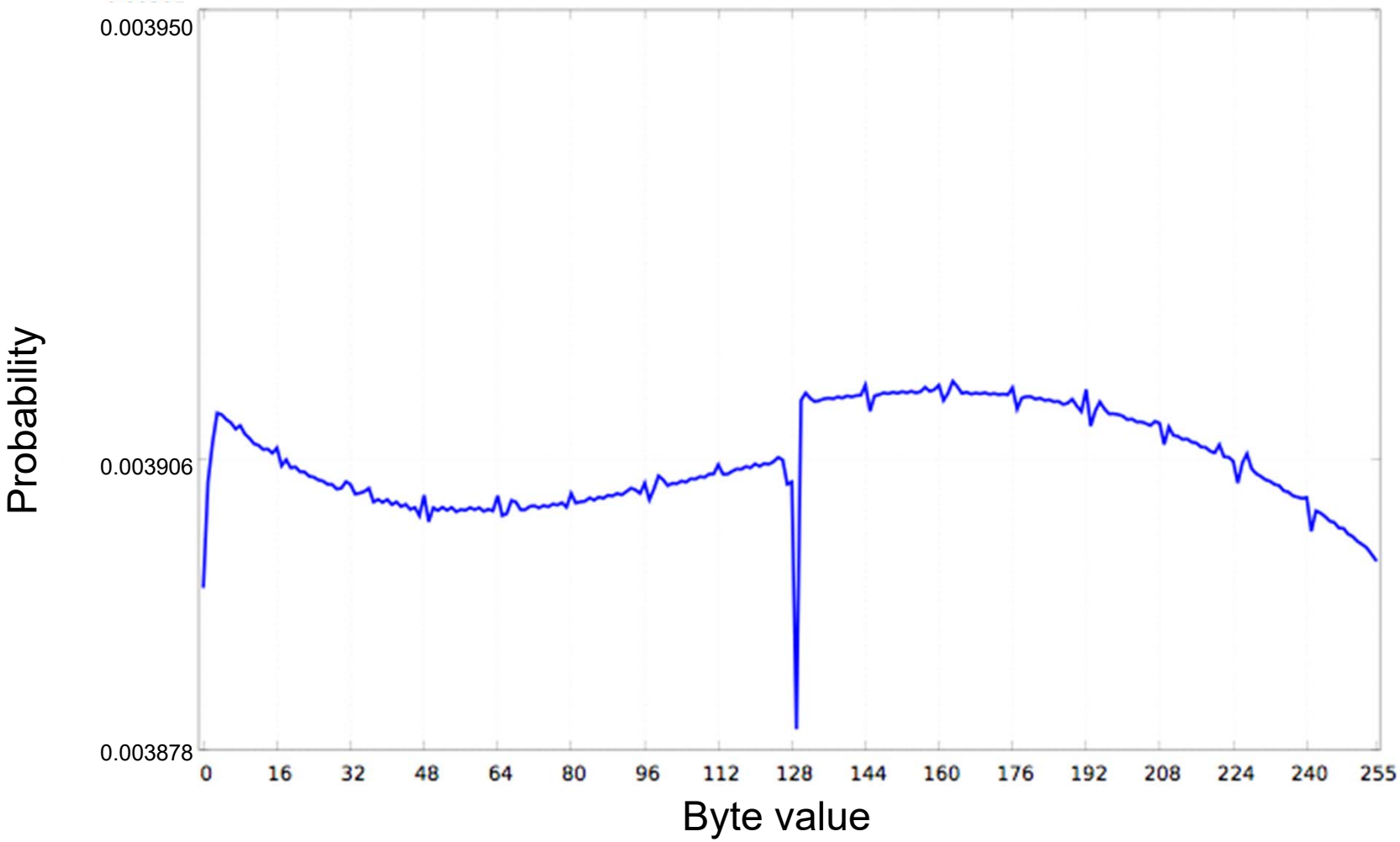
Z_3

...

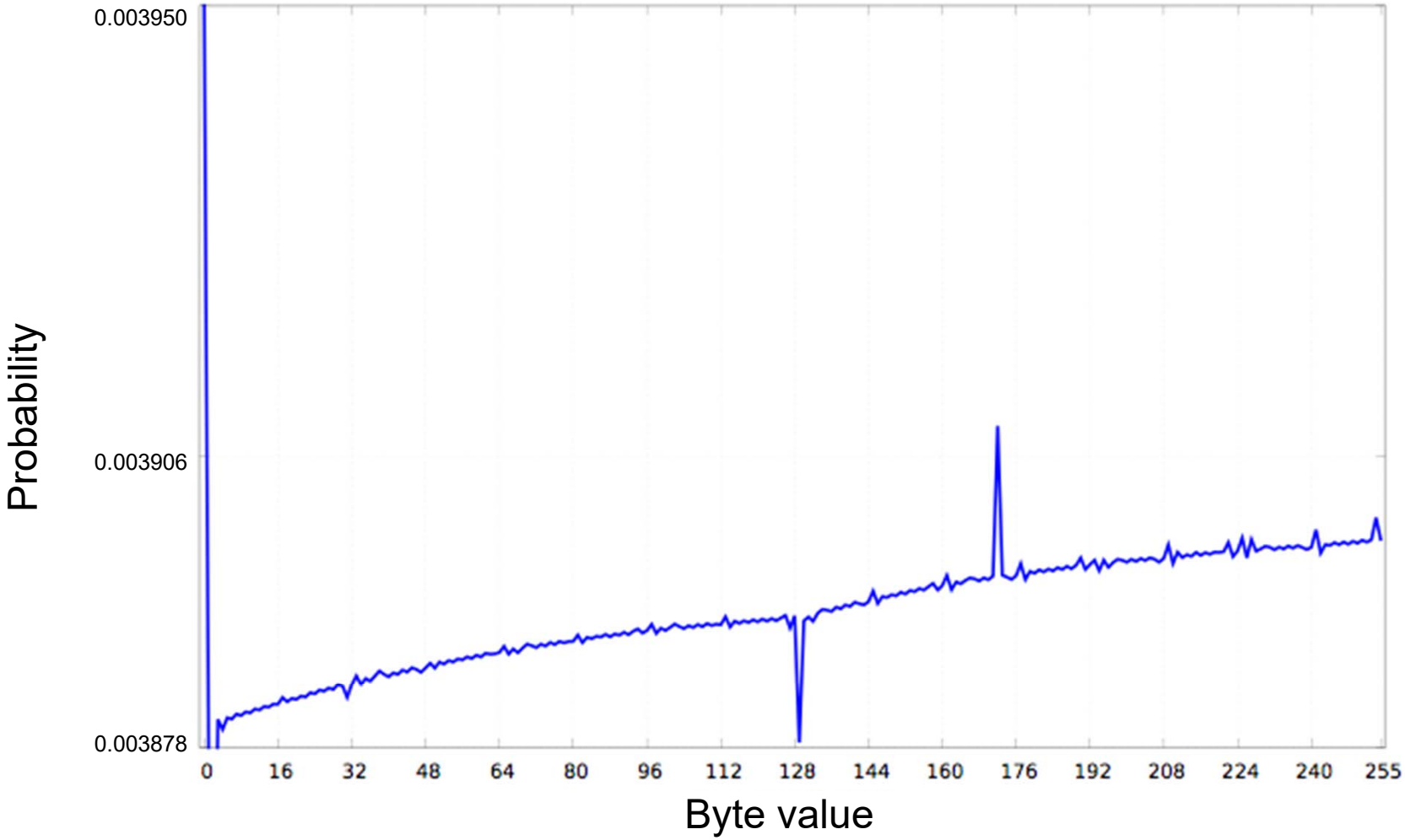
...

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

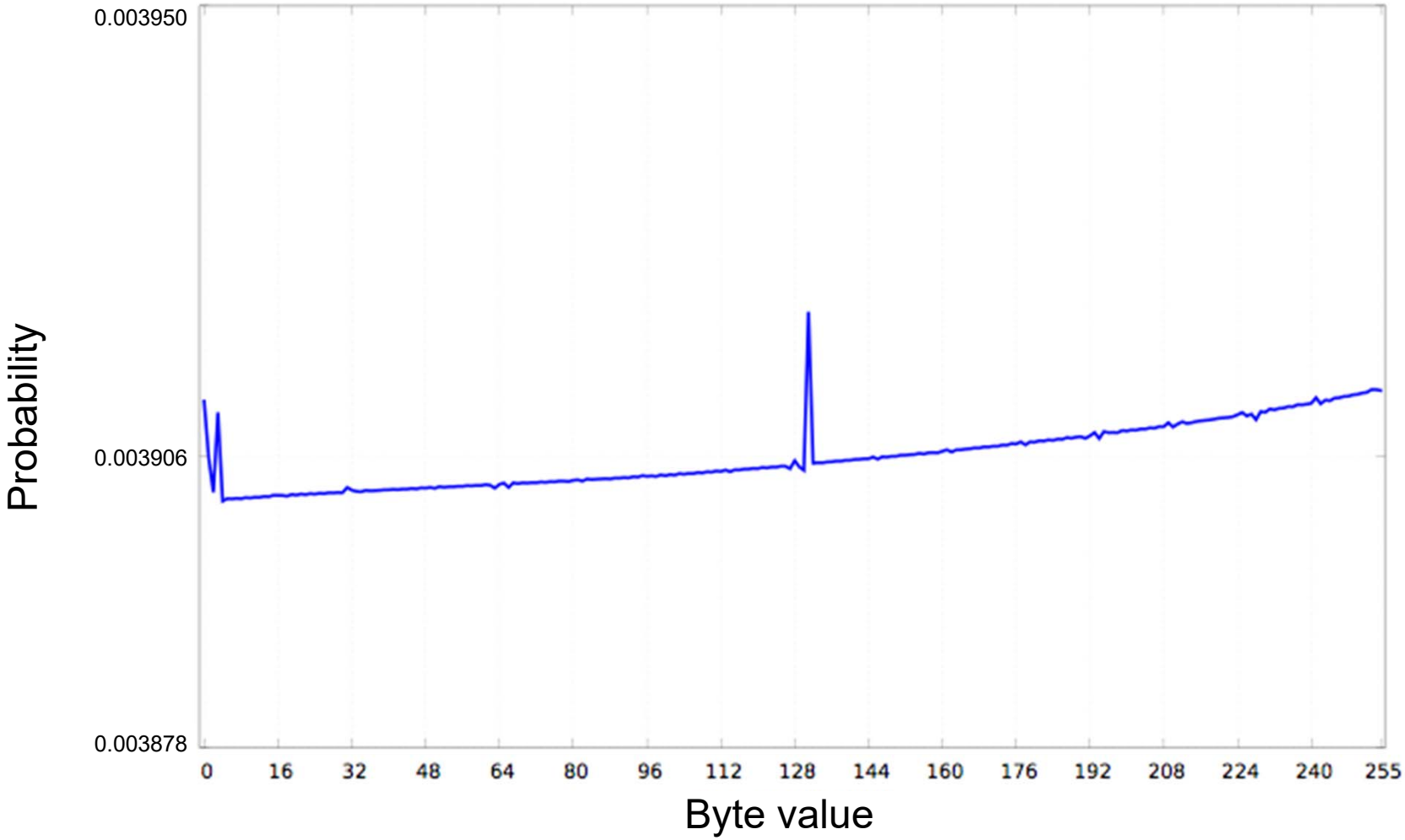
Keystream Distribution at Position 1



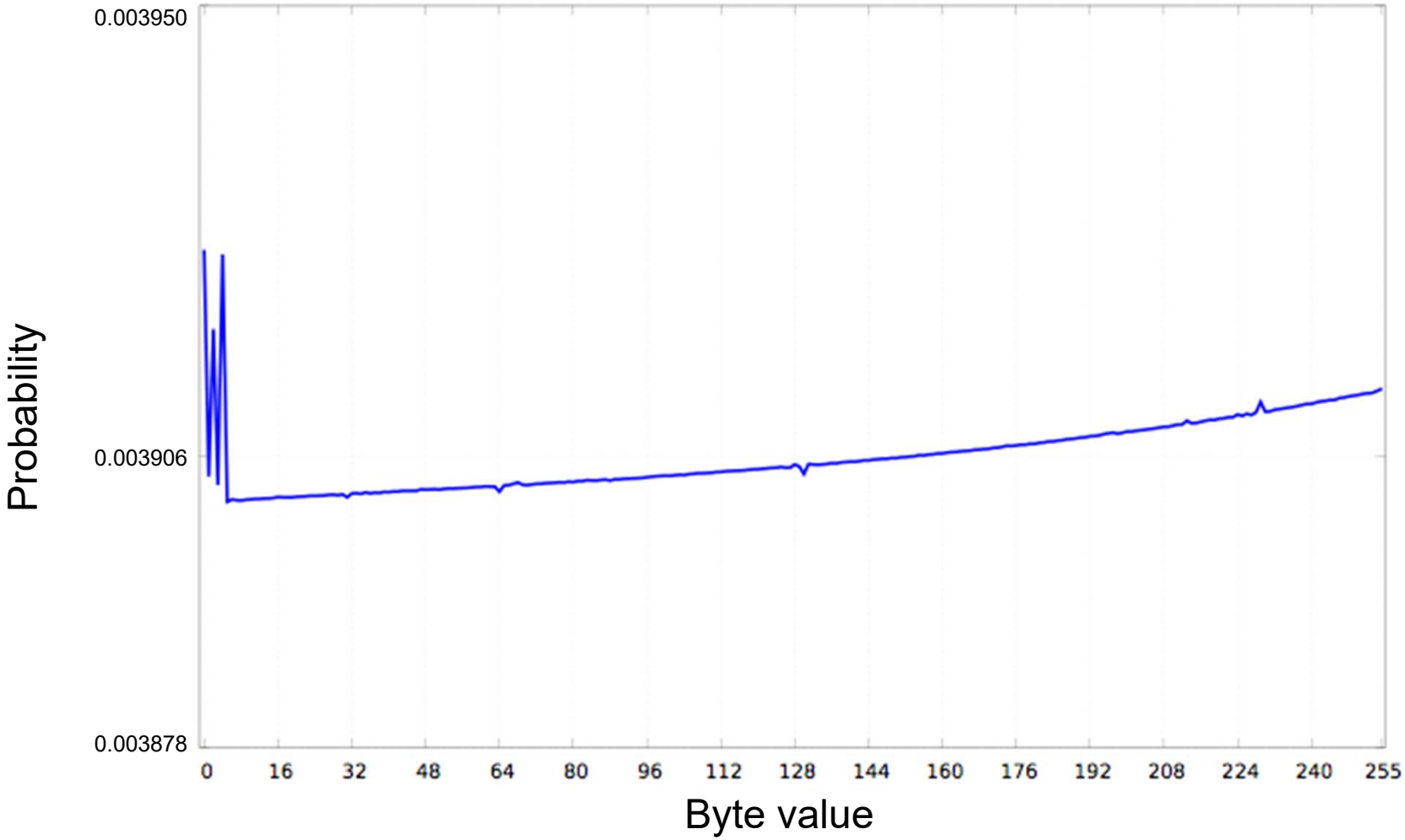
Keystream Distribution at Position 2



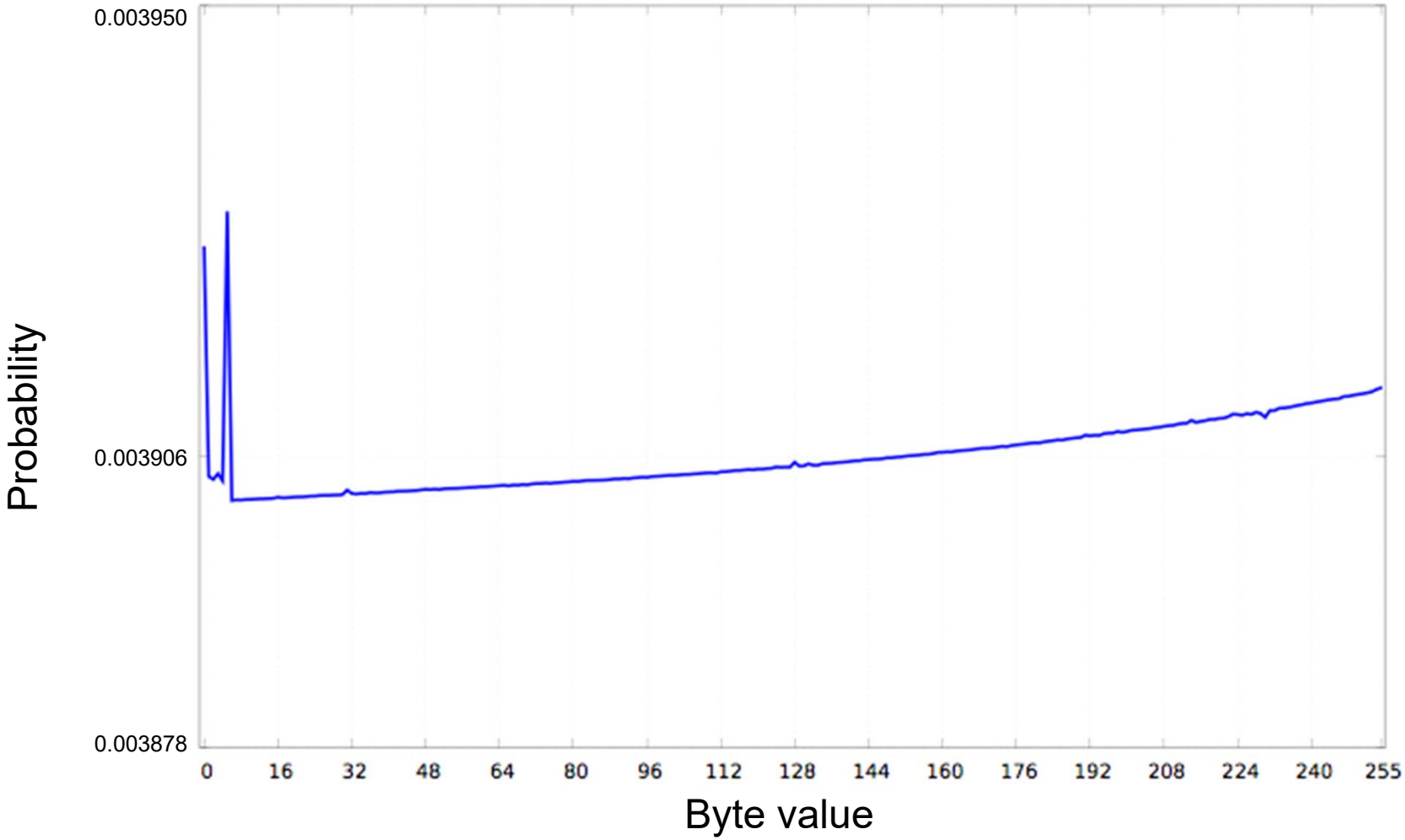
Keystream Distribution at Position 3



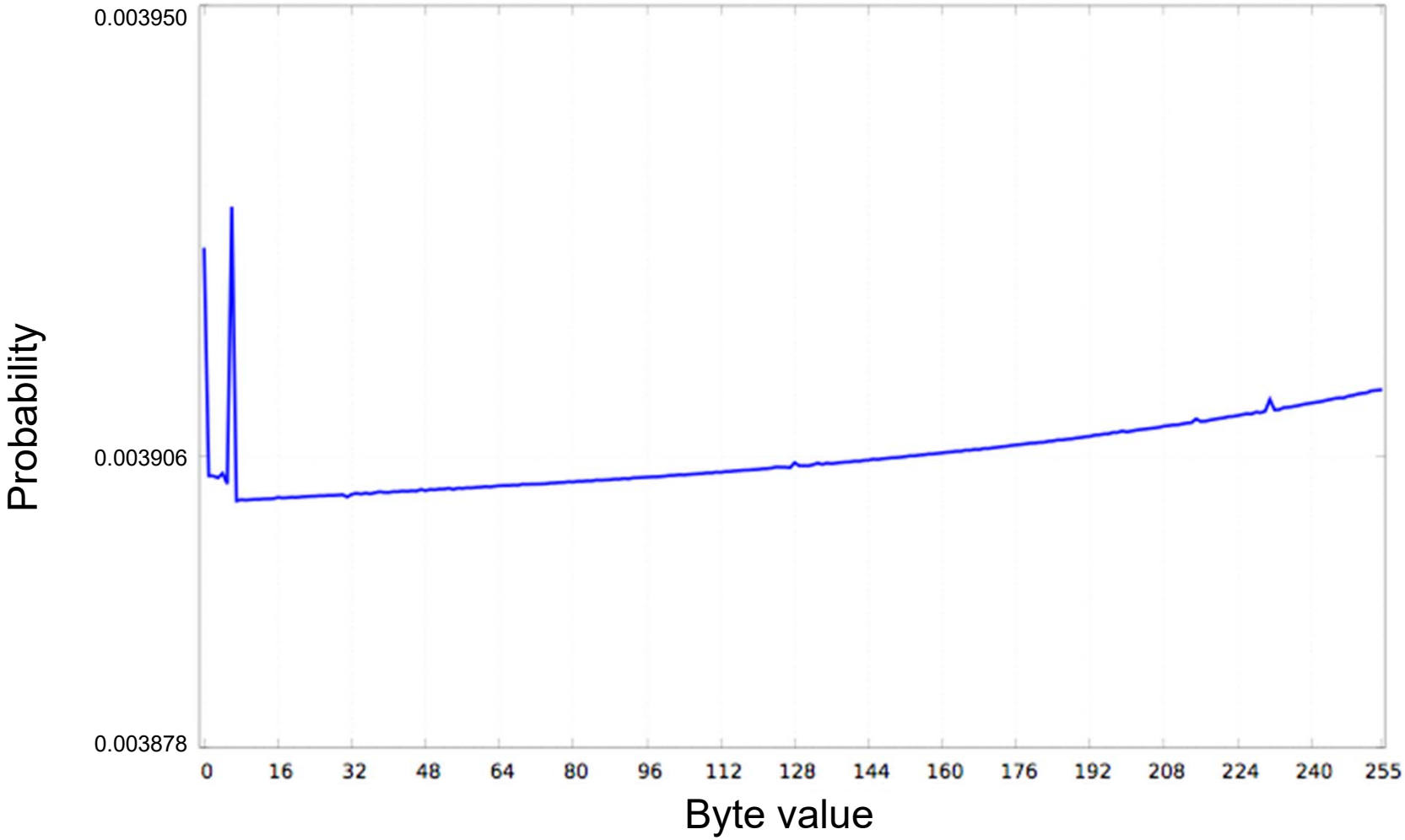
Keystream Distribution at Position 4



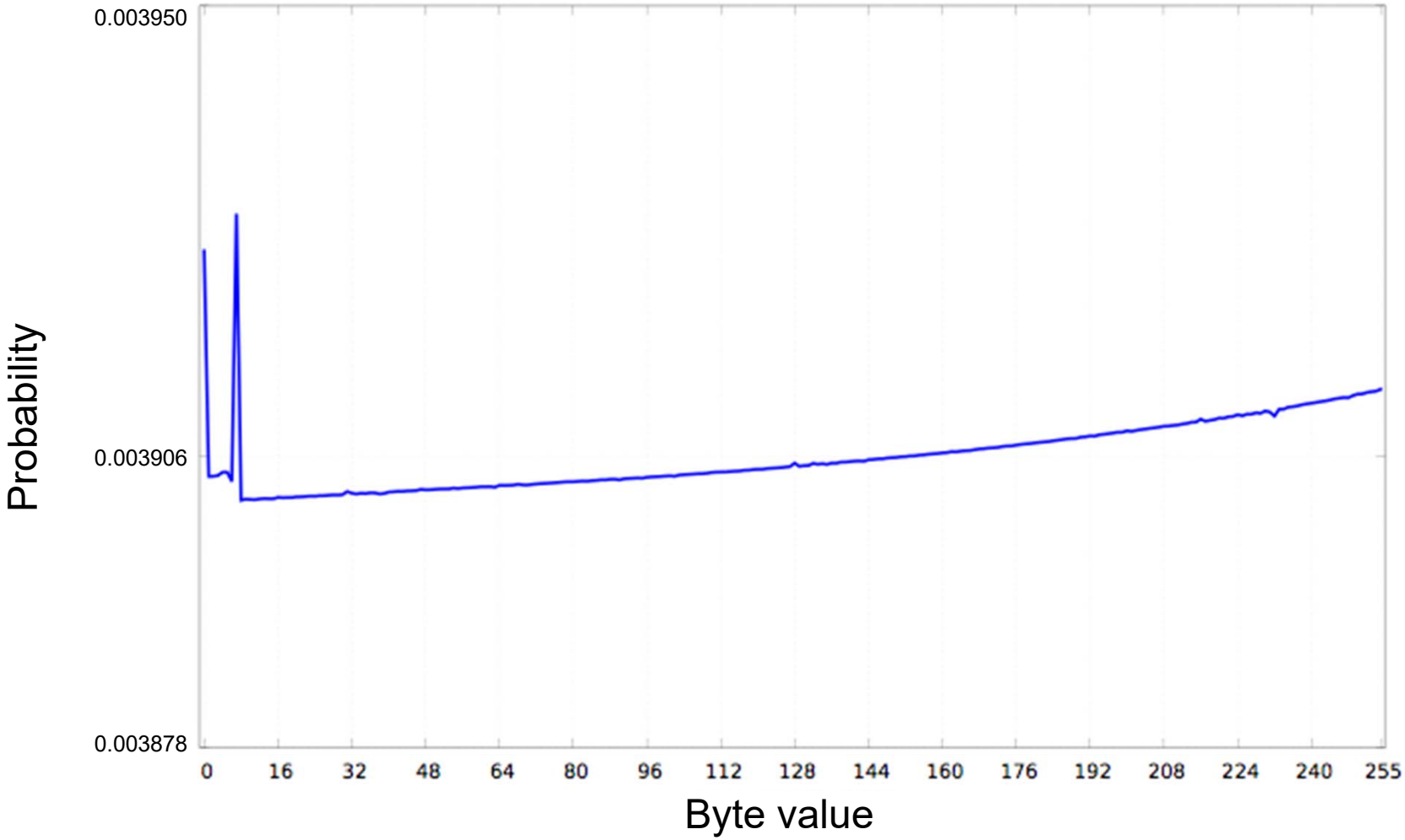
Keystream Distribution at Position 5



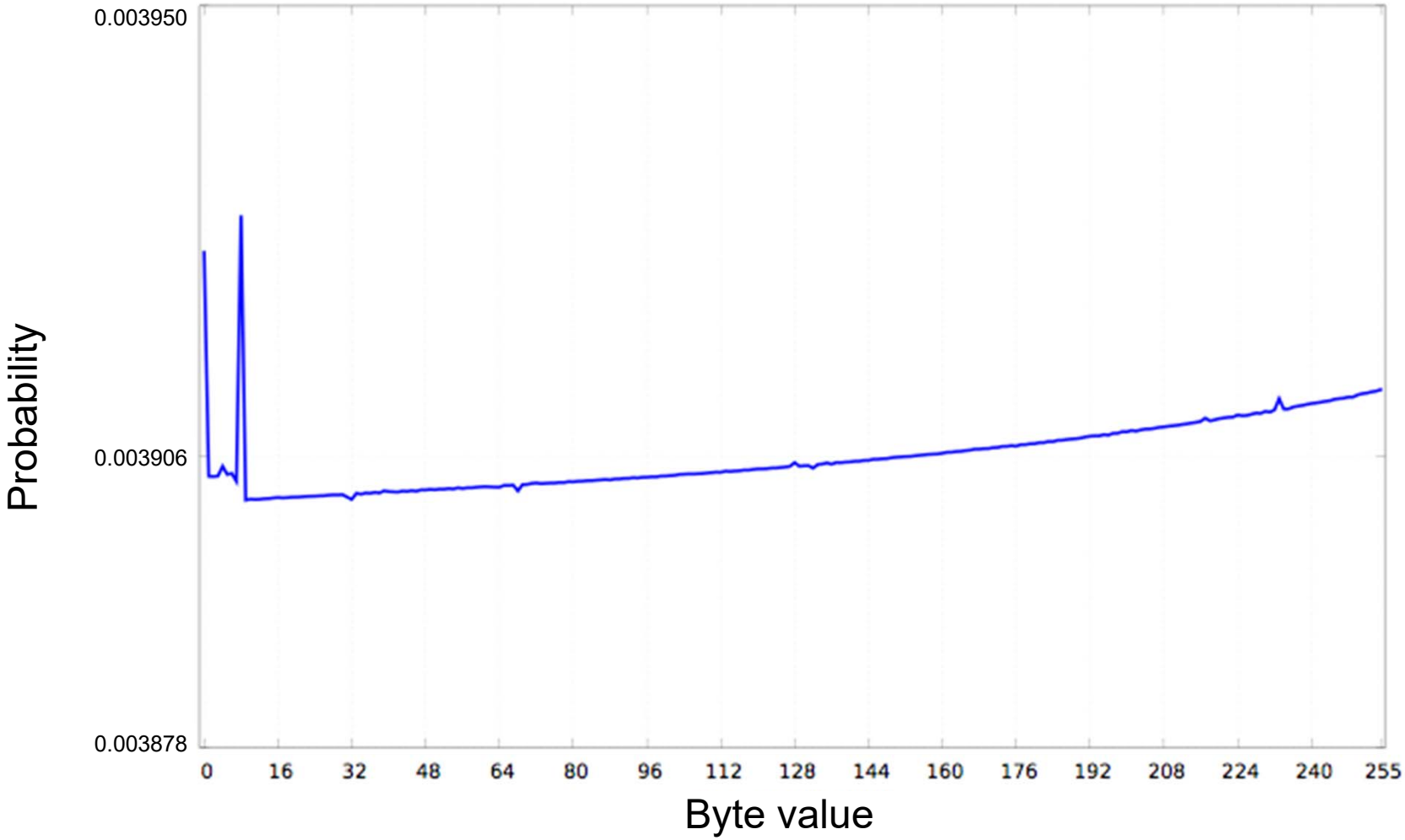
Keystream Distribution at Position 6



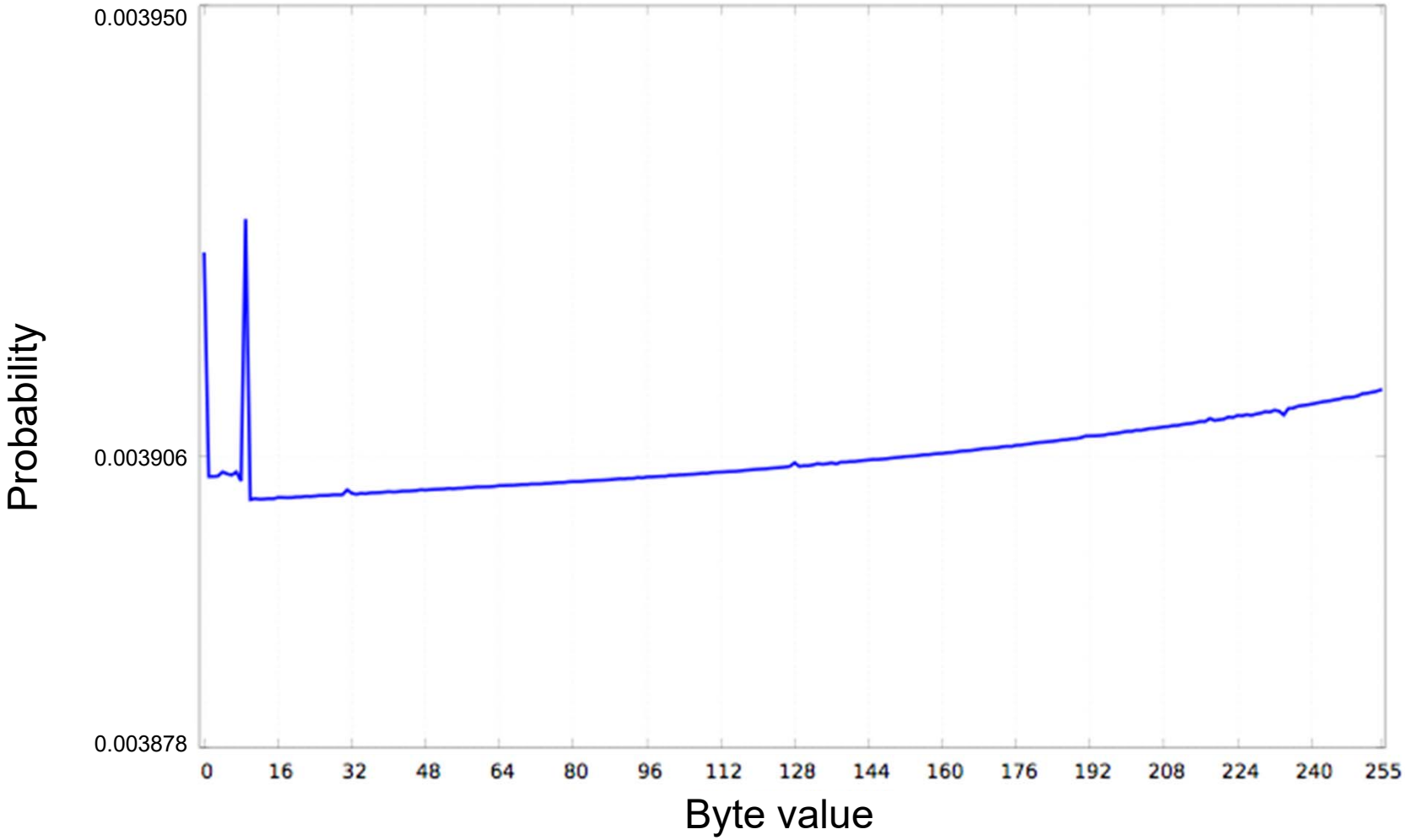
Keystream Distribution at Position 7



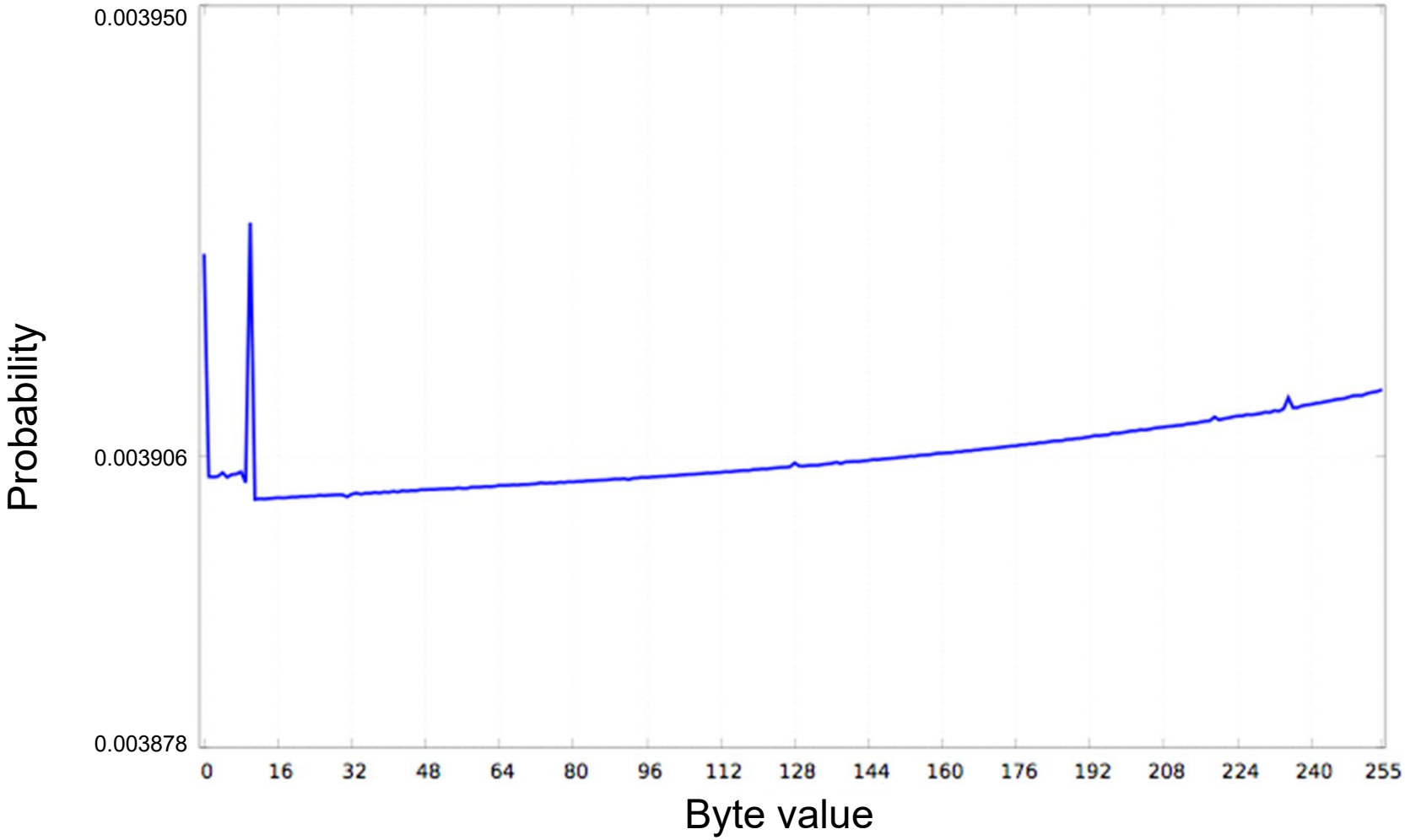
Keystream Distribution at Position 8



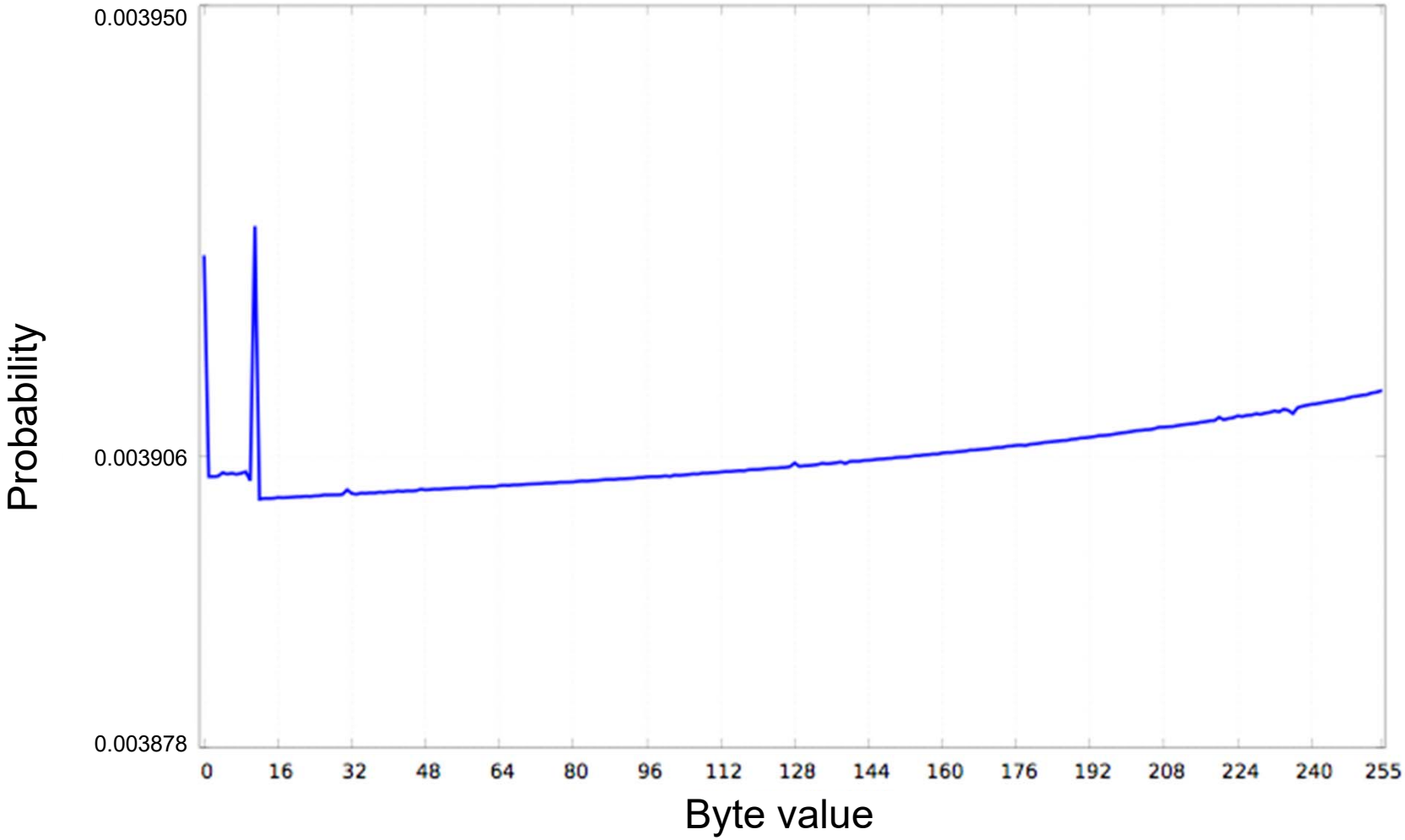
Keystream Distribution at Position 9



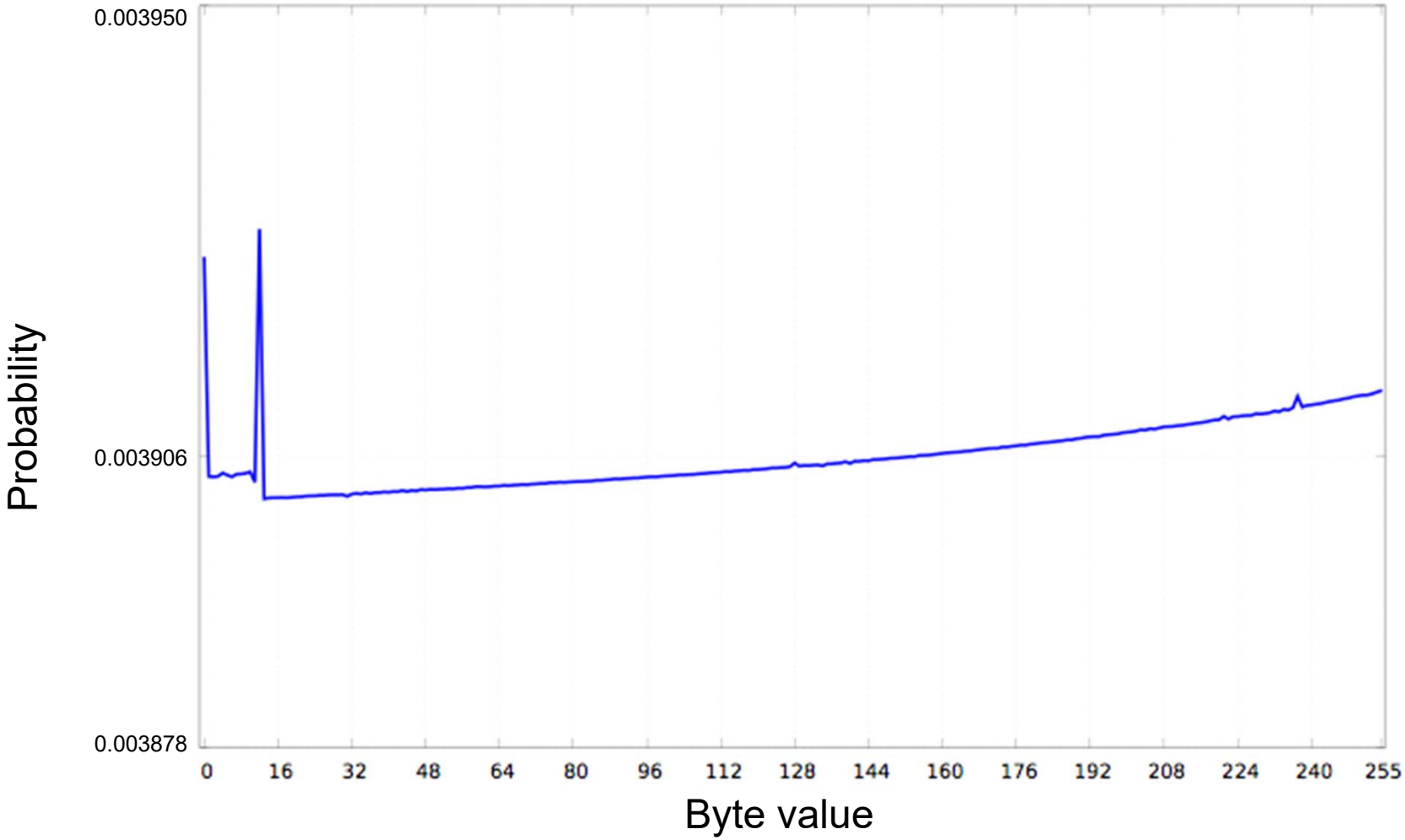
Keystream Distribution at Position 10



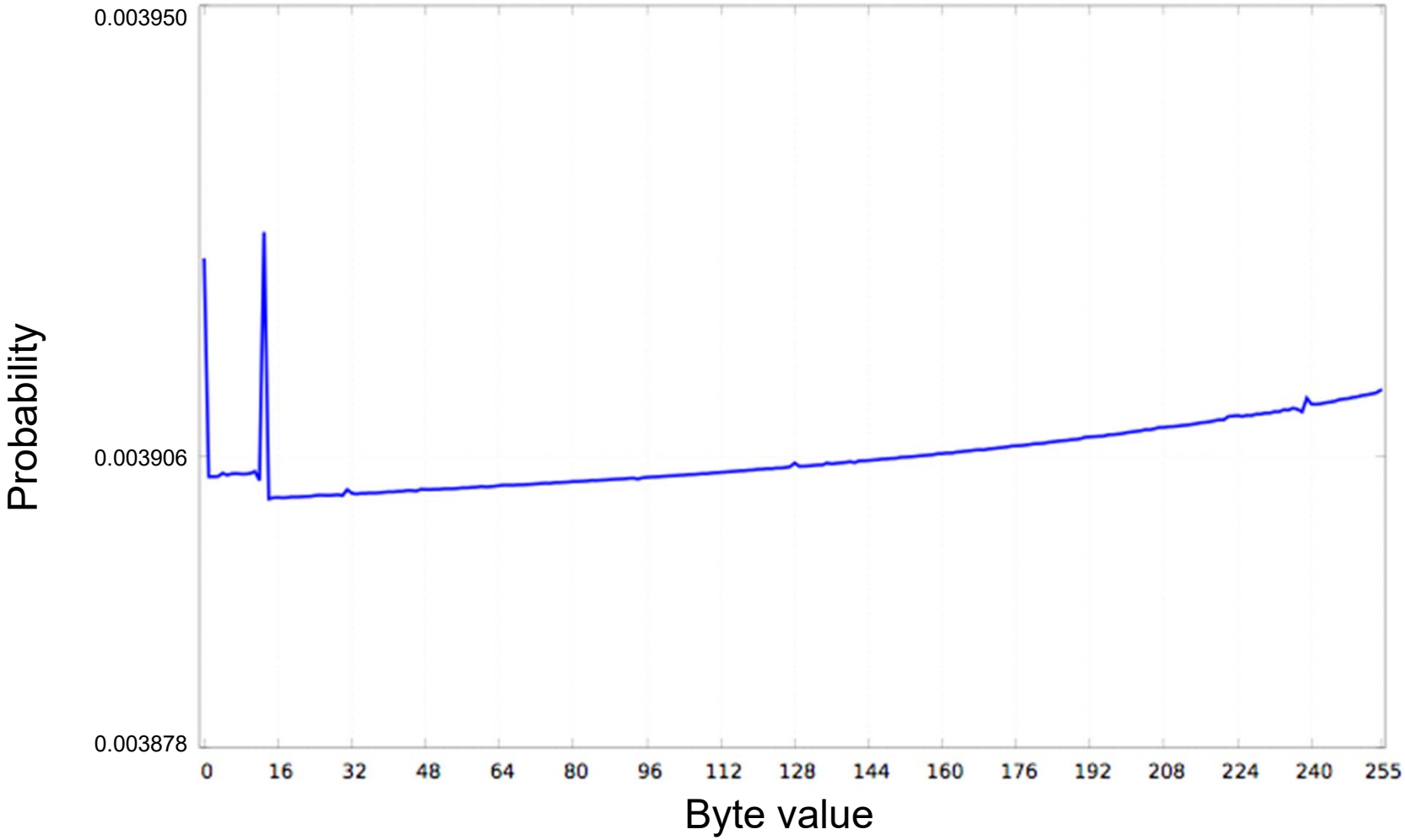
Keystream Distribution at Position 11



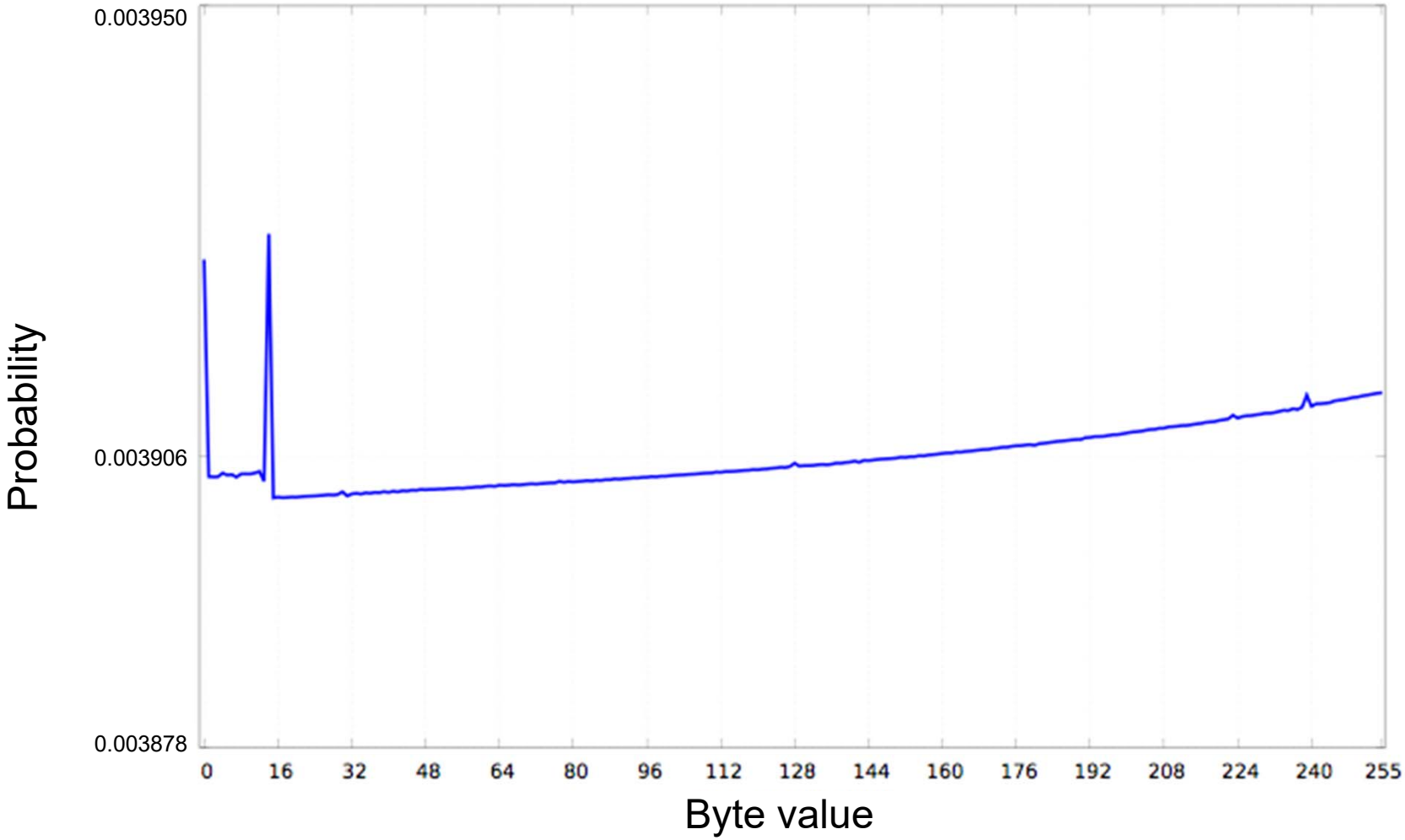
Keystream Distribution at Position 12



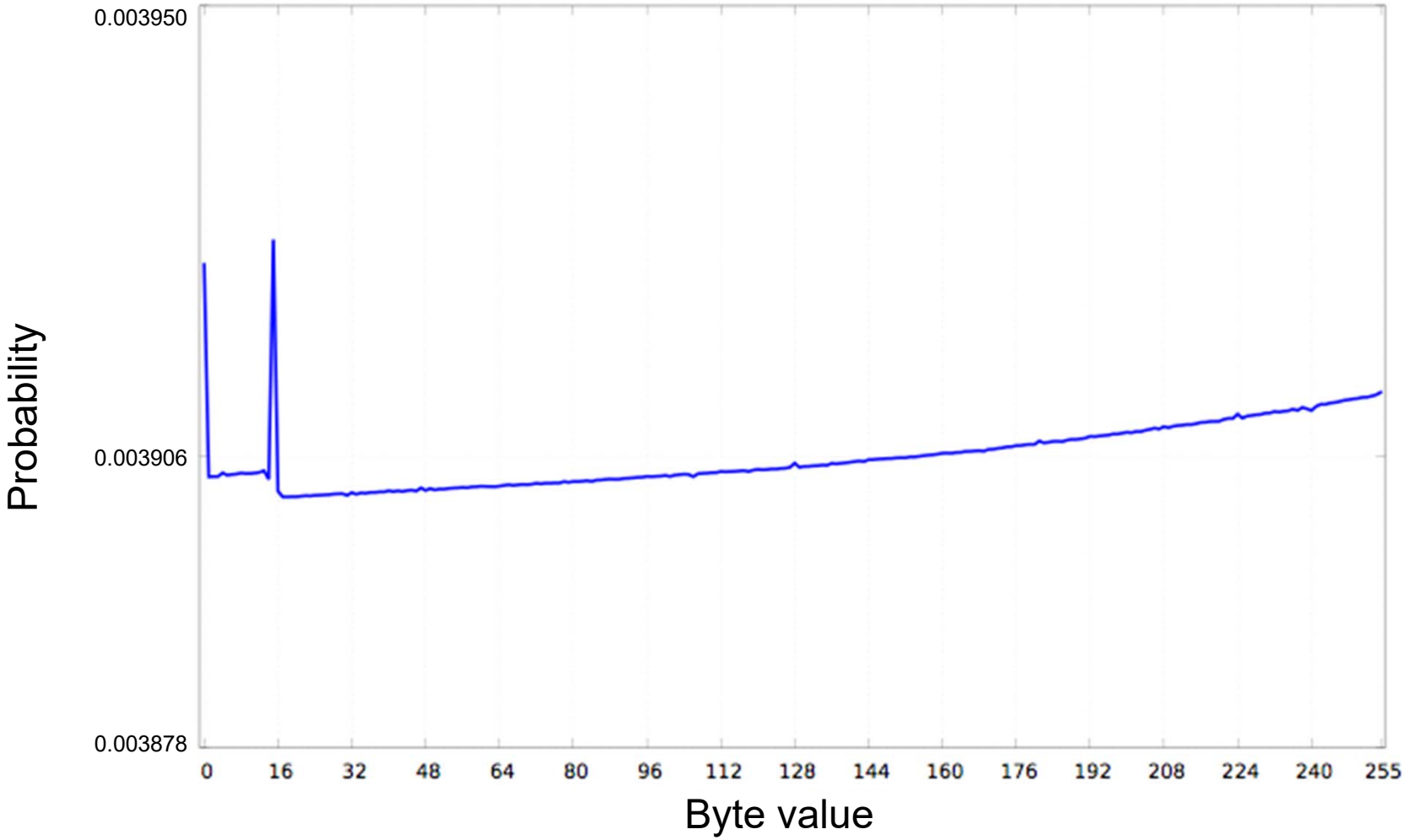
Keystream Distribution at Position 13



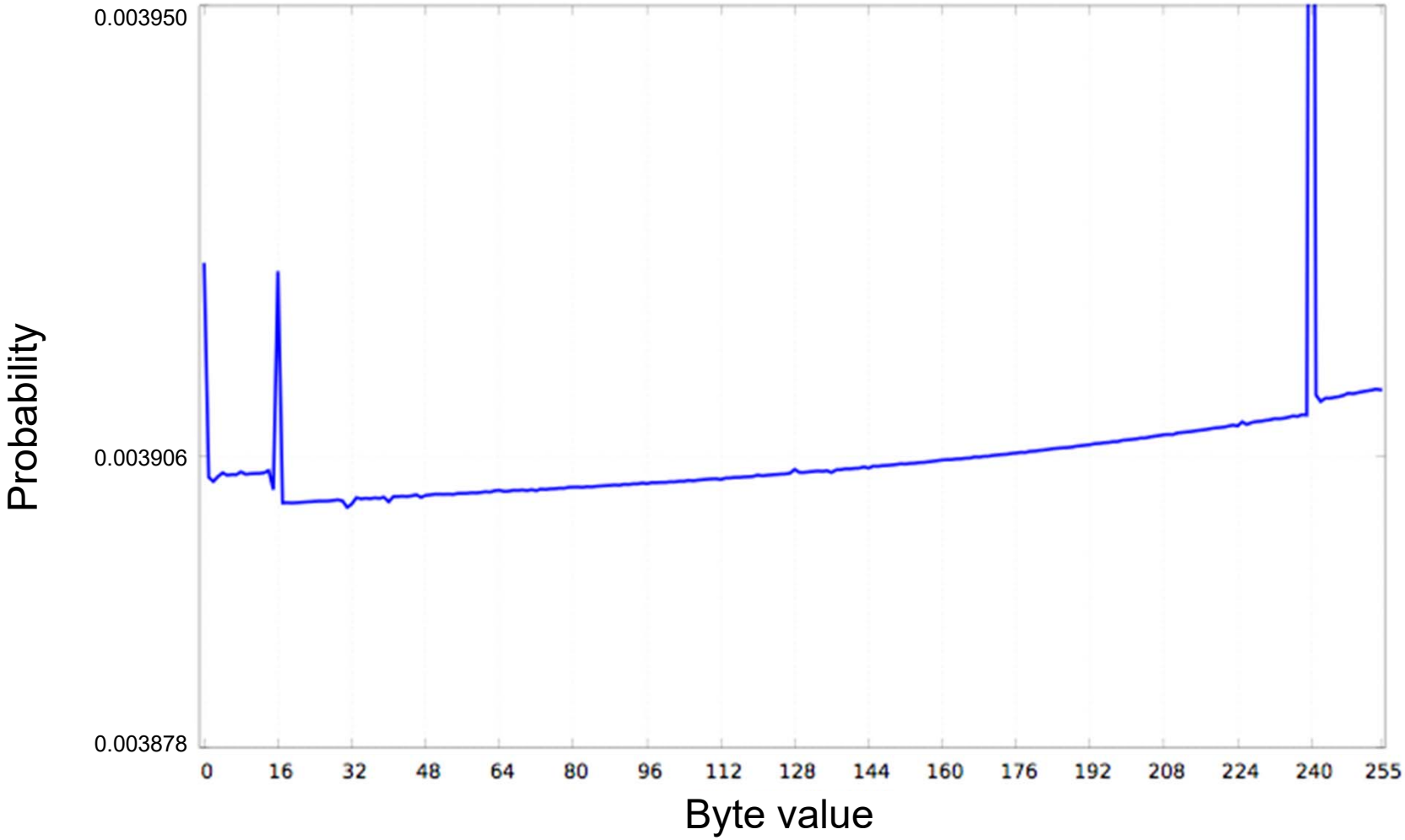
Keystream Distribution at Position 14



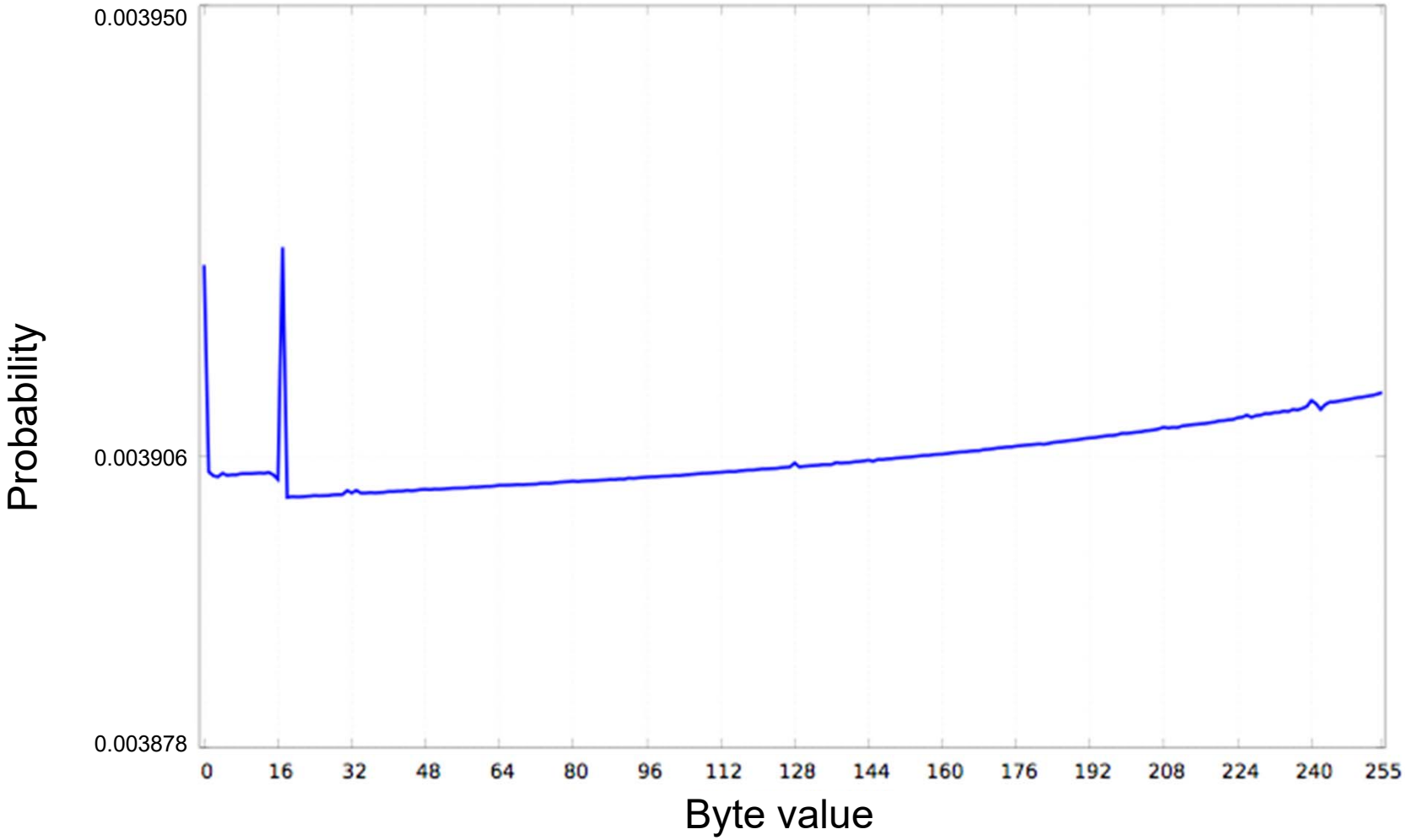
Keystream Distribution at Position 15



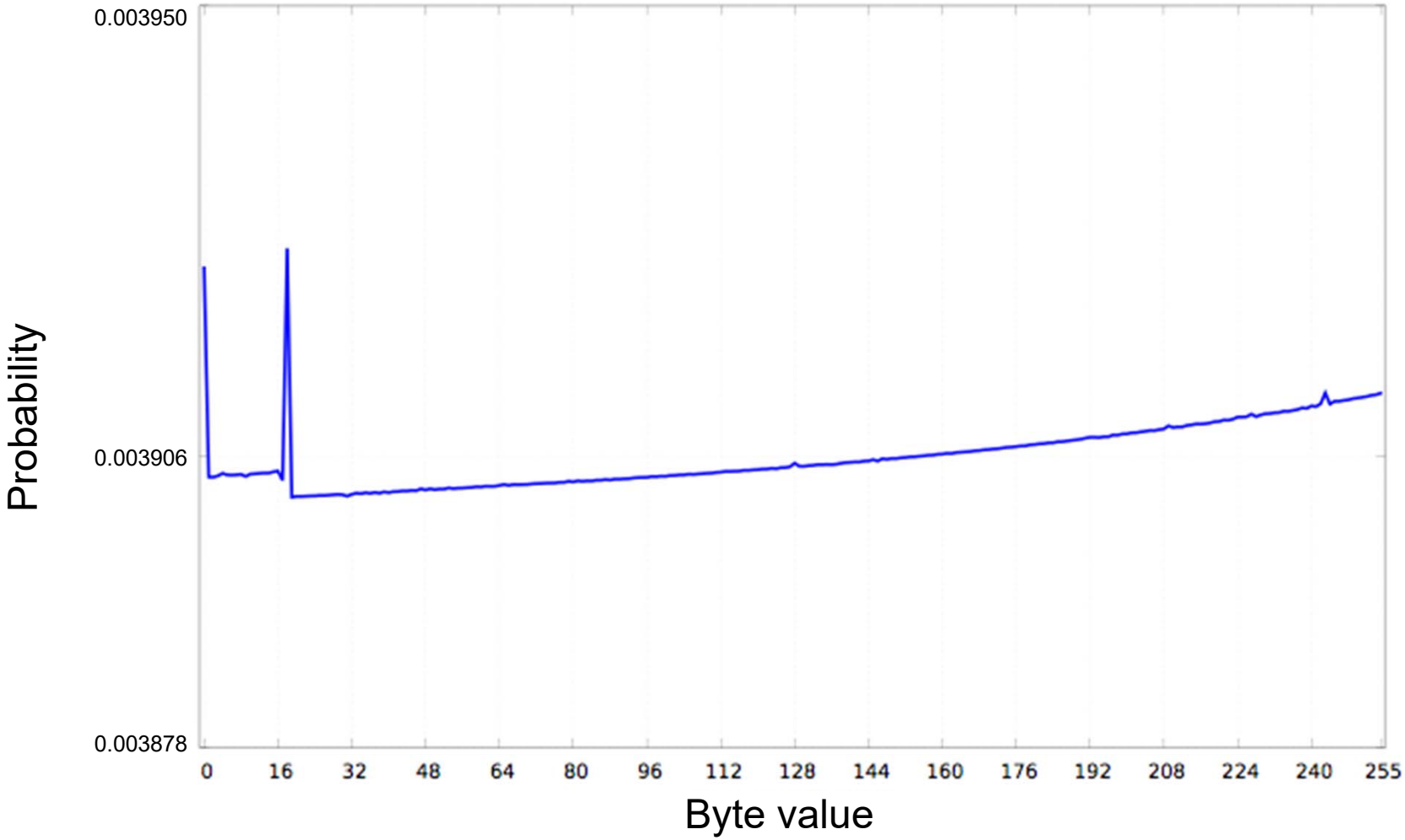
Keystream Distribution at Position 16



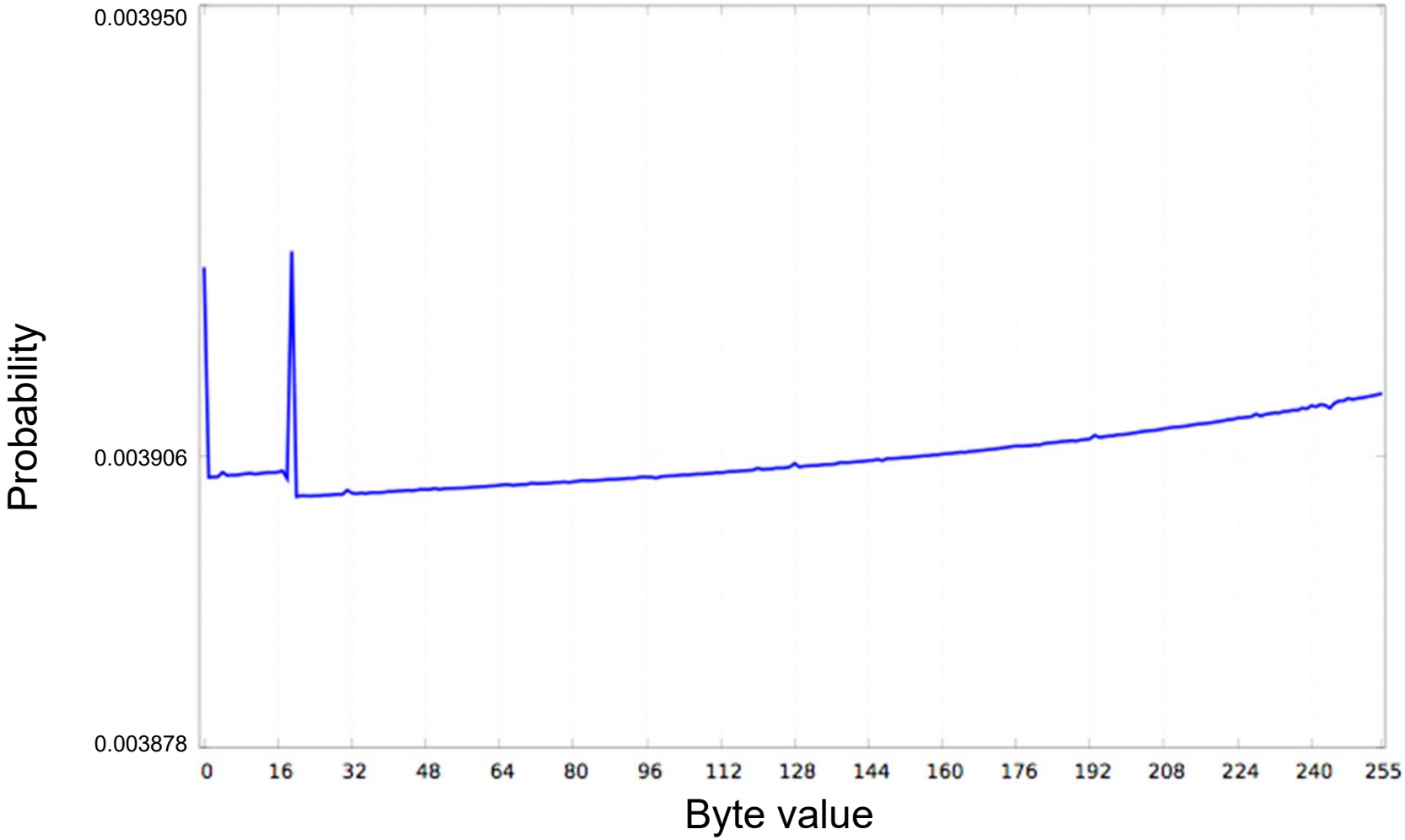
Keystream Distribution at Position 17



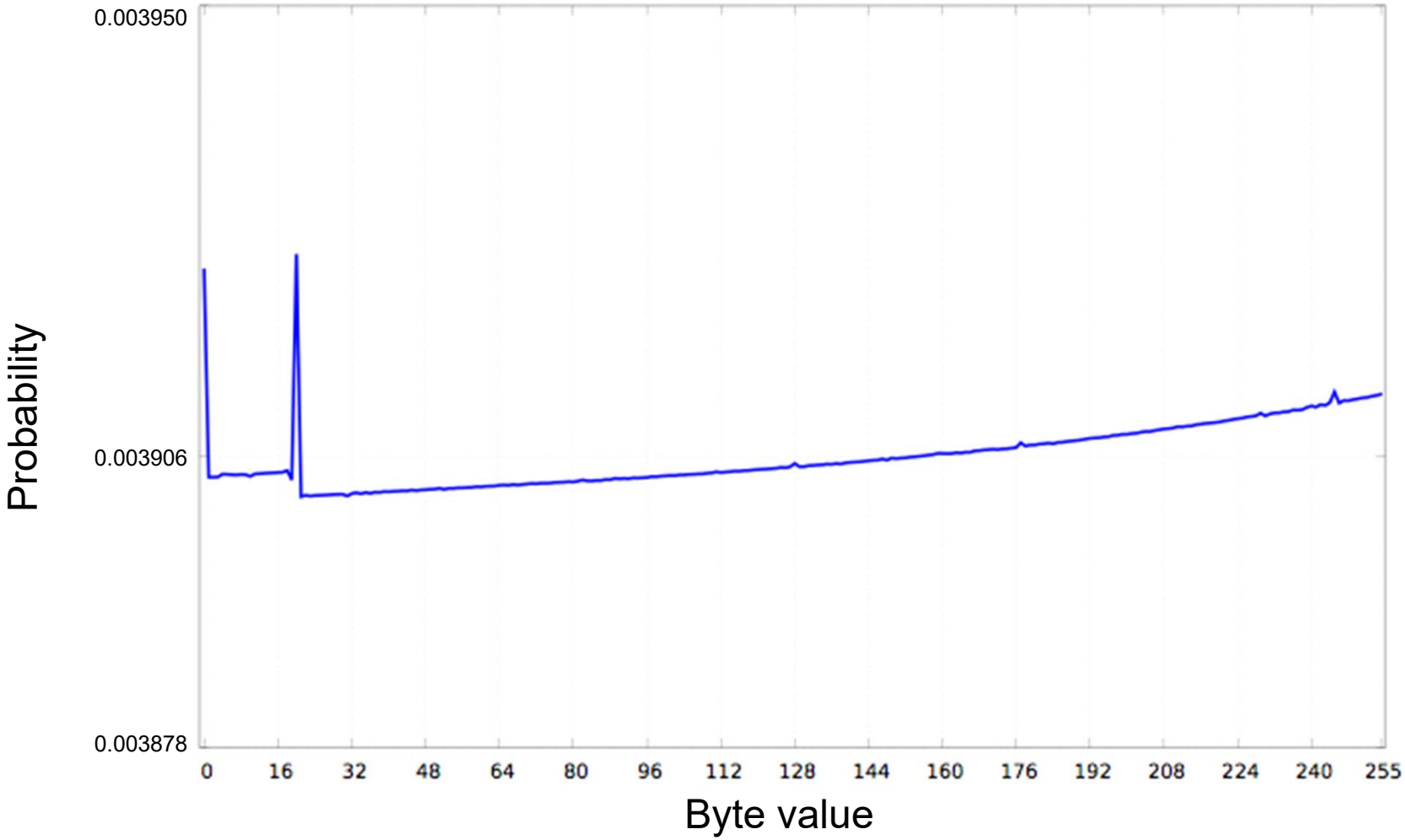
Keystream Distribution at Position 18



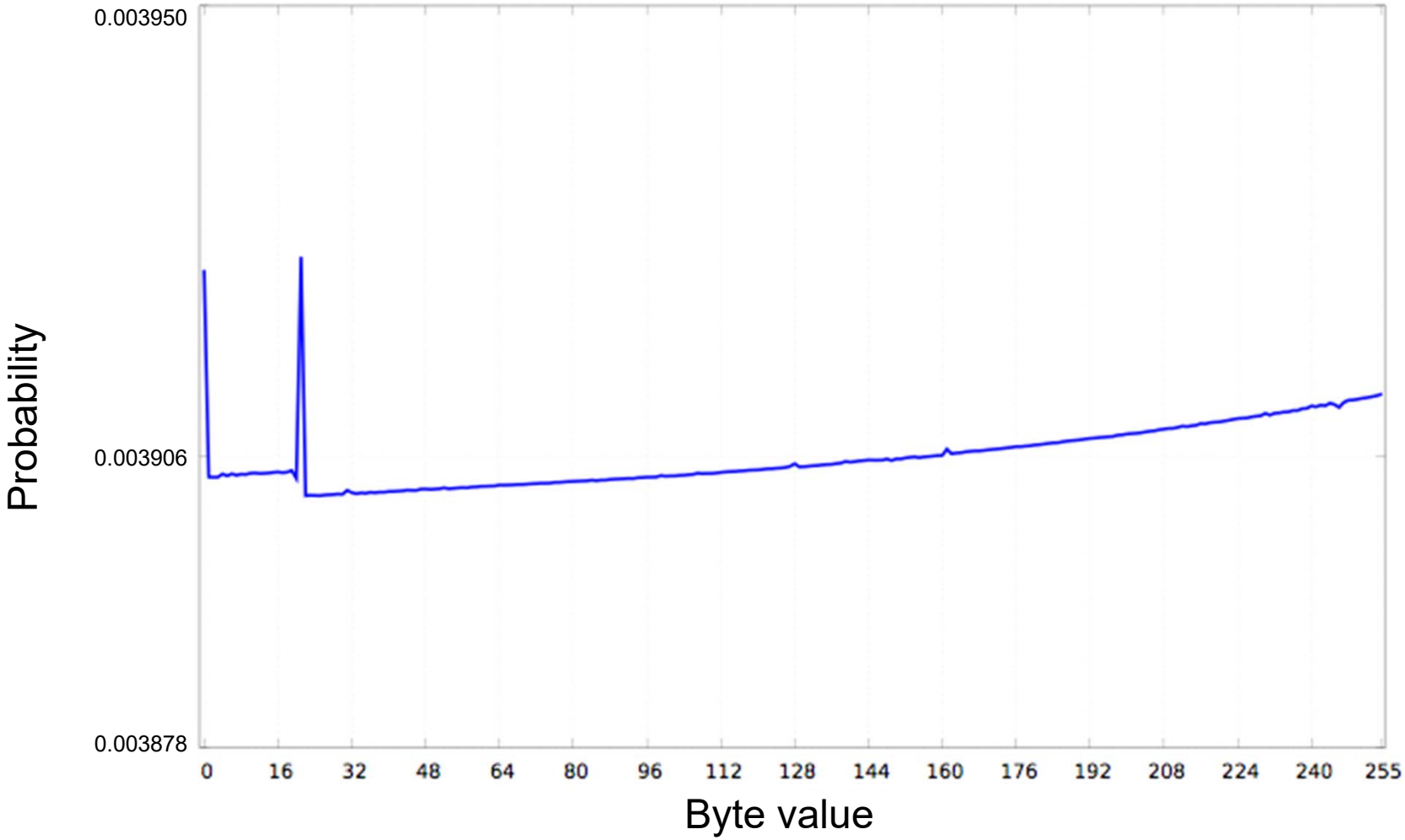
Keystream Distribution at Position 19



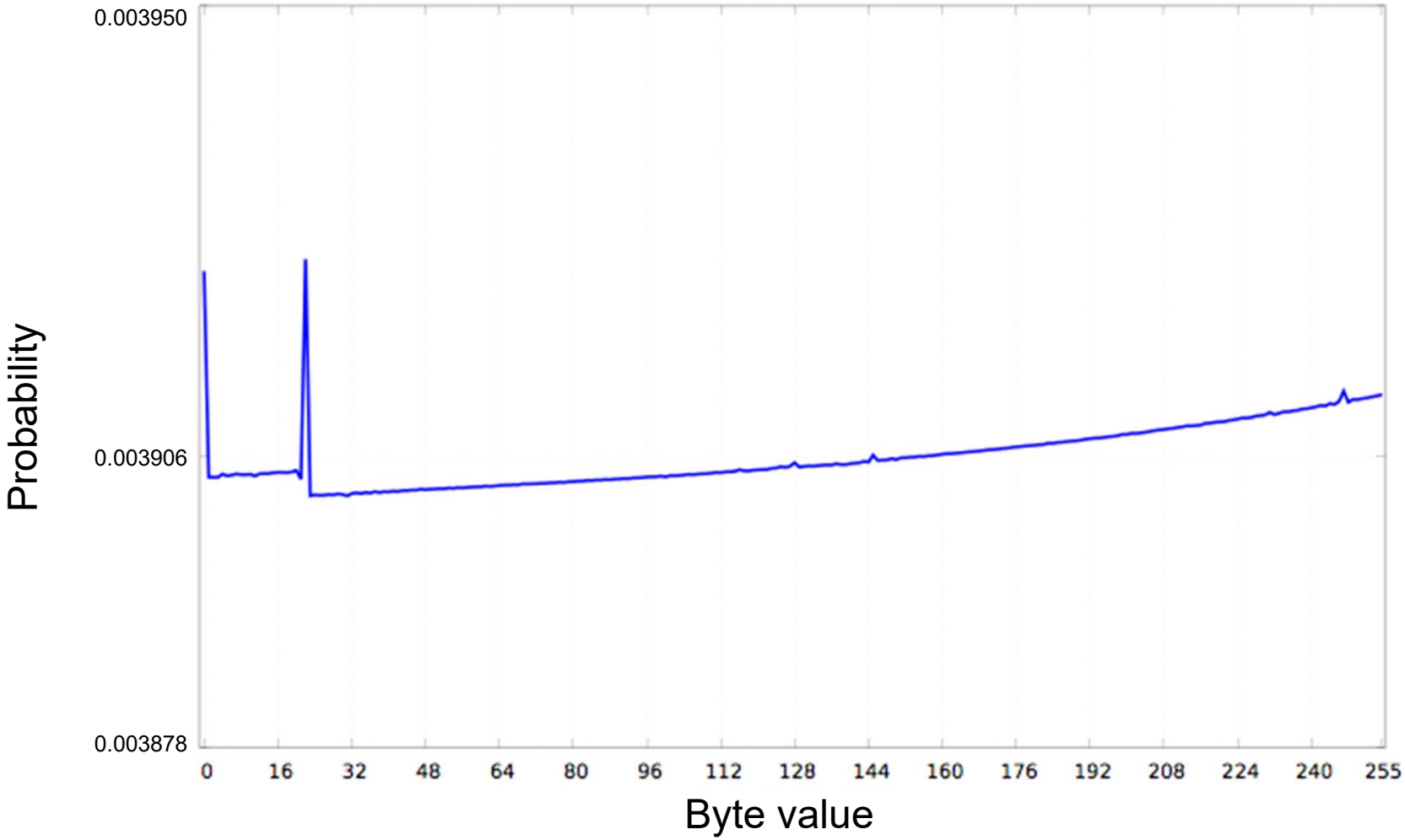
Keystream Distribution at Position 20



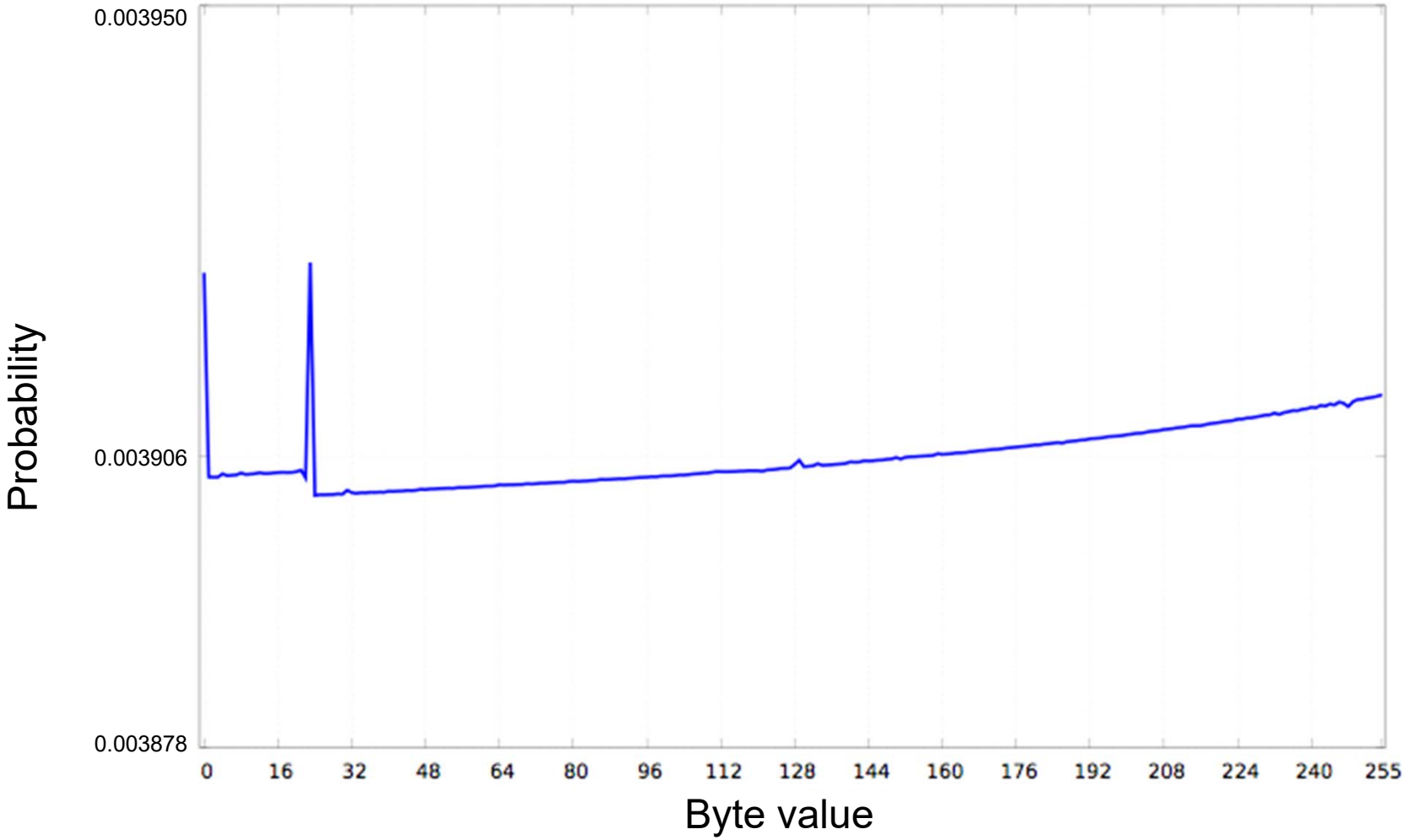
Keystream Distribution at Position 21



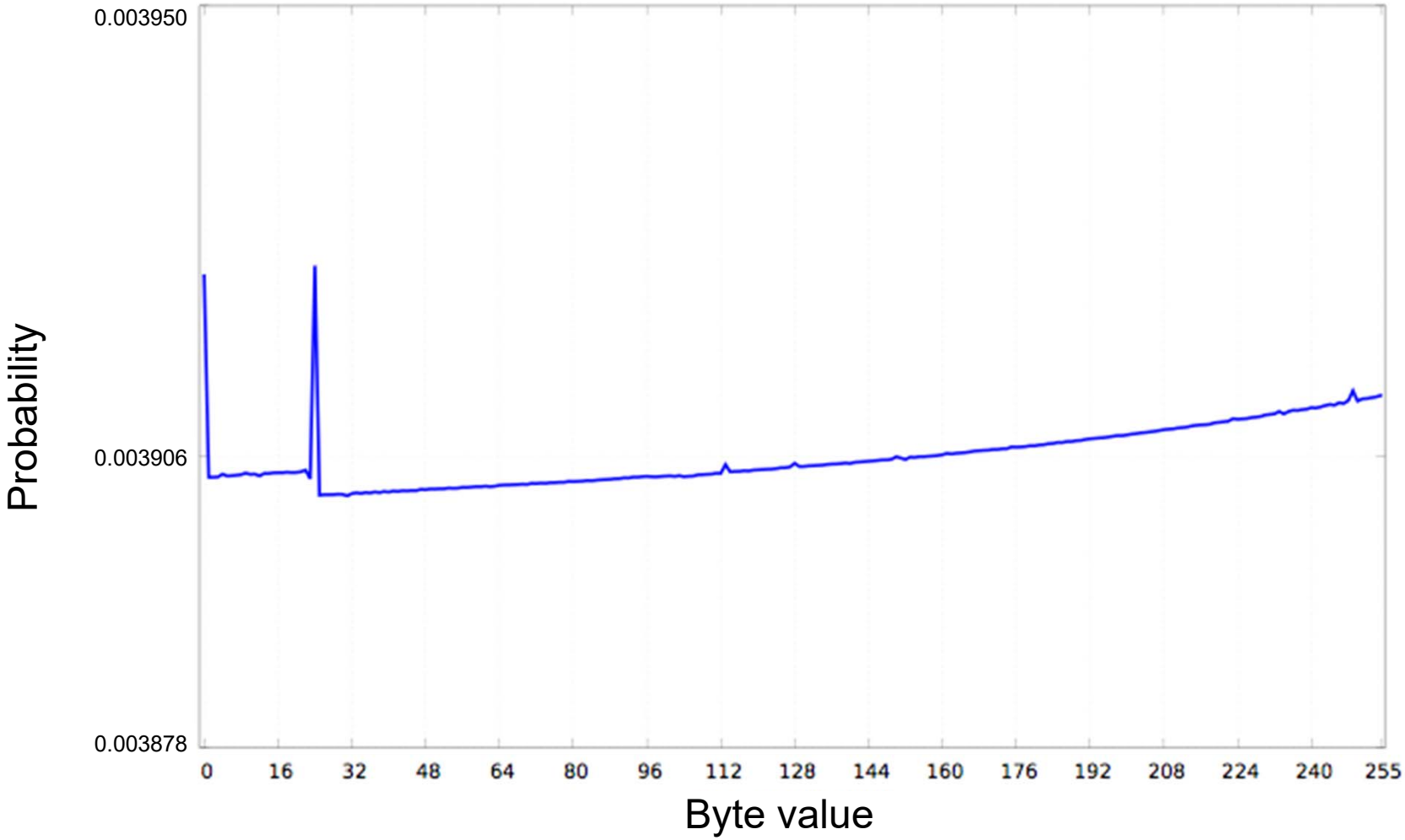
Keystream Distribution at Position 22



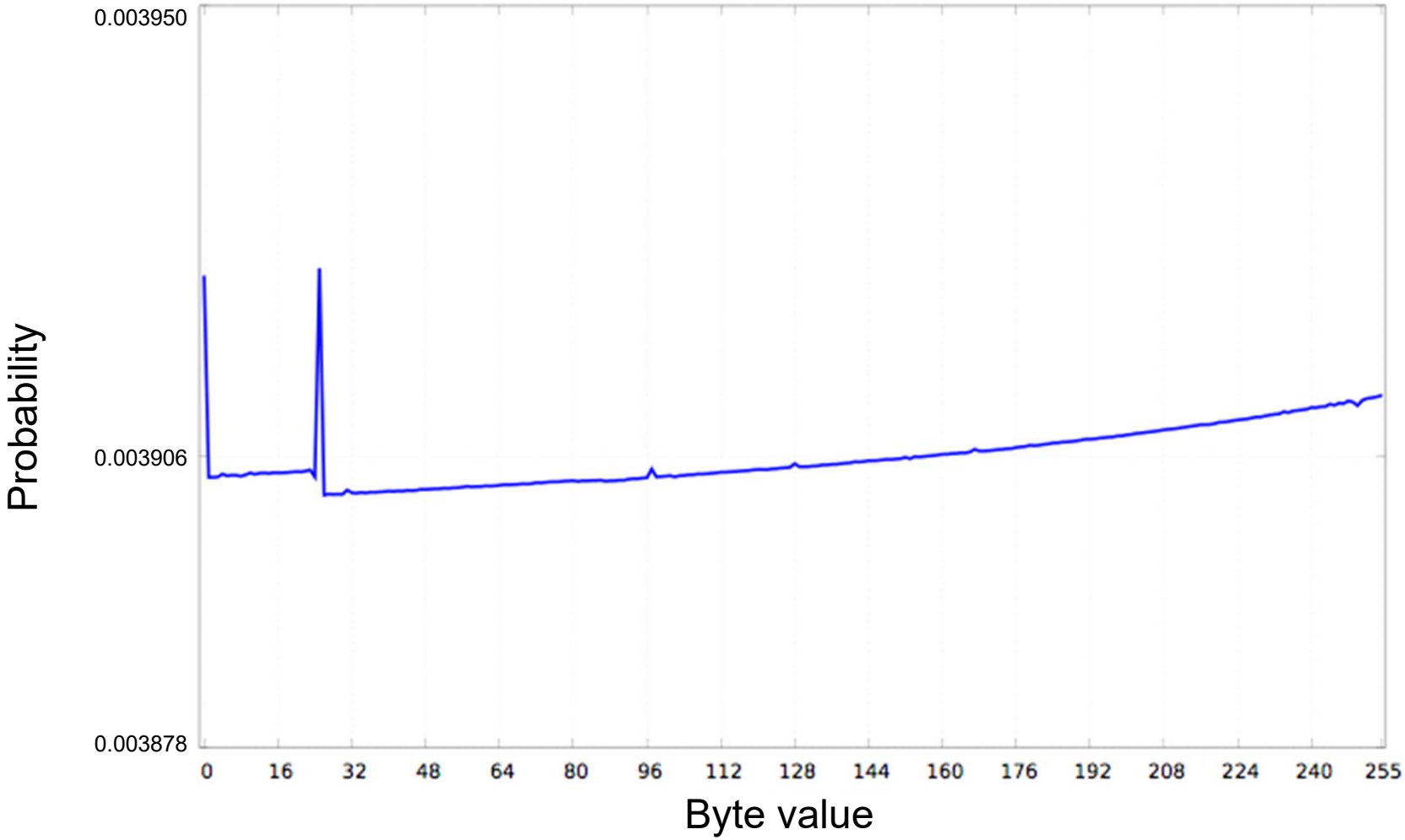
Keystream Distribution at Position 23



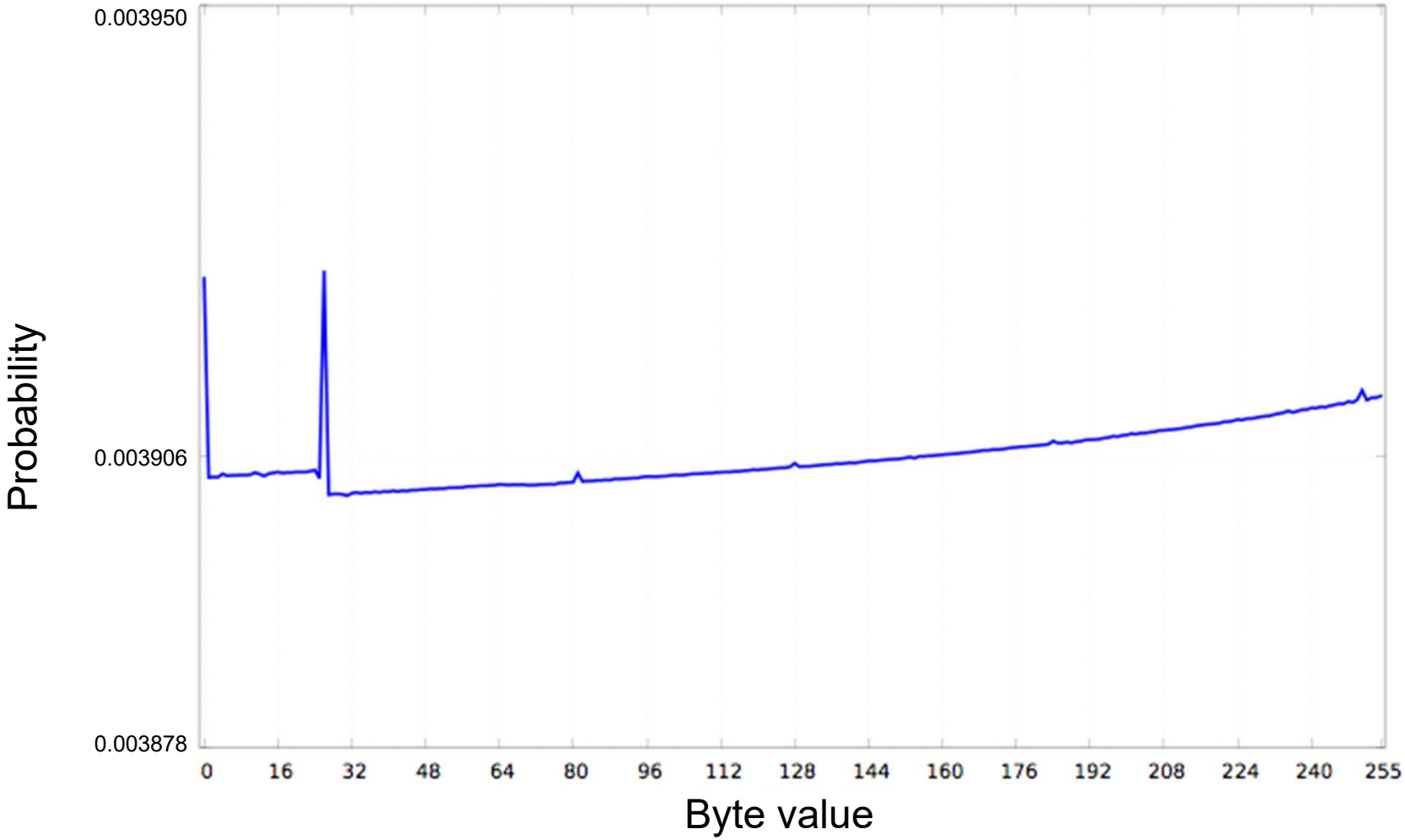
Keystream Distribution at Position 24



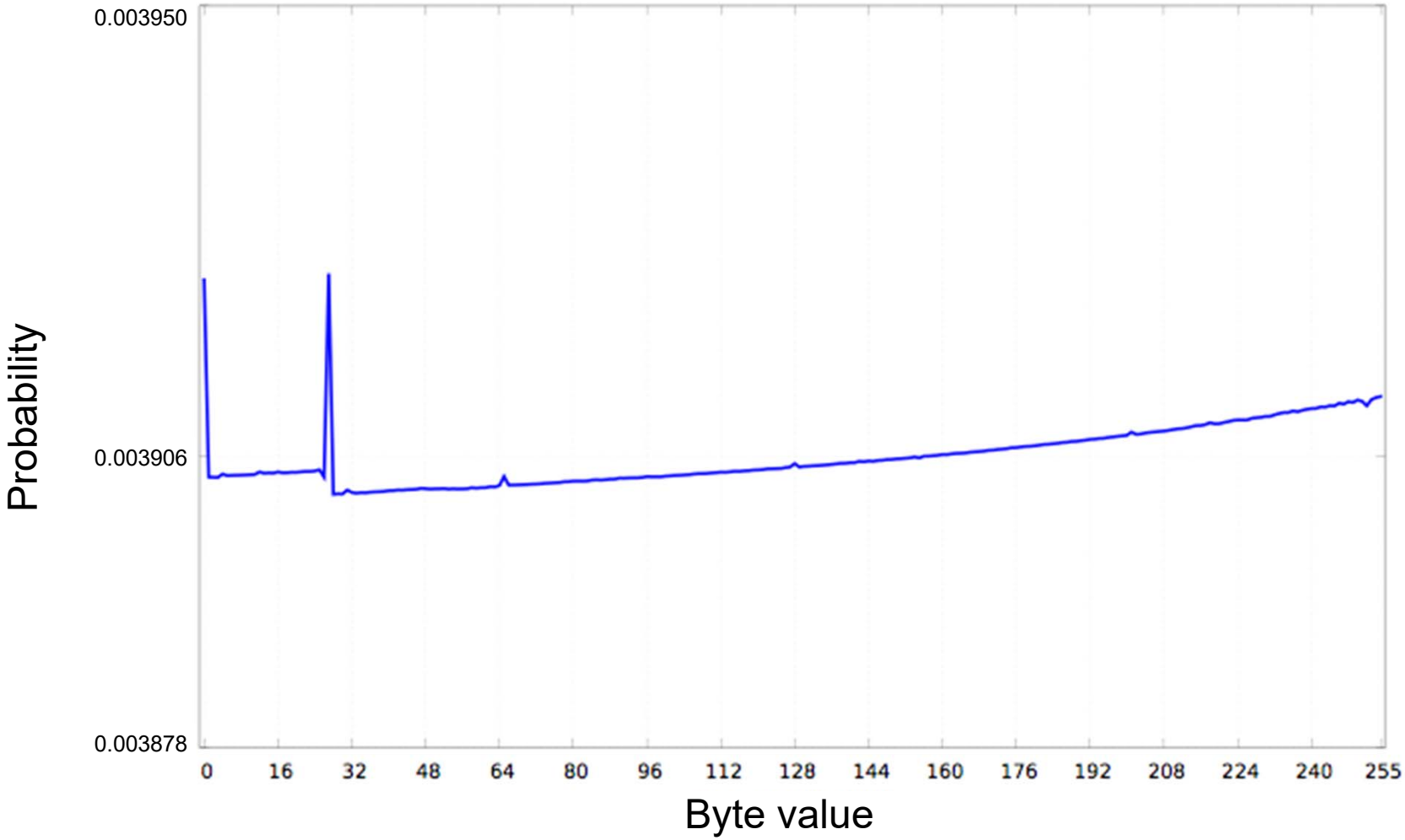
Keystream Distribution at Position 25



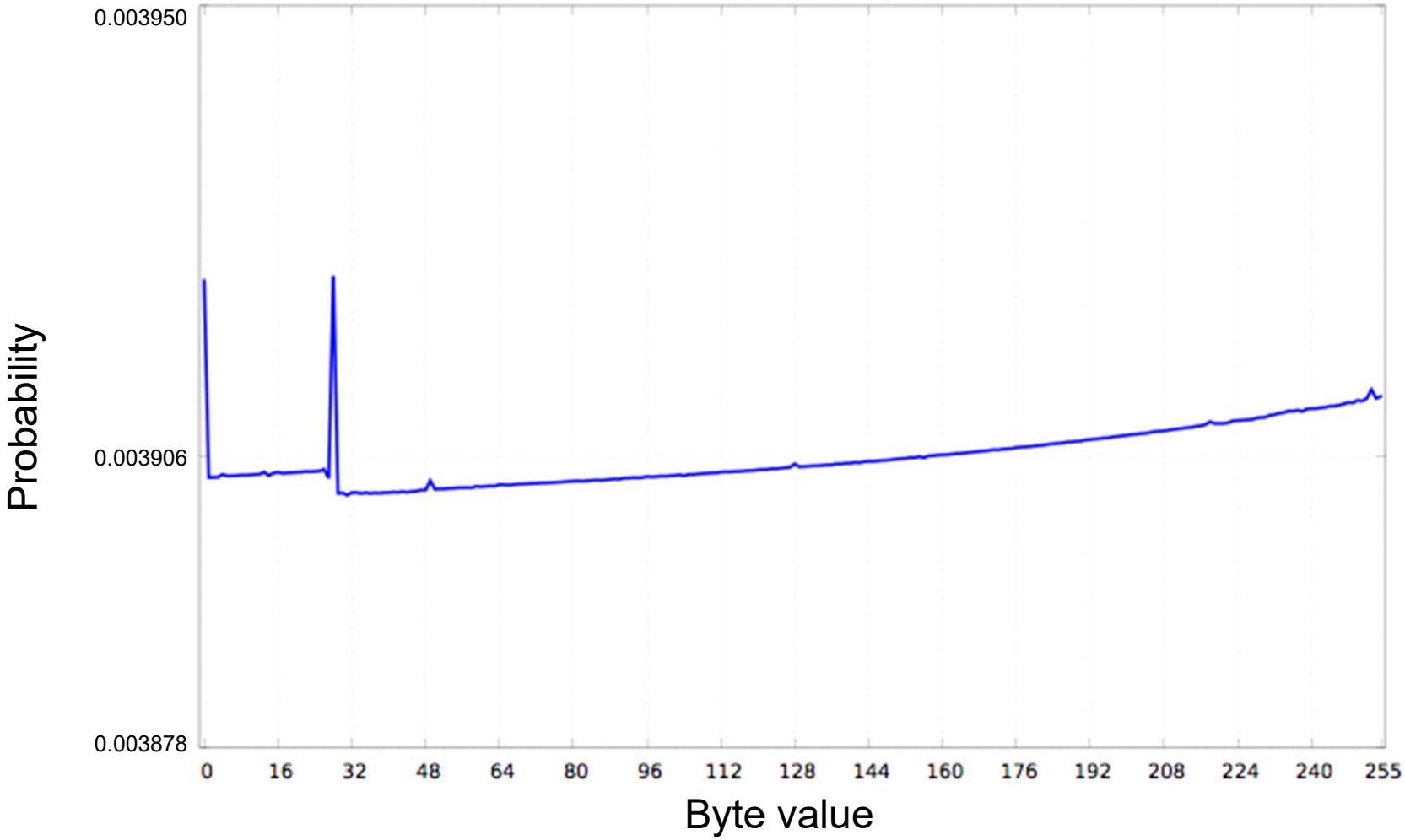
Keystream Distribution at Position 26



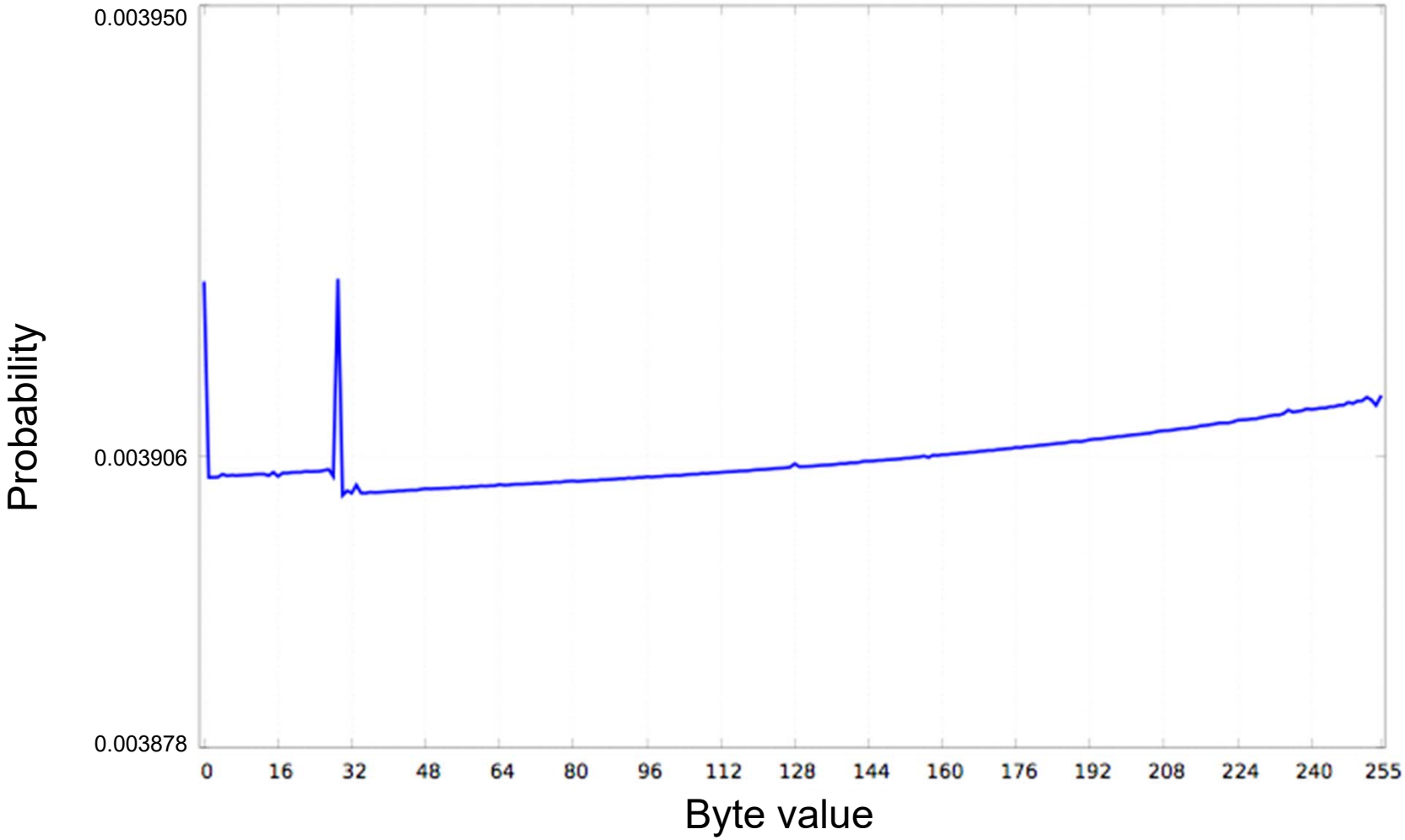
Keystream Distribution at Position 27



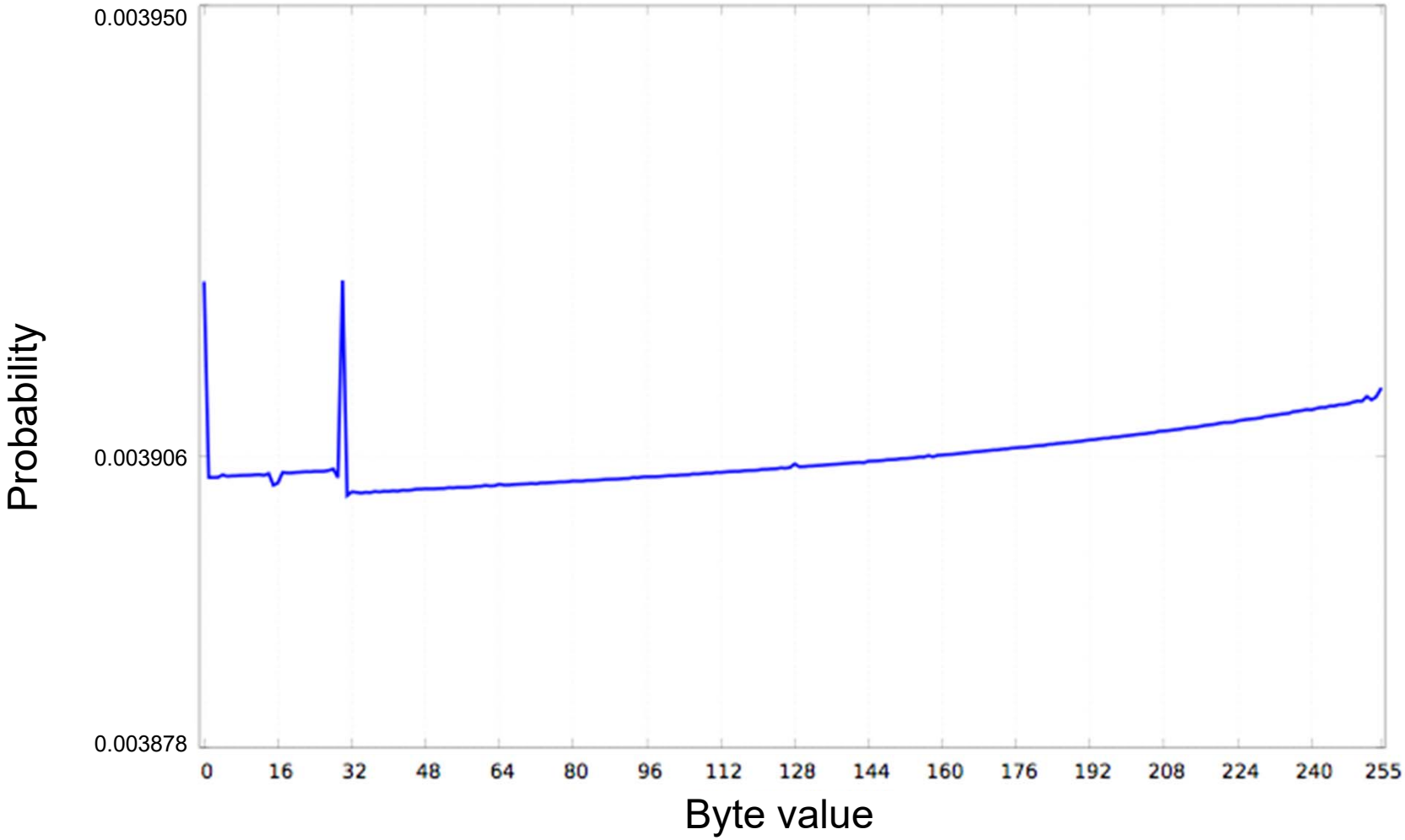
Keystream Distribution at Position 28



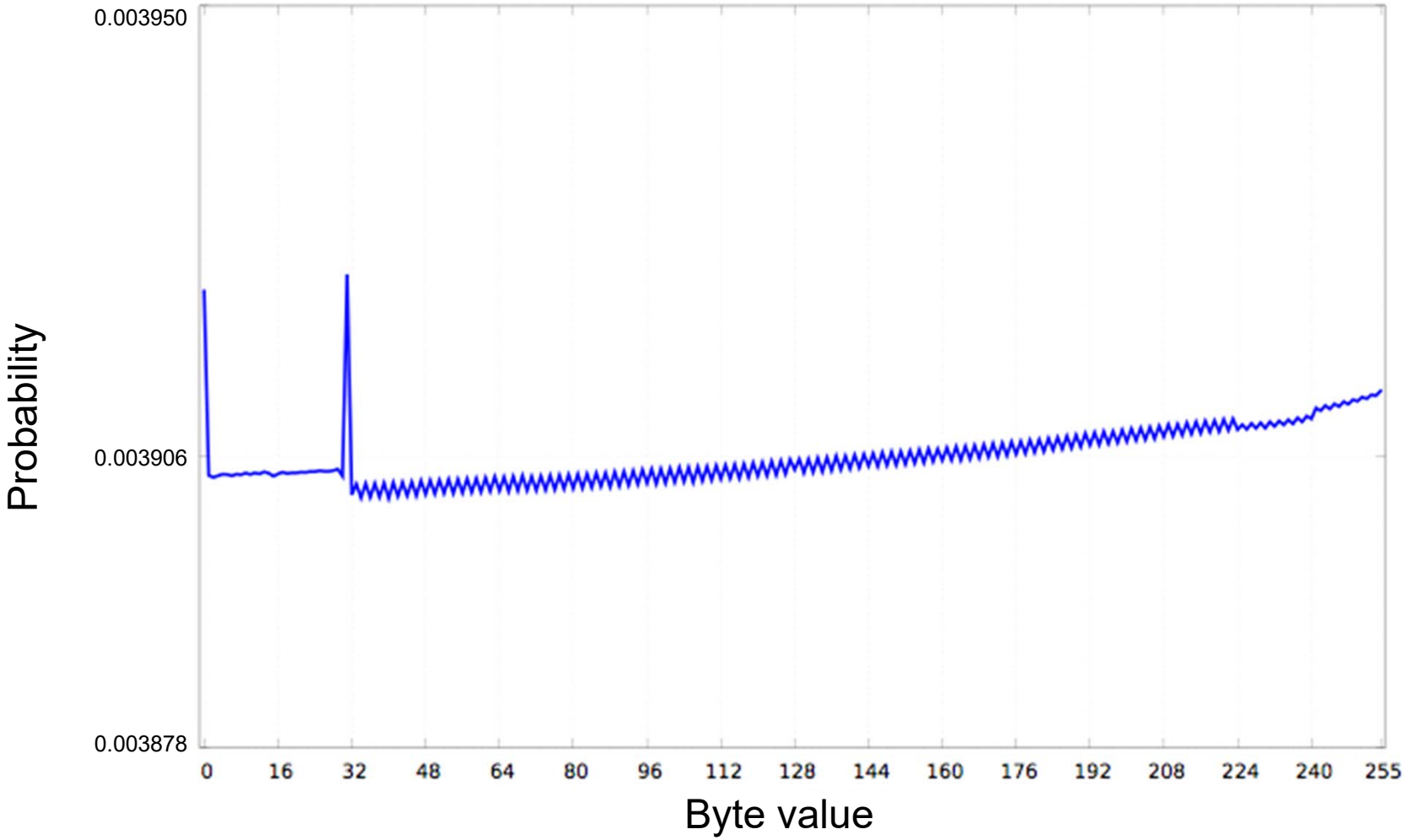
Keystream Distribution at Position 29



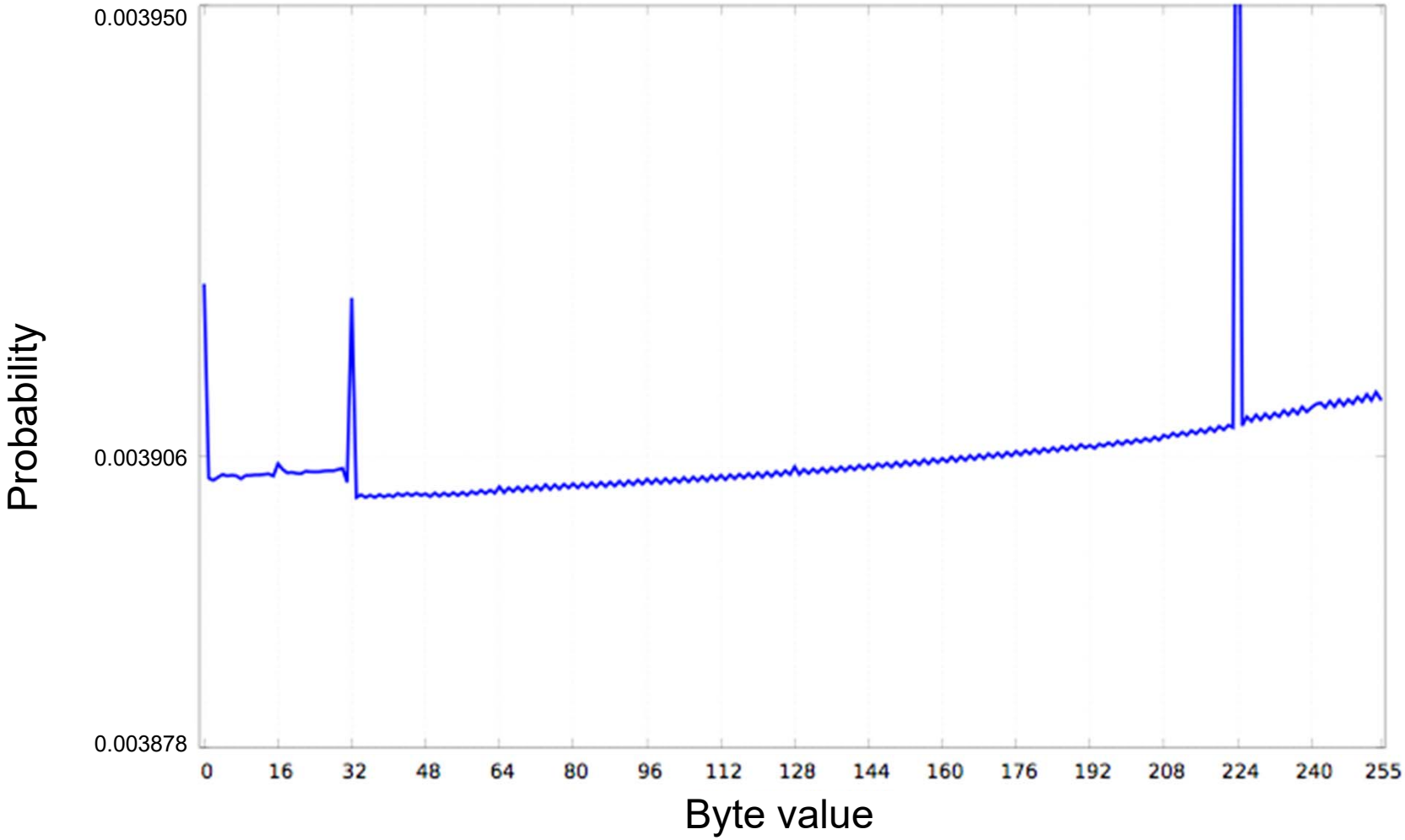
Keystream Distribution at Position 30



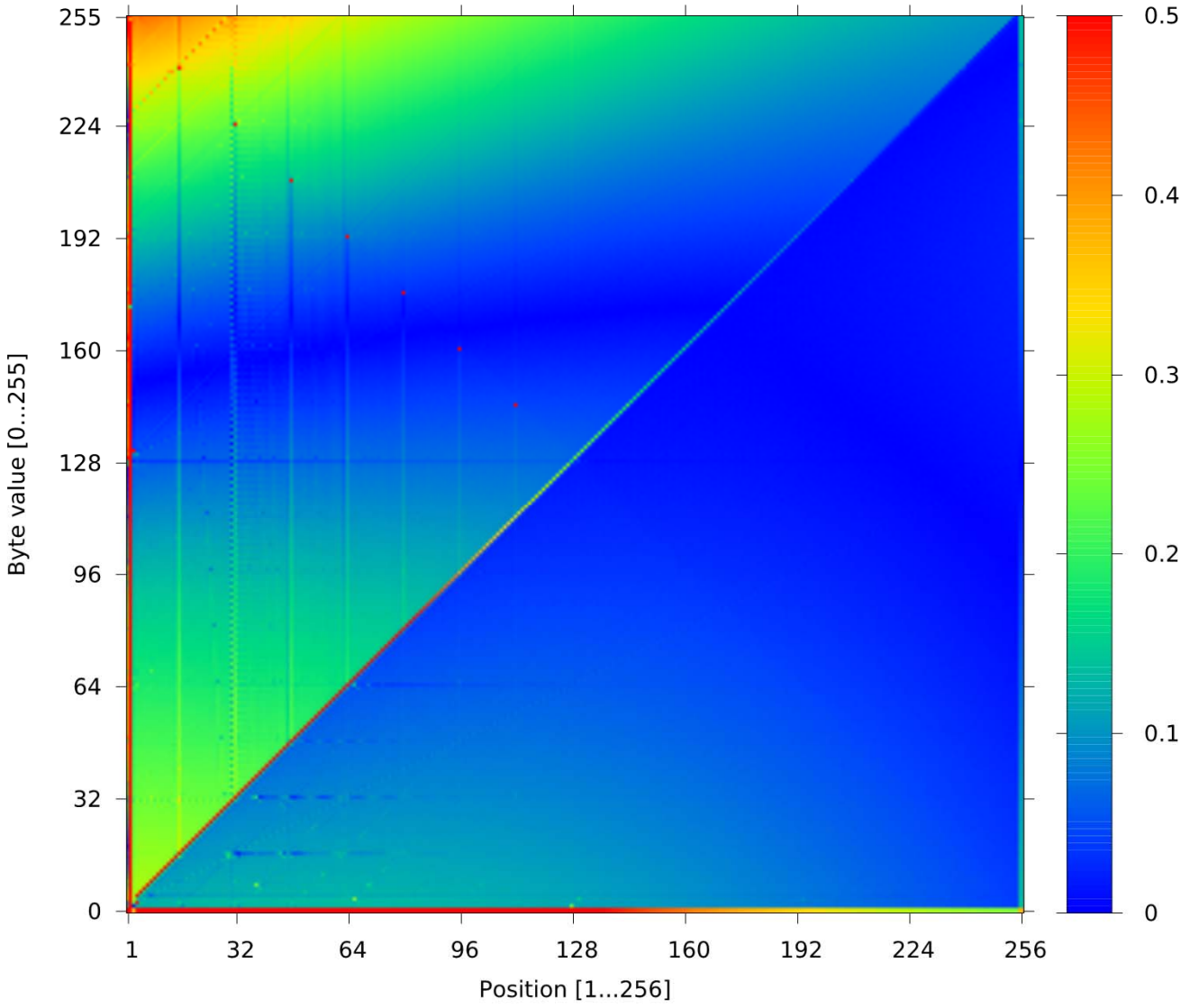
Keystream Distribution at Position 31



Keystream Distribution at Position 32



All the Biases



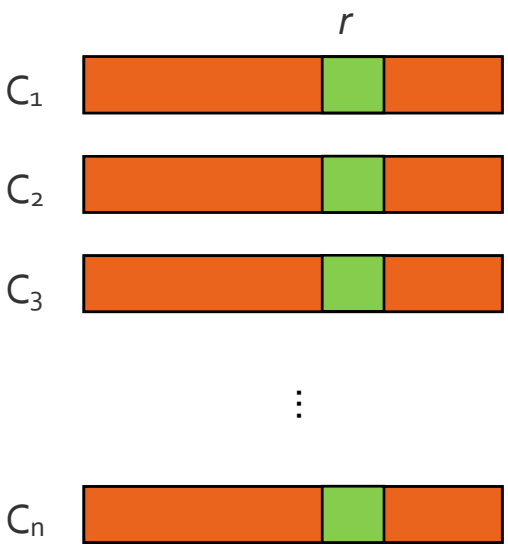
Plaintext Recovery for TLS-RC4

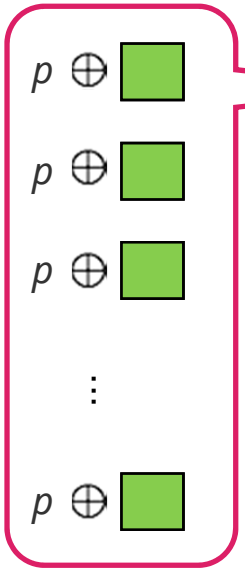
- Pretty picture, but where's the plaintext?
- Using the biased keystream byte distributions, we can construct a plaintext recovery attack against TLS.
- The attack requires the same plaintext to be encrypted under many different keys.
  - Use Javascript in browser as mechanism, cookies as target.
  - Reusing the BEAST mechanism once more.

Plaintext Recovery Using Keystream Biases

Encryptions of fixed plaintext under different keys

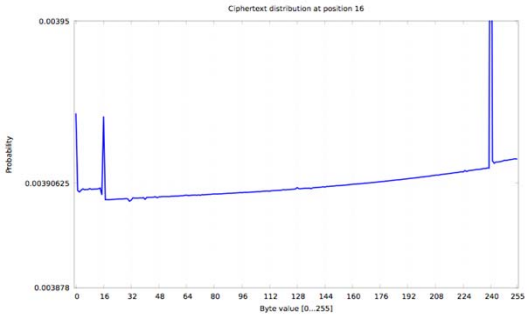
Plaintext candidate byte *p*





yields induced distribution on keystream byte Z_r

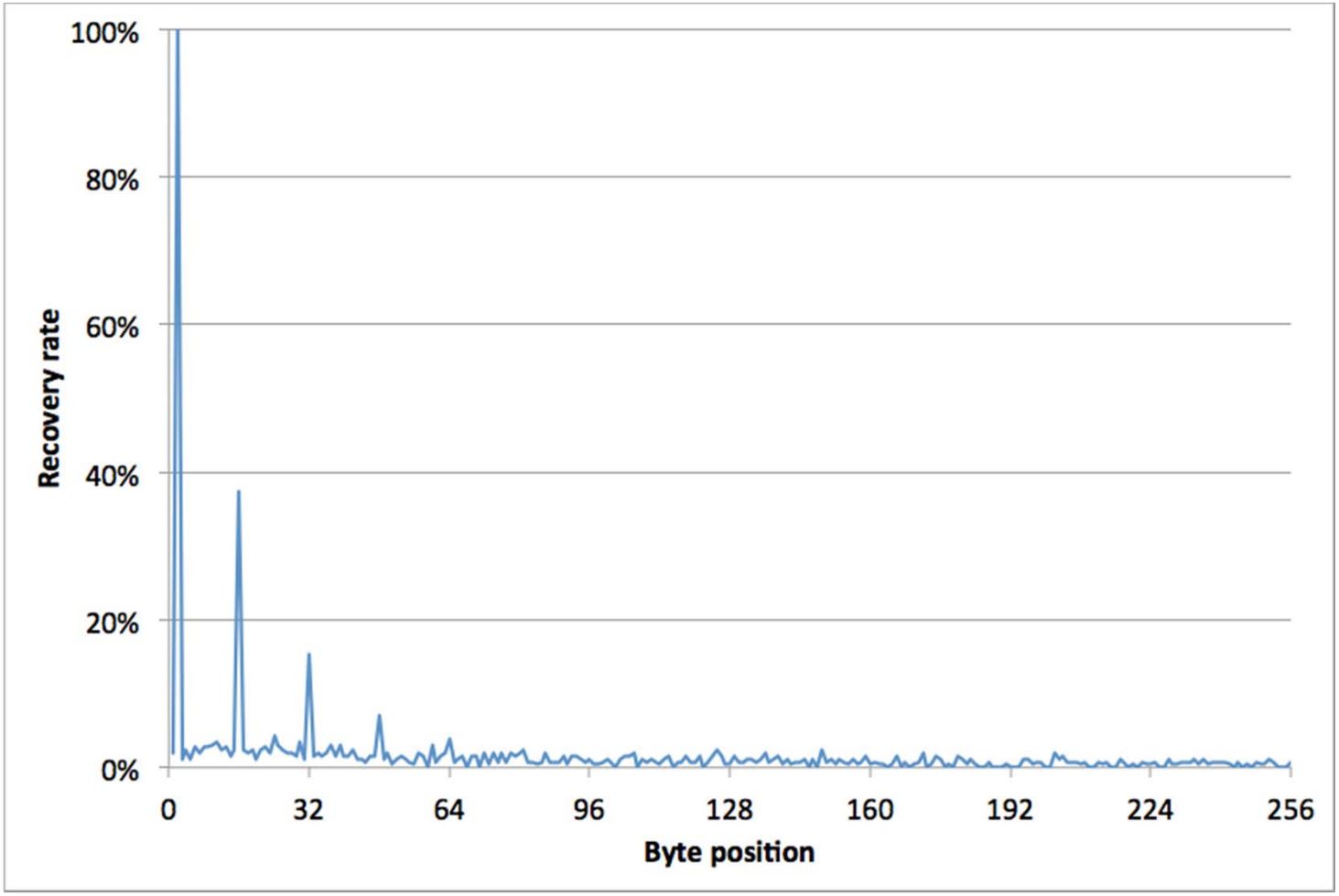
combine with known distribution



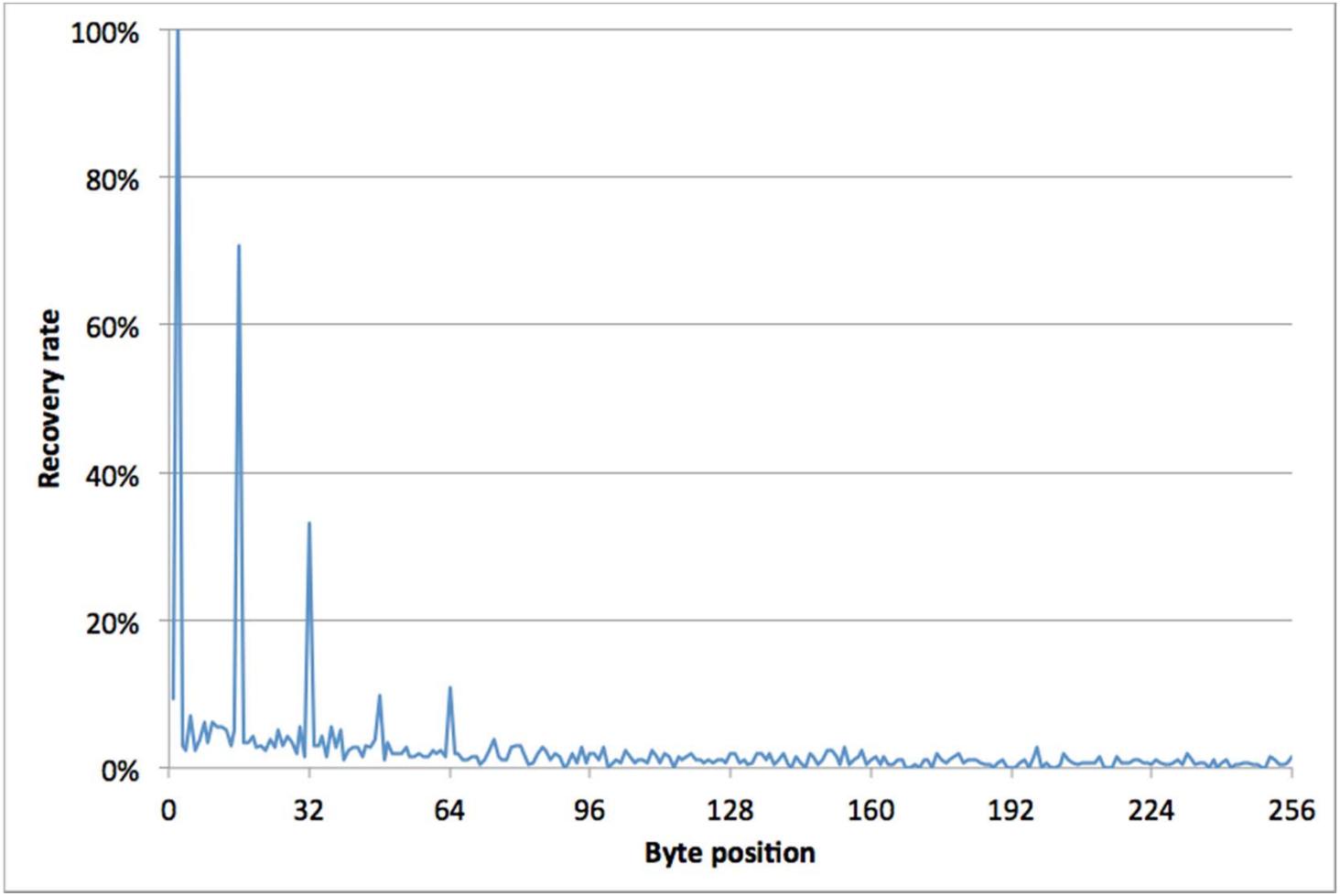
Compute likelihood of *p* being correct plaintext byte

Recovery algorithm:
Compute most likely plaintext byte

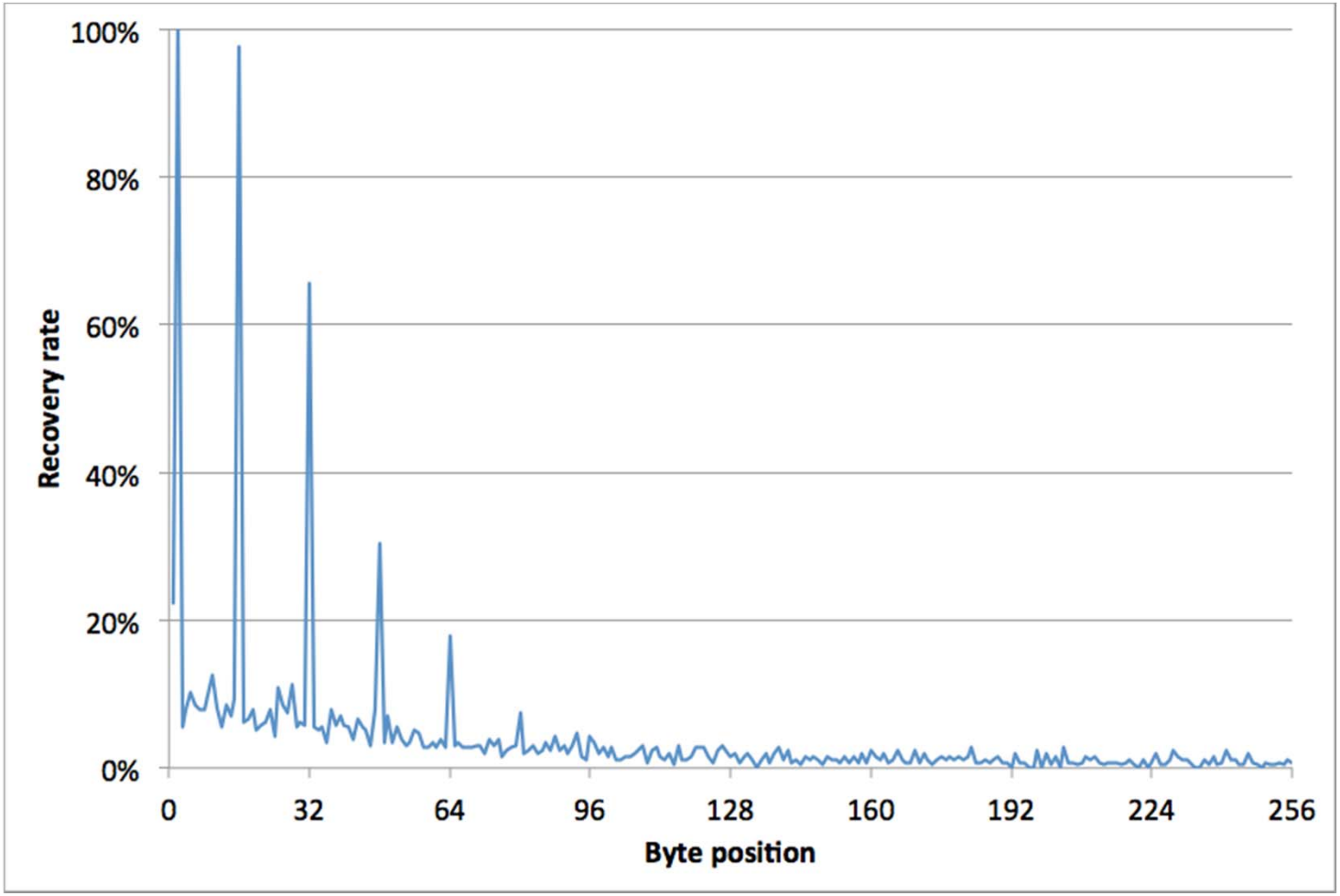
Success Probability 2^{20} Sessions



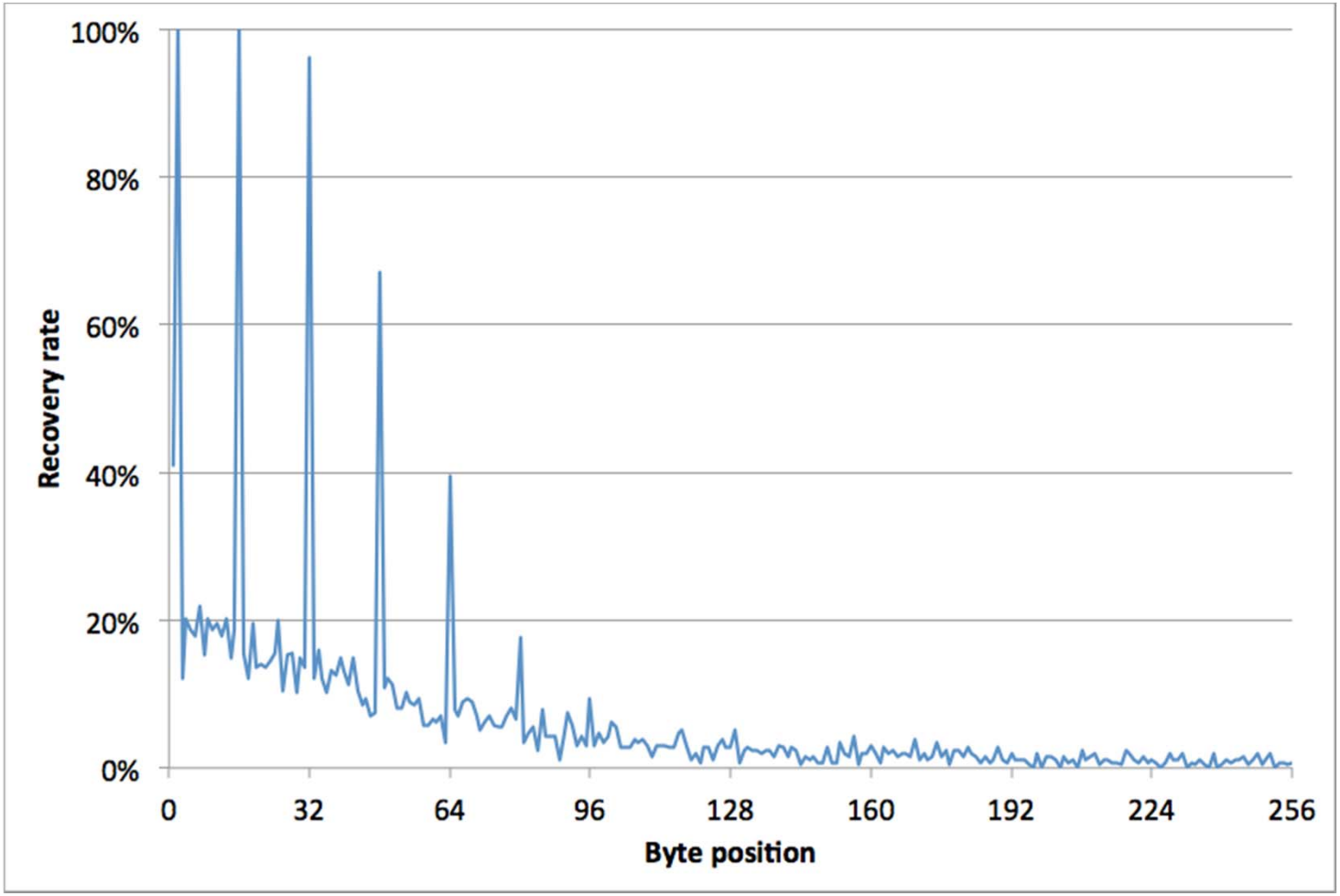
Success Probability 2^{21} Sessions



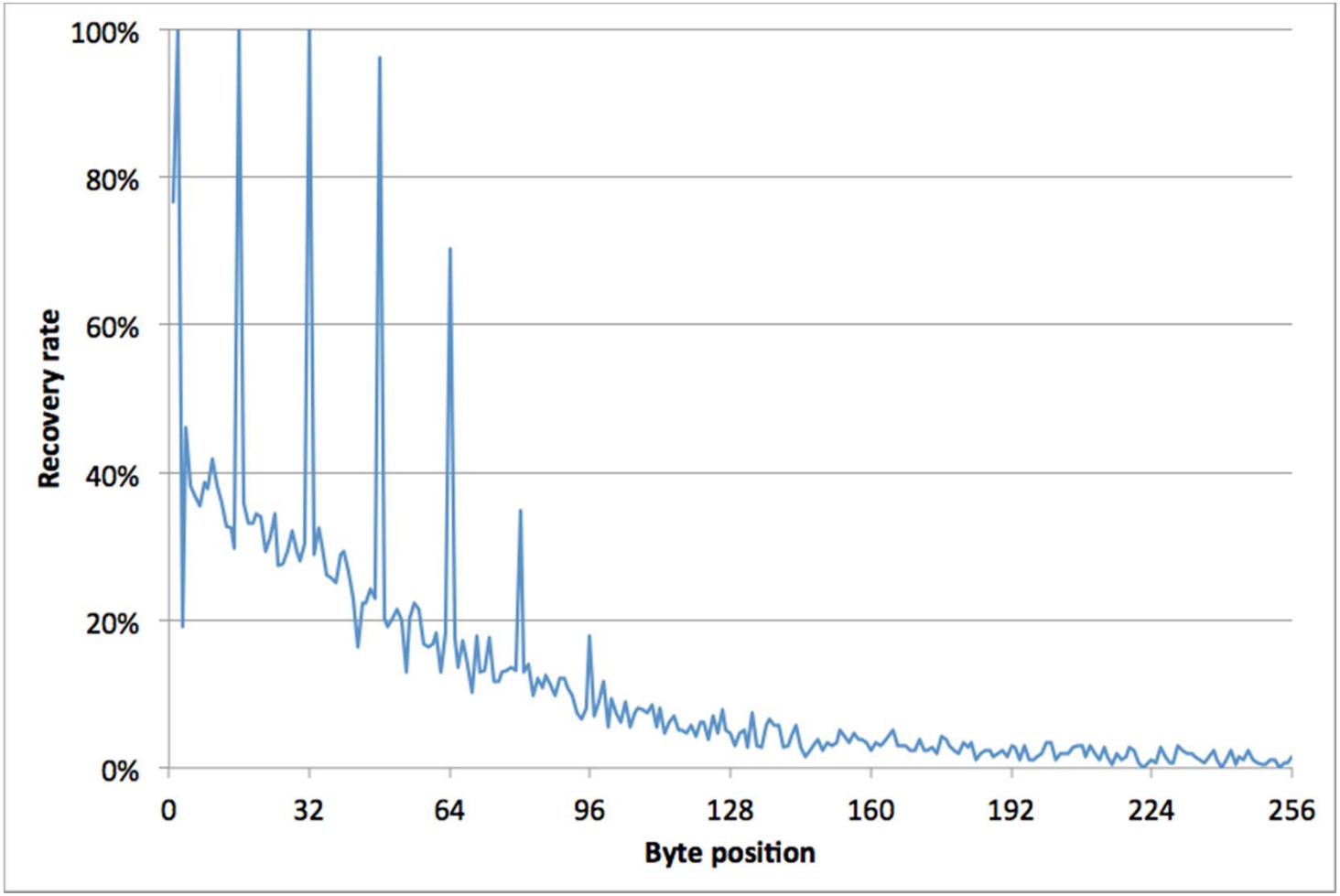
Success Probability 2^{22} Sessions



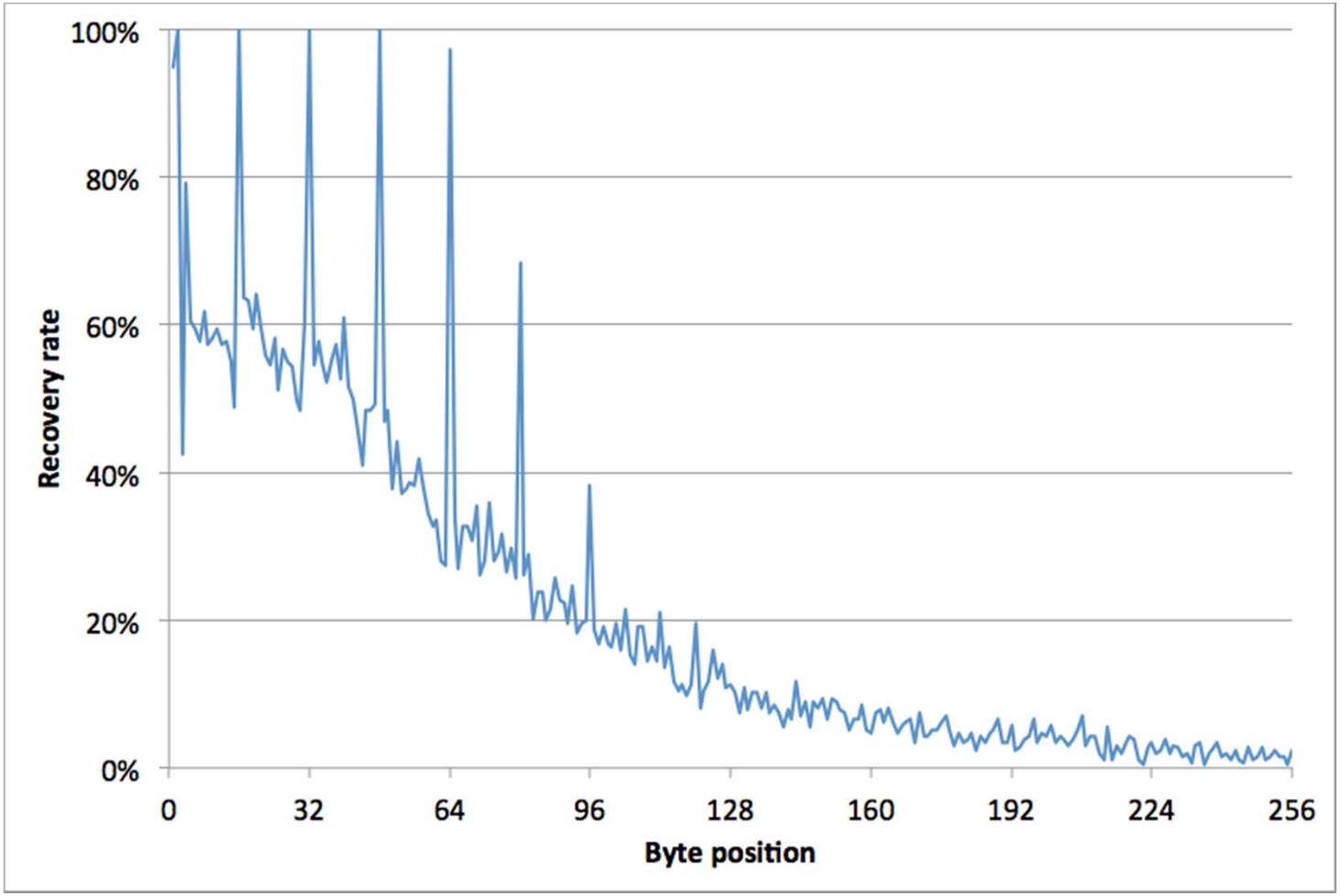
Success Probability 2^{23} Sessions



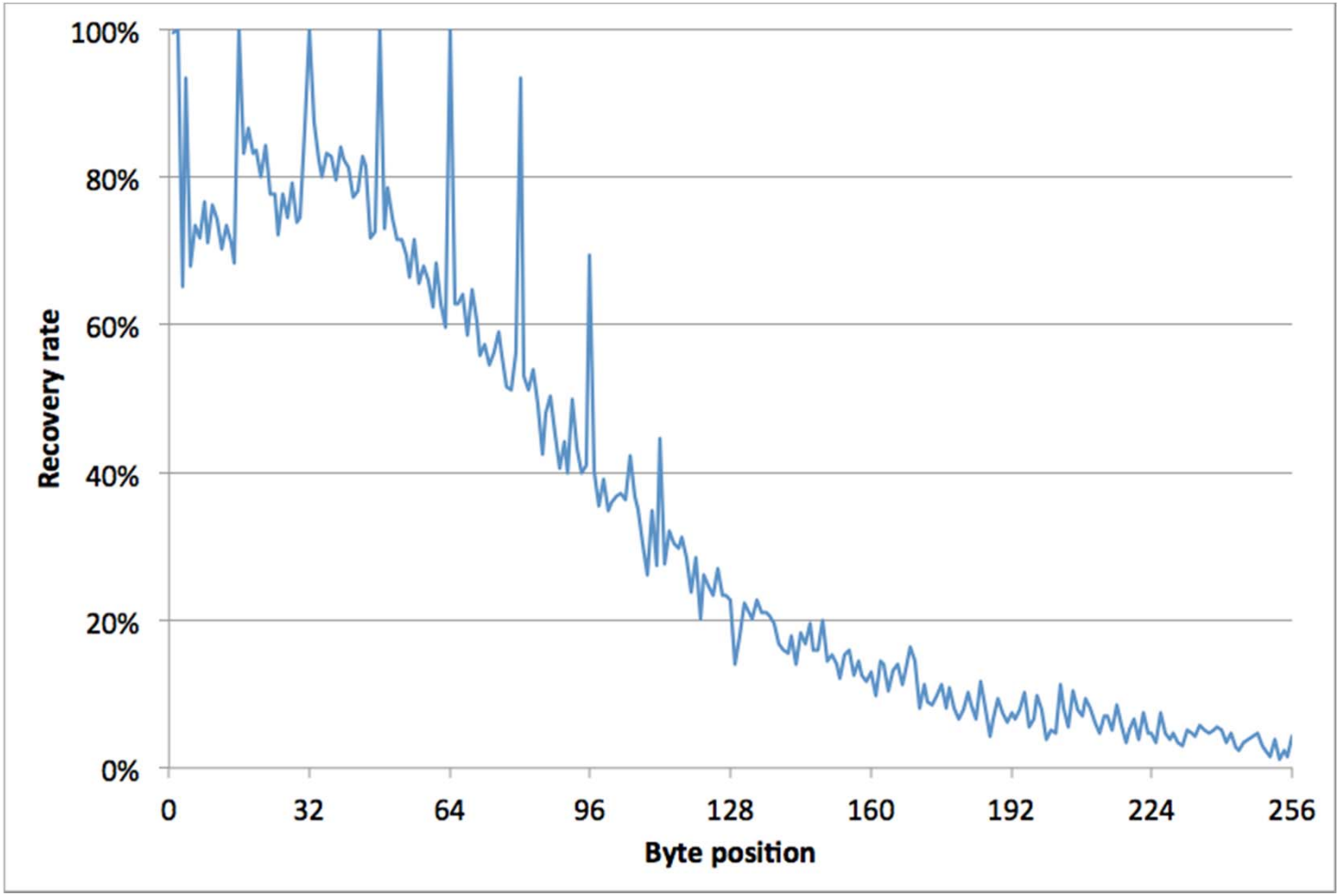
Success Probability 2^{24} Sessions



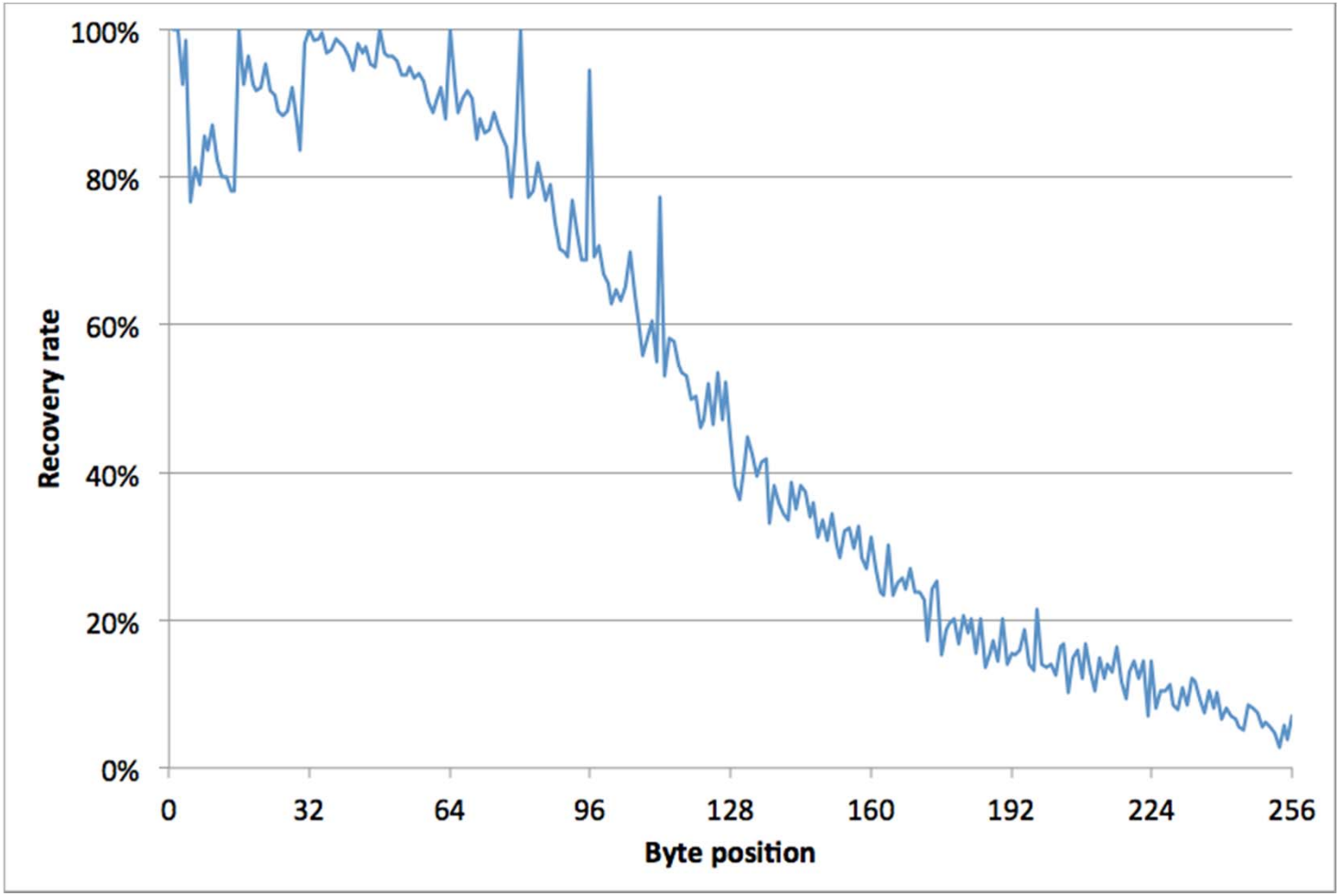
Success Probability 2^{25} Sessions



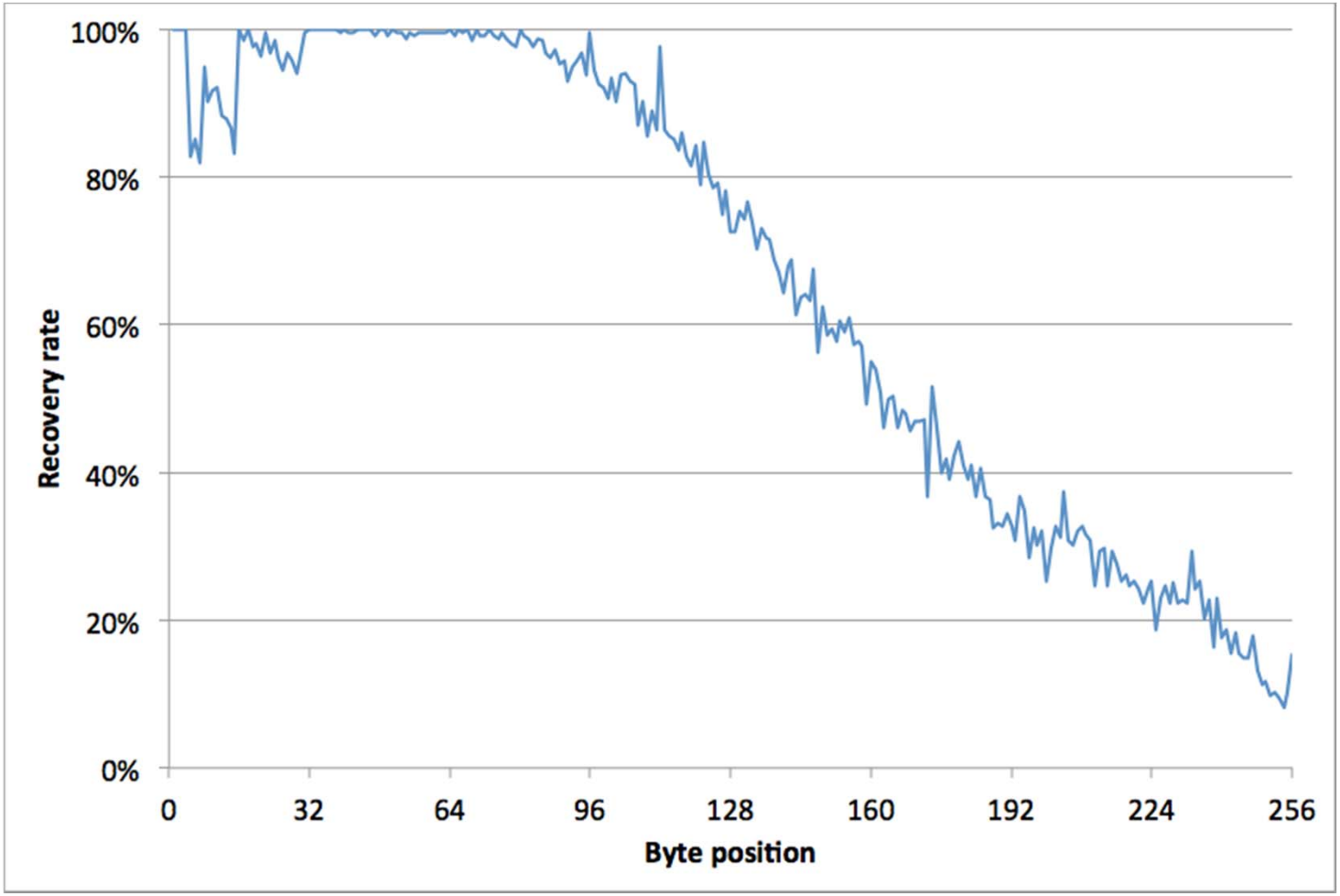
Success Probability 2^{26} Sessions



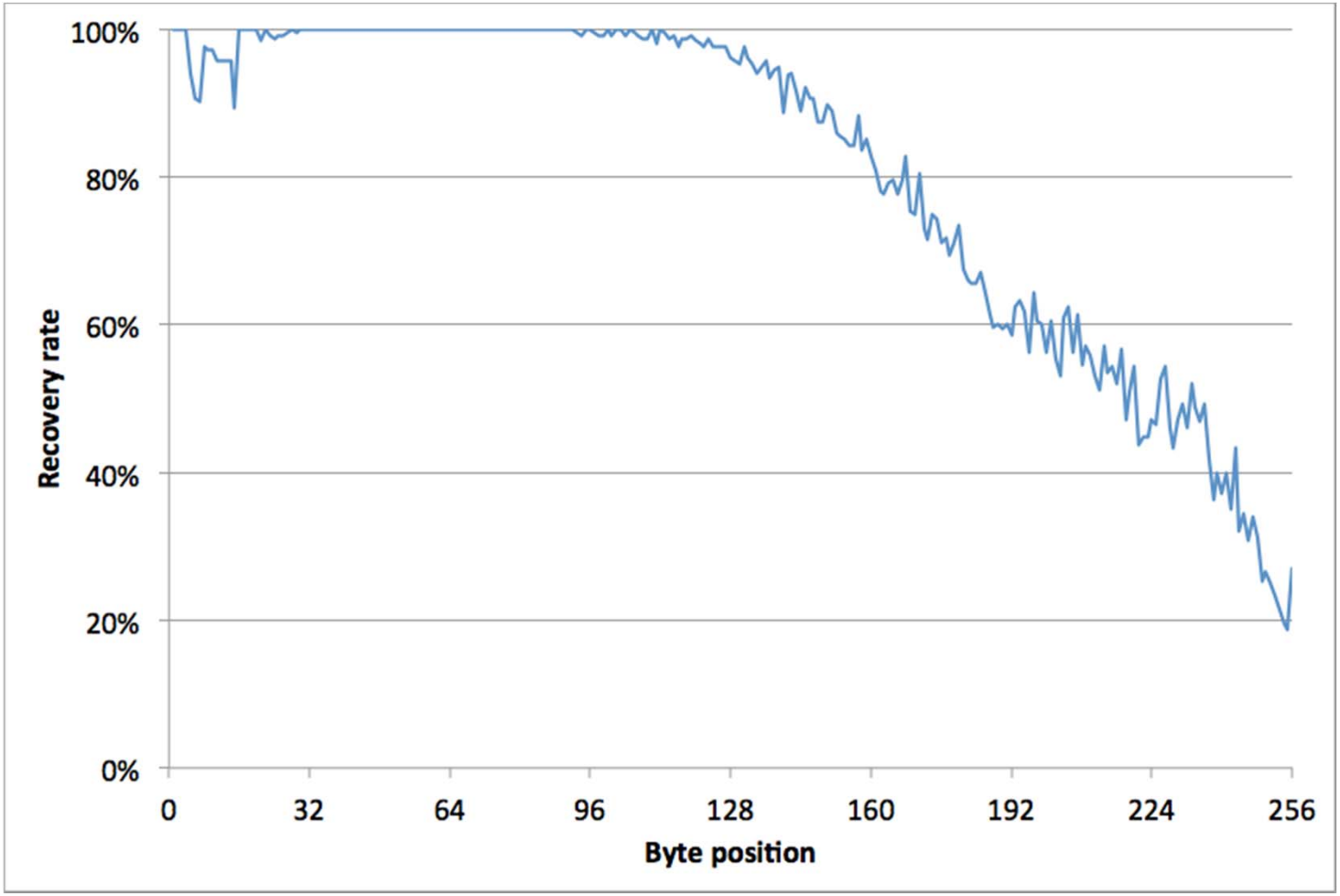
Success Probability 2^{27} Sessions



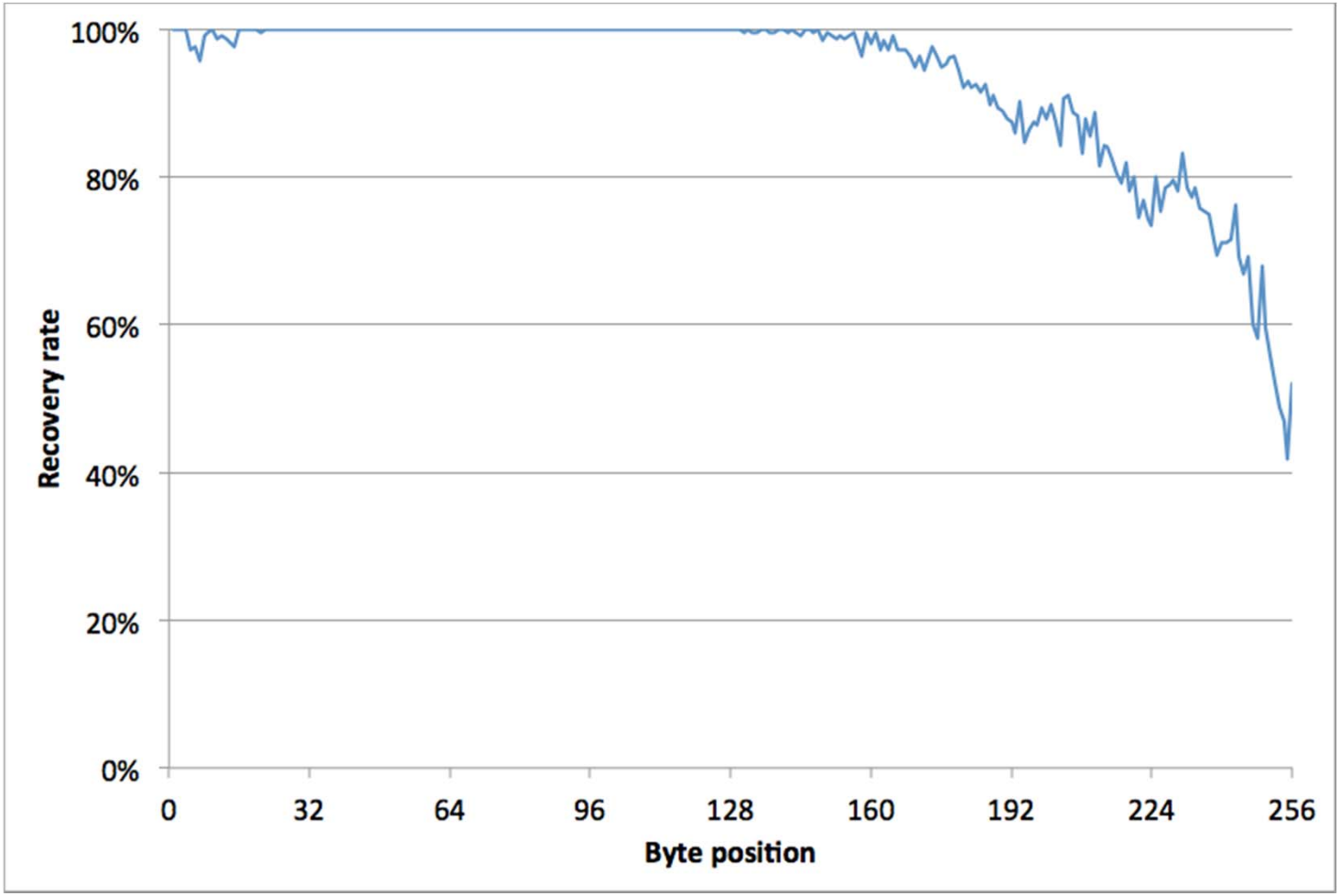
Success Probability 2^{28} Sessions



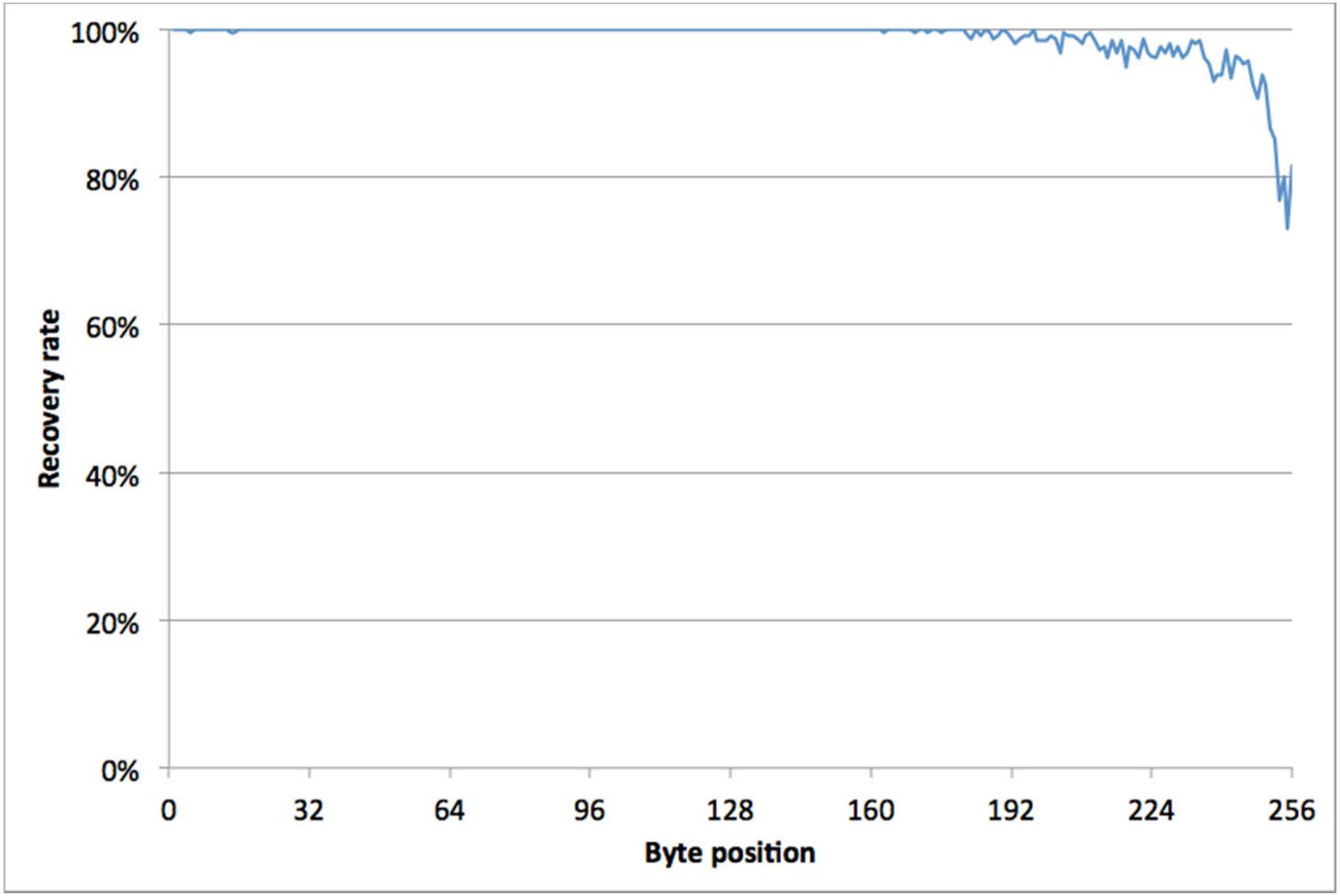
Success Probability 2^{29} Sessions



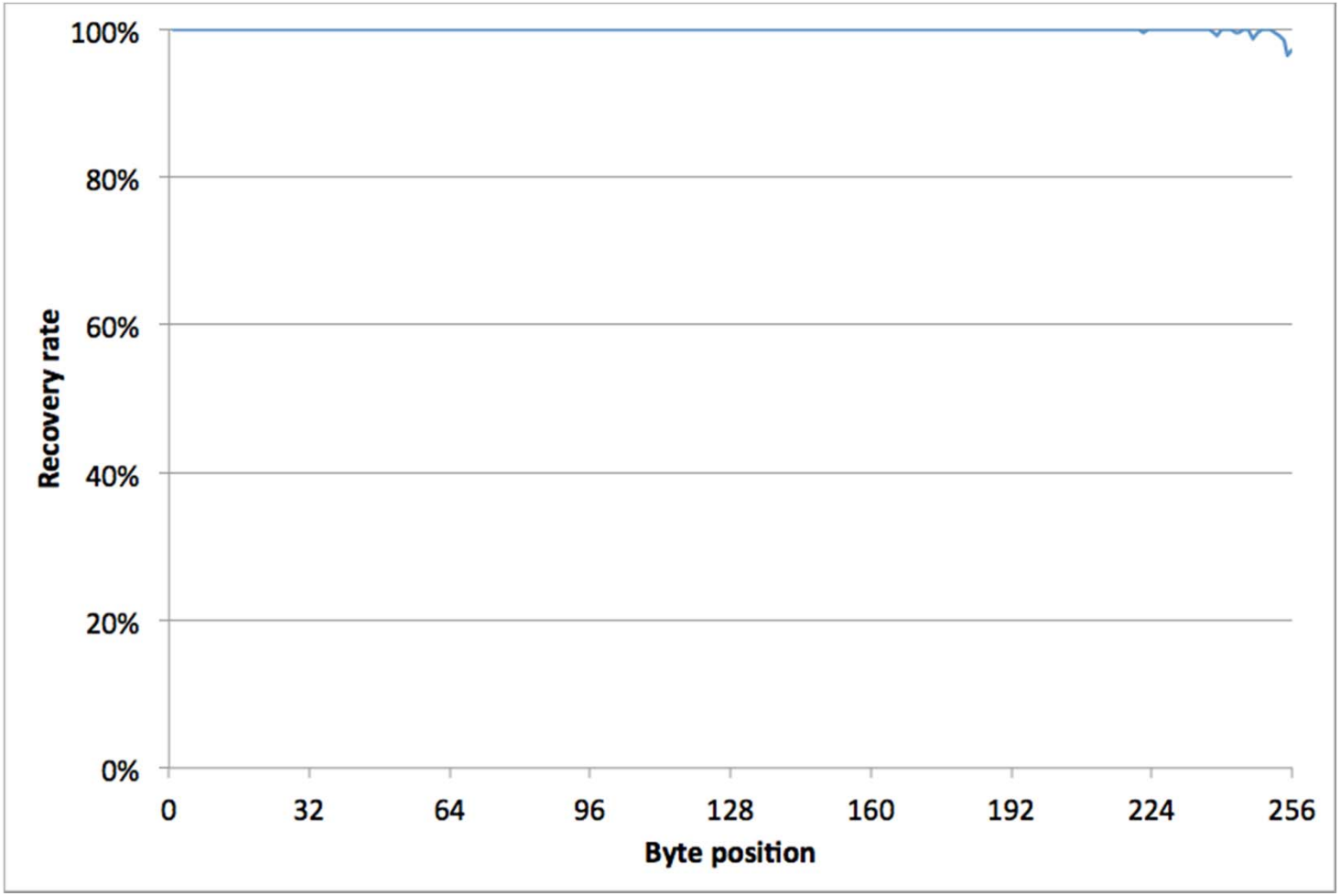
Success Probability 2^{30} Sessions



Success Probability 2^{31} Sessions



Success Probability 2^{32} Sessions



Media – 2013

ars technica

MAIN MENU

RISK ASSES

Two new a
authentica
Aging standard isn't h

by Dan Goodin - Mar 14, 2013

The Register®
Biting the hand that feeds IT

DATA CENTRE SOFTWARE NETWORKS SECURITY INFRASTRUCTURE BUSINESS HARDWARE SCIE

Security

HTTPS cookie crypto CRUMBLES AGAIN in hands

Keep calr

15 Mar 2013 at

Fresh cryptogra
giants to encry

The attack, dev
Illinois at Chica

Schneier on Security

Blog Newsletter Books Essays News Events Crypto About Me

← Unwitting Drug Smugglers The Dangers of Surveillance →

New RC4 Attack

This is a really clever attack on the RC4 encryption algorithm as used in TLS.

RC4: Subsequent Developments

[ABPPS13]: use Fluhrer-McGrew biases on consecutive keystream bytes to avoid many connections and target bytes later in the keystream, 2^{34} encryptions, 2000 hours.

RFC 7465 "Prohibiting RC4 cipher suites", Jan. 2015:

> `This document requires that TLS clients and servers never negotiate the use of RC4 cipher suites.`

[GPV15]: refinement of [ABPPS13] attacks focussed on password recovery from early in the keystream: 60% success rate with 2^{26} encryptions, 350 hours.

[VP15]: use of Mantin biases to recover cookies: 94% success rate with about 2^{30} encryptions, 75 hours.

September 1st, 2015: Microsoft, Google, Mozilla all announce that Rc4 will be fully disabled in their browsers in early 2016. (This eventually happened!).

Today: Almost no RC4 traffic from browsers, some from legacy equipment.

Sweet 32

Sweet 32 (Bhargavan-Leurent, CCS'16)

- Sweet 32 applies to SSL/TLS cipher suites using 64-bit block ciphers (DES, triple DES).
- CBC mode ciphertext block collisions occur at the birthday bound: after 2^{32} blocks have been encrypted under a fixed key for a 64-bit block.
- Ciphertext block collisions leak some plaintext information:

$$C_i = C_j \text{ implies } C_{i-1} \oplus P_i = C_{j-1} \oplus P_j.$$

- So, if P_i is known, then P_j can be recovered.
- Leads to an HTTP cookie recovery attack using roughly same attack resources as are needed in RC4 attacks.
- All data must be in a single TLS connection (single key).
- More details at https://sweet32.info/

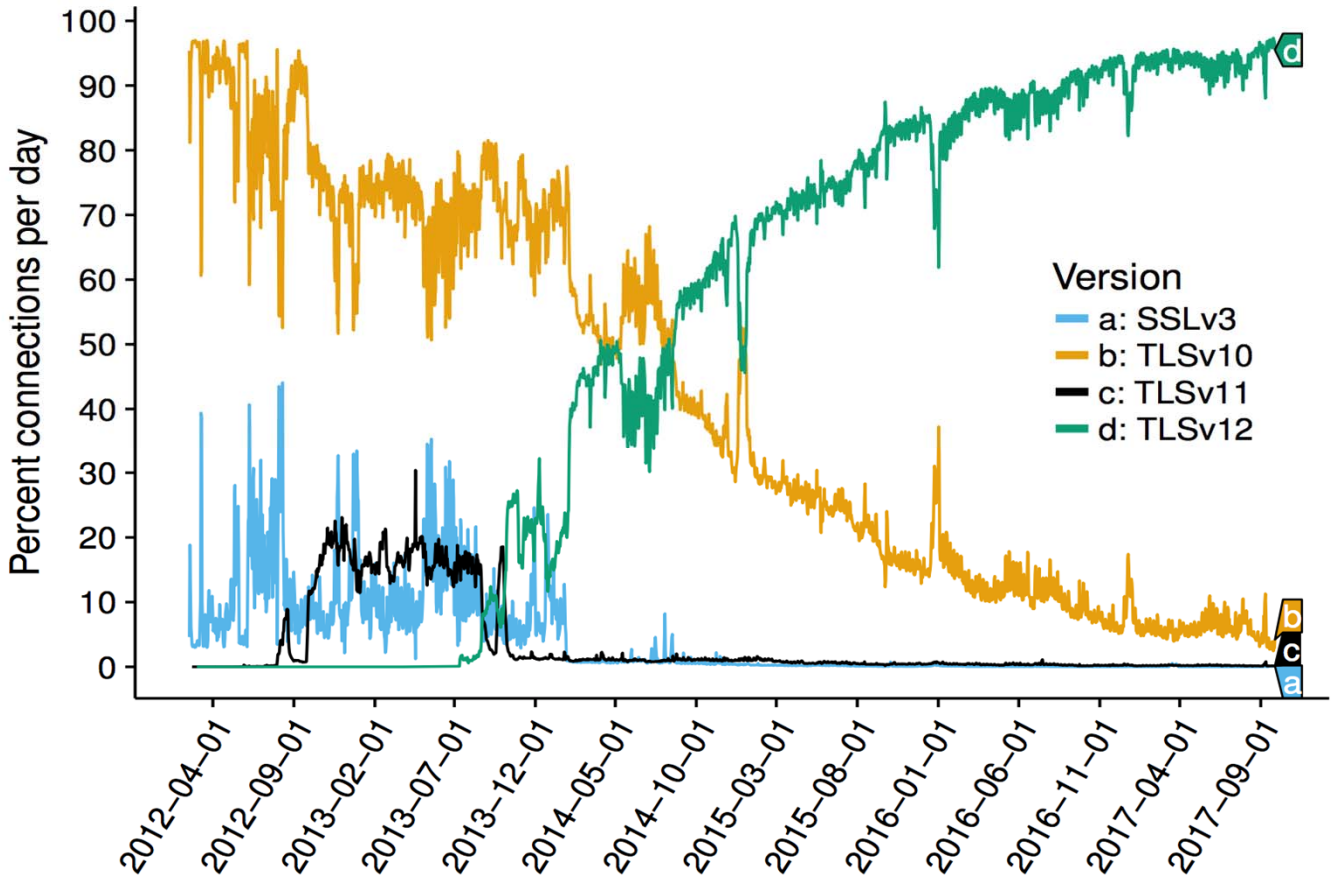
Summary of TLS Record Protocol Attacks

- **BEAST** (2011)– exploits TLS 1.0's use of predictable IVs.
- **CRIME (and BREACH, TIME)** (2012) – exploits TLS and application support for compression.
- **Padding oracle attack** (2002, 2003) – exploits TLS 1.0's use of distinguishable error messages for padding and MAC failures.
- **Lucky 13** (2013) – padding oracle attacks are still possible, even after application of recommended countermeasures; MEE with CBC is hard to implement without side channels.
- **POODLE** (2014) – kind of padding oracle attack for SSL3.0, based on error messages rather than timing; SSL3.0-killer.
- **RC4 attacks** (2013-2015) – RC4 is not such a good stream cipher after all.
- **Sweet 32** – small block-size block ciphers in CBC mode start to leak plaintext after $2^{32} – 2^{35}$ blocks.

Current Status

- CBC-mode ciphersuites can be patched against BEAST and Lucky 13, but their reputation has been damaged by the long series of attacks.
  - Relative performance also an issue (AES-CBC + HMAC quite slow).
- RC4 is pretty much dead.
- AES-GCM and AES-CCM are only available for TLS 1.2, driving the switchover to TLS 1.2.
- ChaCha20-Poly1305 for environments where AES-NI not available.

SSL/TLS Versions in Use on the Internet



Source: Amann et al. "Mission Accomplished? HTTPS Security after DigiNotar", IMC 2017.

Attacks on AES-GCM

- AES-GCM is tricky to implement securely.
  - One issue is avoiding leakage of hash key via side-channel attack.
  - Also needs side-channel resistant implementation of AES.
- AES-GCM absolutely must avoid repeating nonces, otherwise there are well-known injection and plaintext recovery attacks.
  - RFCs do not insist on using SQN as nonce, possibly to support environments where AES-GCM processing happens on multiple devices for a single connection.
  - The inevitable happened: some implementations got it wrong or used bad RNGs.
  - See: Nonce Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. (Böck et al, WOOT'16)


TLS 1.3 Record Protocol

TLS 1.3 Record Protocol

- TLS 1.3 abandons legacy MEE mode, keeping only AEAD-based modes.
- Significant changes to TLS 1.2:
  - Header keeps content type (1 byte), version (2 bytes), length (2 bytes).
  - But content type is now a dummy value; true content type is now encrypted.
  - Plaintext can be arbitrarily padded to frustrate traffic analysis attacks.
  - Explicit, SQN-based nonce construction.
  - Support for AES-GCM, ChaCha20-Poly1305 so far.
  - No compression.
  - No additional data.
  - Explicit limits on key usage.


Streaming Secure Channels

Streaming Secure Channels

- TLS is not alone in presenting a streaming interface to applications.
- Also SSH "tunnel mode", QUIC.
- What security can we hope for from such a channel?
- In Fischlin-Günther-Marson-Paterson (2015), we provide a systematic study of *streaming secure channels*.
- We took an API-centric view, working top-down from application view rather than bottom-up from AEAD.

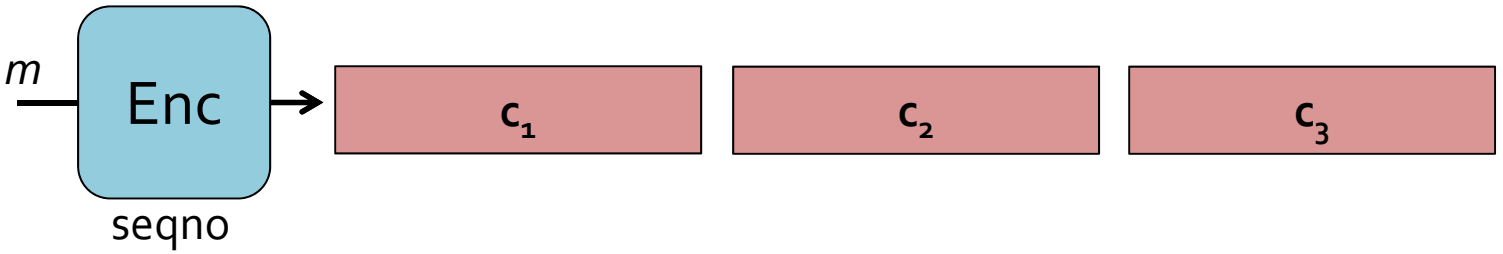
Streaming Secure Channels

- Defining CCA and integrity notions in the full streaming setting is non-trivial!
  - Hard part is to define when adversary's decryption queries deviate from a sent stream, and from which point on to suppress decryption oracle outputs
- We develop streaming analogues of IND-CPA, IND-CCA, INT-PTXT and INT-CTXT
- We recover an analogue of the classic relation

IND-CPA + INT-CTXT ➔ IND-CCA

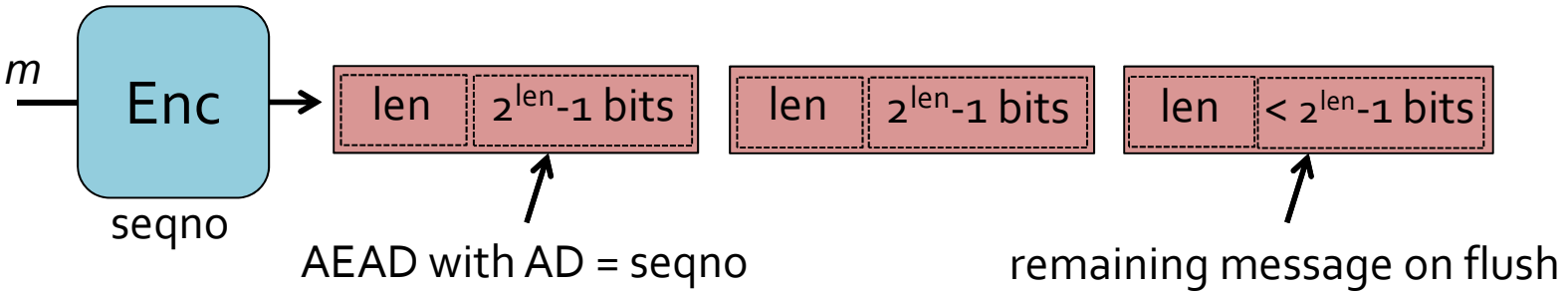
Streaming Secure Channels

- We give a generic construction for a secure streaming channel that validates the TLS design
- The construction uses AEAD as a component
- Security as streaming channel follows from standard AEAD security properties



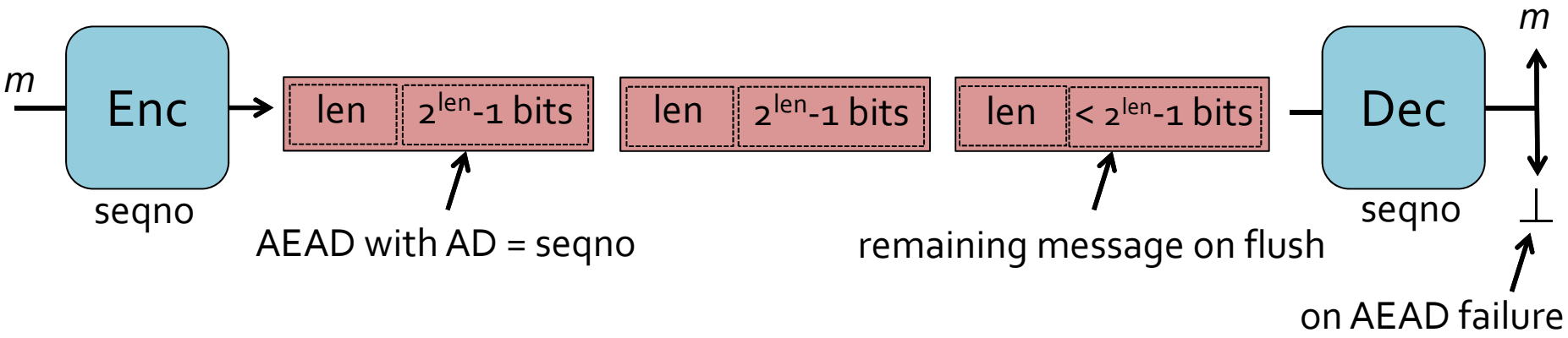
Streaming Secure Channels

- We give a generic construction for a secure streaming channel that validates the SSL/TLS design
- The construction uses AEAD as a component
- Security as streaming channel follows from standard AEAD security properties



Streaming Secure Channels

- We give a generic construction for a secure streaming channel that validates the SSL/TLS design
- The construction uses AEAD as a component
- Security as streaming channel follows from standard AEAD security properties



Concluding Remarks

Concluding remarks

- Secure channels are one of the most basic cryptographic applications.
- We do not have formal models for secure channels that accurately capture all the features expected by implementers.
- TLS as a case study highlights many of the real-world issues.
    - Design in the absence of good theory.
    - Legacy and slow adoption of better crypto.
    - Weak algorithms were hard to remove.
    - Exploitation of novel network-based, side-channel attacks.

Concluding Remarks

- Gap between AEAD and desired properties of secure channels is large.
- AEAD is still a very useful primitive: it's a good place for cryptographers to put their abstraction boundary, and a core component of secure channel constructions.
- But the programmer's API may look very different.
- Many interesting and challenging open problems yet to be worked on.