Lucky 13, BEAST, CRIME,… Is TLS dead, or just resting?

Kenny Paterson

Information Security Group

ROYAL HOLLOWAY UNIVERSITY OF LONDON

Overview

Introduction to TLS (and why YOU should care)

BEAST and CRIME

Lucky 13 and RC4 attacks

Current/future developments in TLS

About the Speaker

Academic

- But spent 5 years in industrial research lab
- Still involved in IPR, consulting, industry liaison

RHUL since 2001

- "You are teaching Network Security"
- Leading to research into how crypto is used in Network Security

EPSRC Leadership Fellow, 2010-2015

- "Cryptography: Bridging Theory and Practice"
- Support from I4, HP, BT, Mastercard, CPNI
- Attacks on IPsec (2006, 2007), SSH (2009)
- So what about TLS?

A Word from my Sponsors

EPSRC

Pioneering research and skills

TLS – And Why You Should Care

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?

TLS – And Why You Should Care

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, …
- Not yet Yahoo webcam traffic, sadly.

TLS has become the de facto secure protocol of choice.

- Used by hundreds of millions of people and devices every day.
- A serious attack could be catastrophic, both in real terms and in terms of perception/confidence.

TLS – And Why You Should Care

TLS has been in the news…..

BEAST, CRIME, Lucky 13, RC4 weaknesses.

Renegotiation attack (2009), triple Handshake attack (3/2014).

Poor quality of implementations (particularly in certificate handling).

Not only the Apple *goto fail*, but also "Why Eve and Mallory Love Android" and "The most dangerous code in the world".

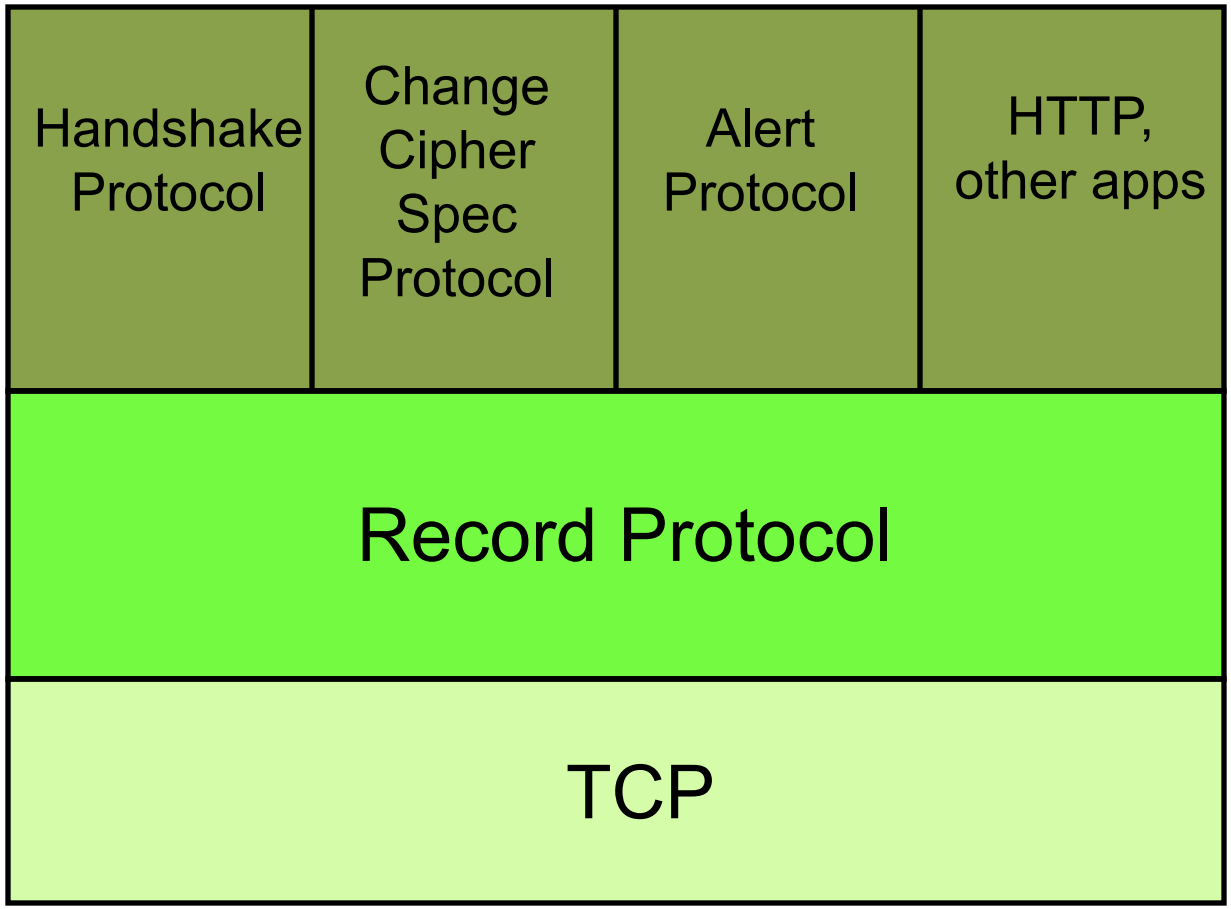
Why?

Higher profile means more attention.

More attention begets researcher interest, creating a **virtuous** cycle.

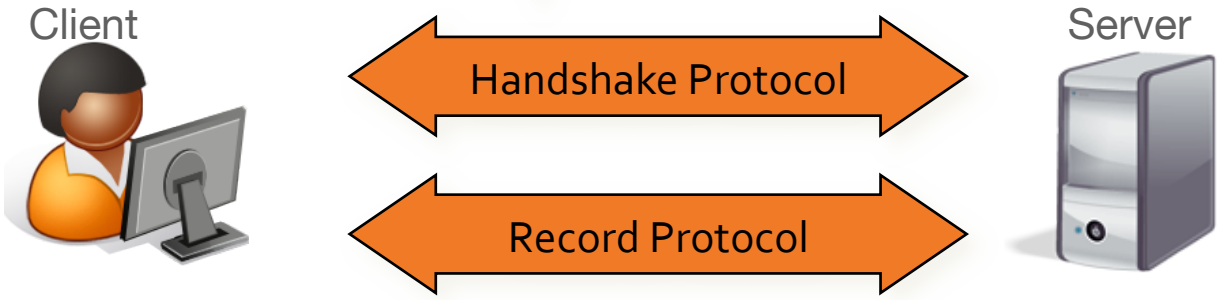
Virtuous because TLS gets better the more we analyse it and eliminate its weaknesses.

TLS Protocol Architecture



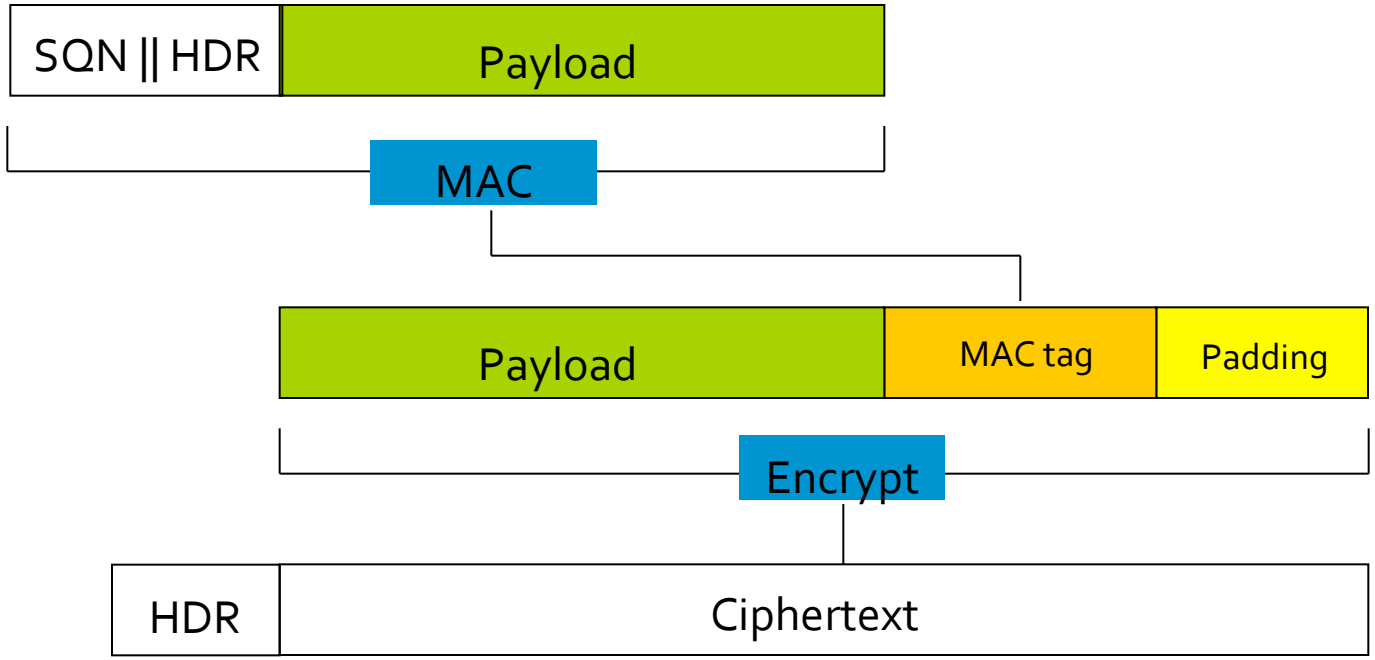
Simplified View of TLS

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



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

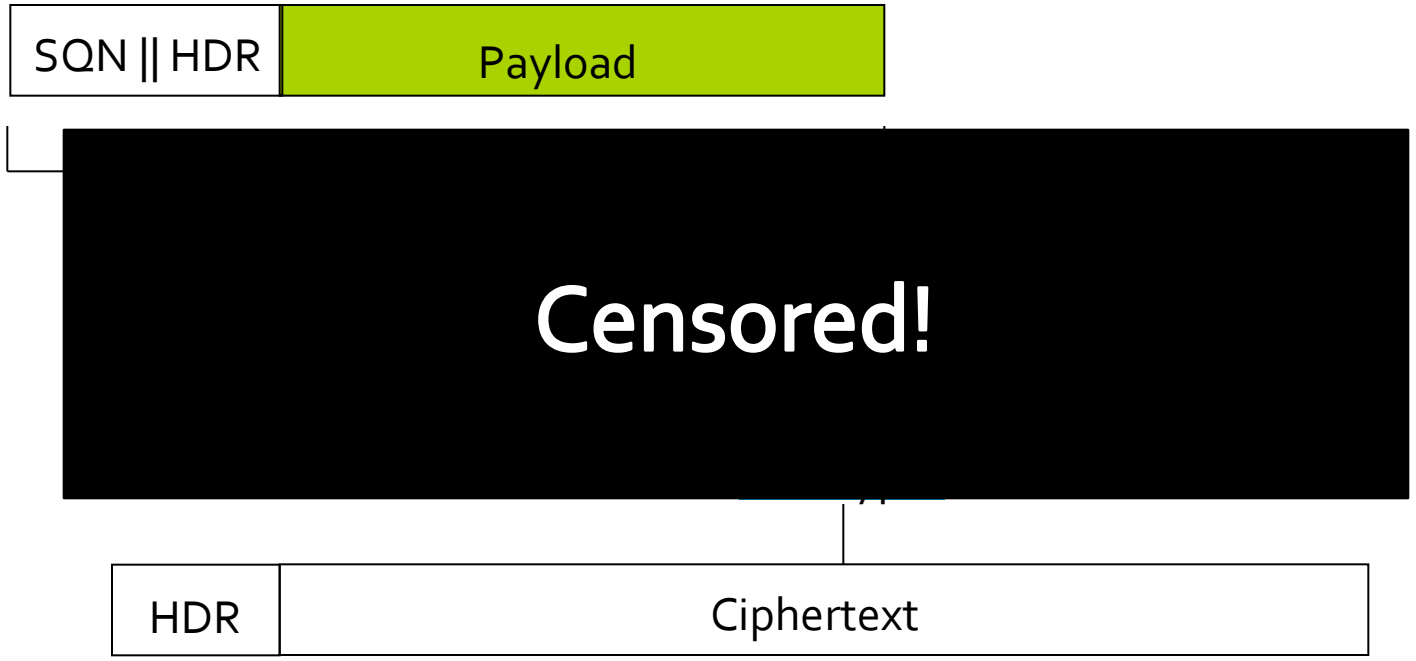
TLS Record Protocol: MAC-Encode-Encrypt



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

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

TLS Record Protocol: AE (GCM and CCM)



AE and TLS Record Protocol

28% of Alexa top 200k servers support TLS 1.1 or higher.

(source: ssl pulse, Feb. 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)

Stronger, modern AE designs are not yet in universal use.

6 months ago, the situation was much bleaker!

Attacks have driven accelerating pace of adoption of TLS 1.2.

CRIME and BEAST Attacks

BEAST

Relevant only for CBC-mode ciphersuites in SSL 3.0 and TLS 1.0.

Exploits use of non-random IV in those ciphersuites.

- Theoretical attack pointed out to IETF in 1995 (Rogaway)

Made practical by Duong and Rizzo in 2011.

- Via malicious Javascript running in the browser.
- Recovery of HTTP session cookies (and more).

Now (mostly) mitigated in browsers using 1/n-1 record splitting.

- Makes it *almost* defensible to continue using CBC-mode in TLS 1.0.

Many experts recommended RC4 as a mitigation.

- More later!

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.

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 2011.

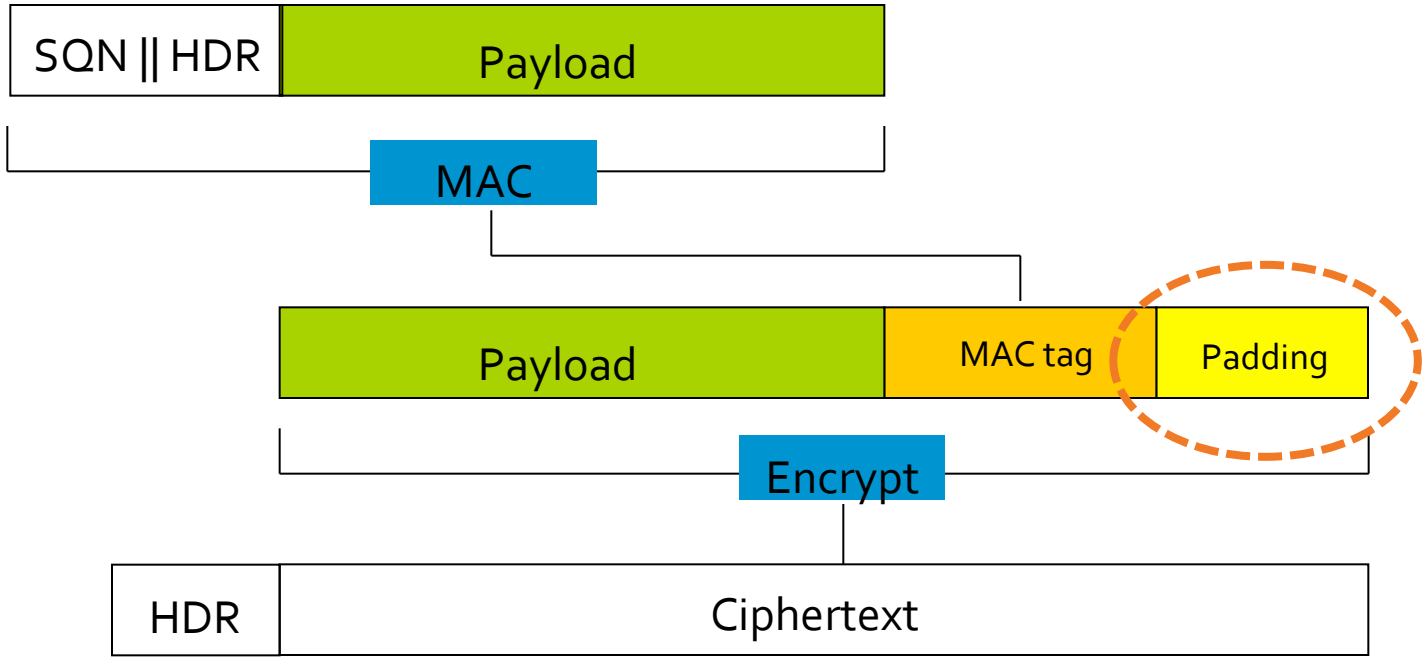
- Attacks really do get better (worse!) with time.
- Do you see the emerging theme here?

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

- And maybe they can be used yet elsewhere?

Lucky 13 Attack

TLS Record Protocol: MAC-Encode-Encrypt



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

TLS and Padding Oracles

[Vo2,CHVVo3]:

Specifics of TLS padding format can be exploited to mount a plaintext recovery attack.

No chosen-plaintext requirement, c.f. BEAST.

The attack depends on being able to distinguish good from bad padding.

- In practice, this is done via a timing side-channel.
- The MAC is only checked if padding good, and the MAC is always bad in the attack.
- Distinguish cases by timing TLS error messages.

TLS and Padding Oracles

[V02,CHVV03]:

The attack is multi-session.

- Each trial in the attack causes a fatal error and TLS session termination.
- The attack still works if a fixed plaintext is repeated in a fixed location across many TLS sessions.
- e.g. a password in an automated login.
- Modern viewpoint: use BEAST-style malware and target HTTP cookies.

Padding oracle attack worked for OpenSSL.

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

Countermeasures?

Redesign TLS:

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

Too invasive, did not happen.

Switch to RC4?

Or add a fix to ensure uniform errors?

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

So how should TLS implementations ensure uniform errors?

Ensuring Uniform Errors

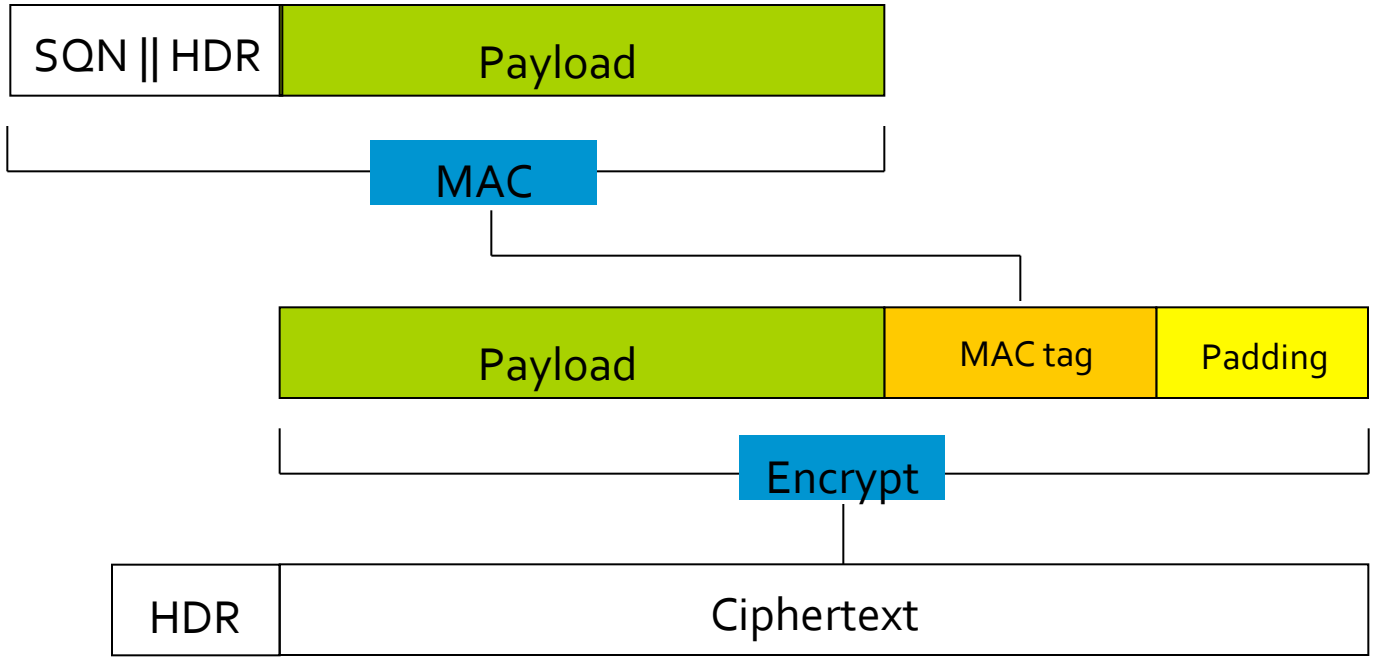
From the TLS 1.2 specification:

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

In general, the best way to do this is to compute the MAC even if the padding is incorrect, and only then reject the packet.

Compute the MAC on what though?

TLS Record Protocol: MAC-Encode-Encrypt



Problem is how to parse plaintext as payload, padding and MAC fields?

Ensuring Uniform Errors

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

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

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

- And variant attacks against those that do not.

Applies to all versions of SSL/TLS.

- SSLv3.0, TLS 1.0, 1.1, 1.2.
- And DTLS too!

Demonstrated in the lab against OpenSSL and GnuTLS.

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

Lucky 13 Timescales

We* started work in December 2011.

Key breakthrough in March 2012 (+4 months)

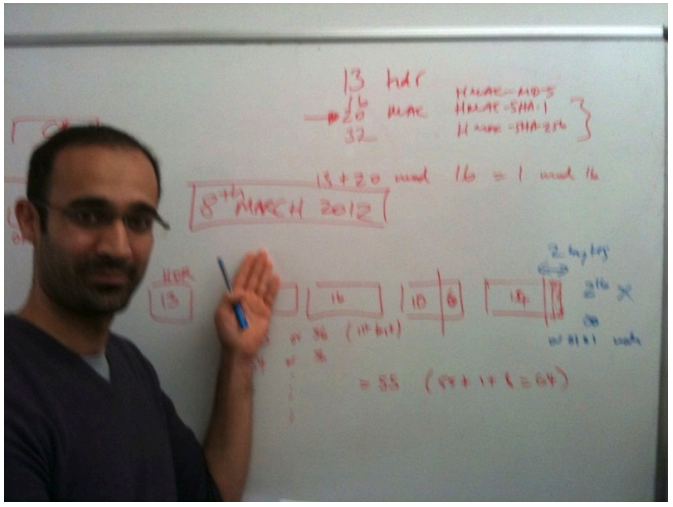
Research paper completed November 2012 (+11 months).

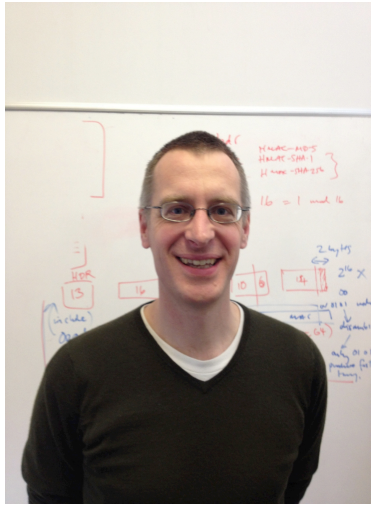
Disclosure and patching November 2012 – February 2013.

Attack publicly disclosed in February 2013 (+15 months).

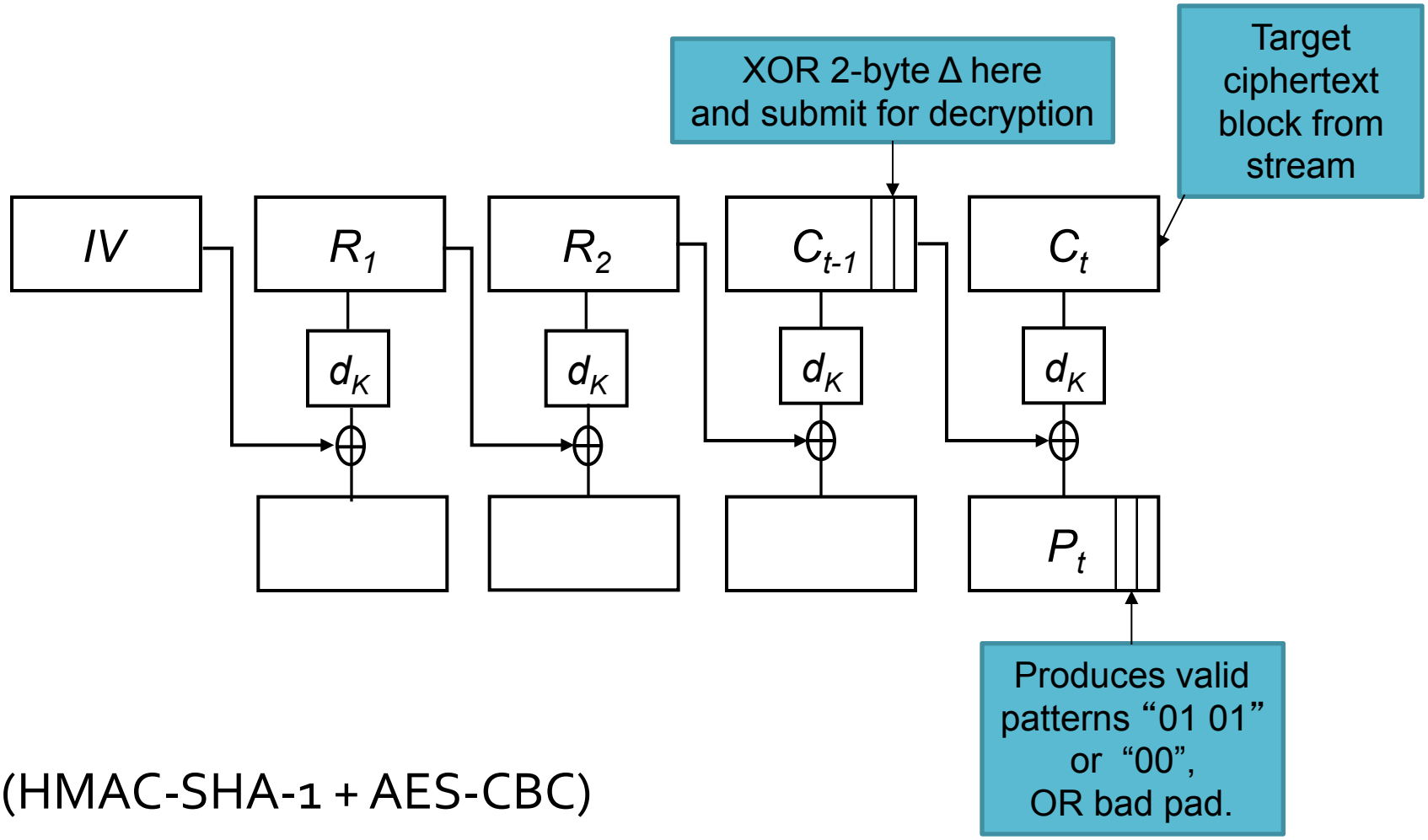
Research paper presented in May 2013 (+18 months).

*



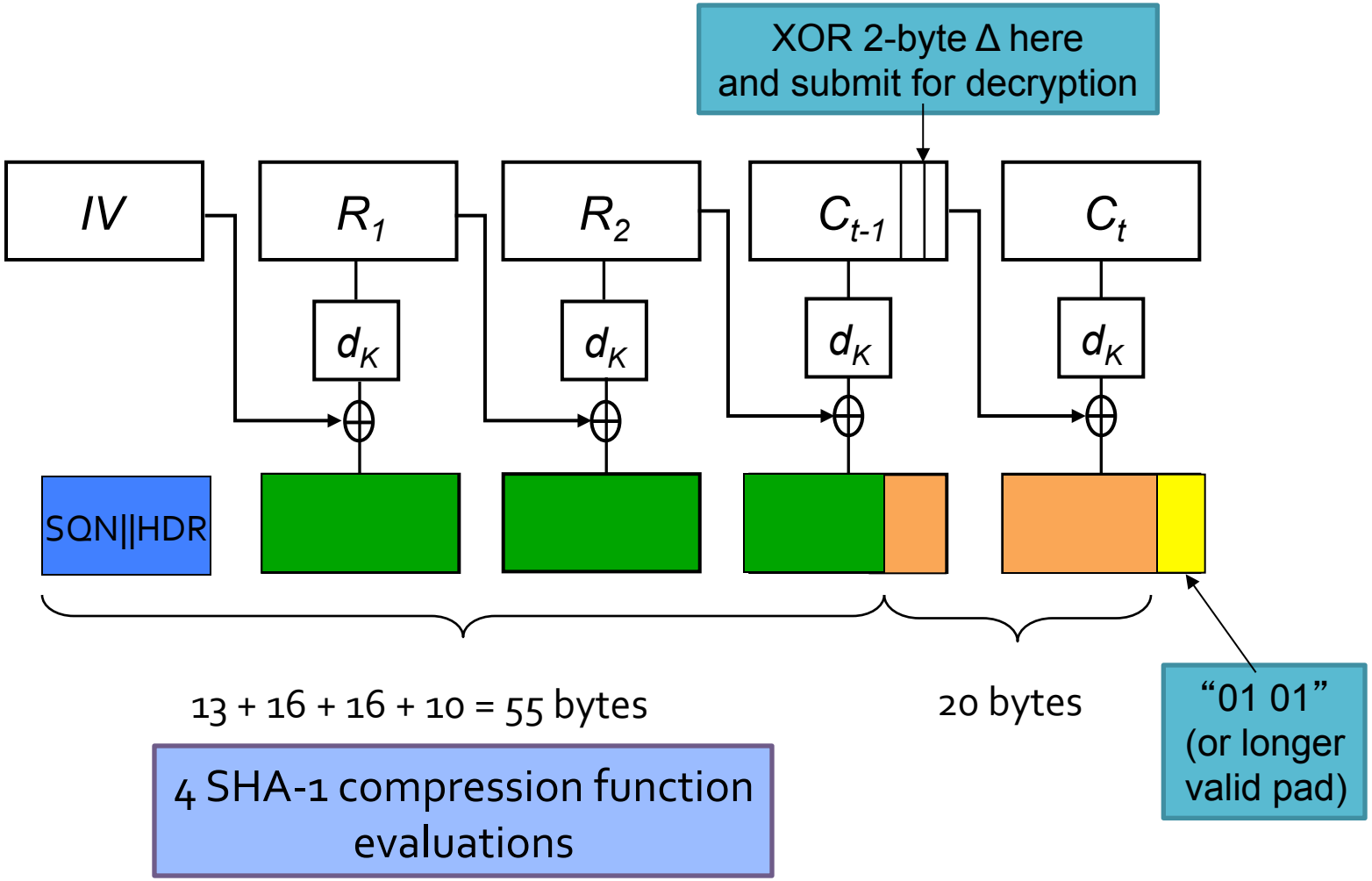


Lucky 13 – Plaintext Recovery

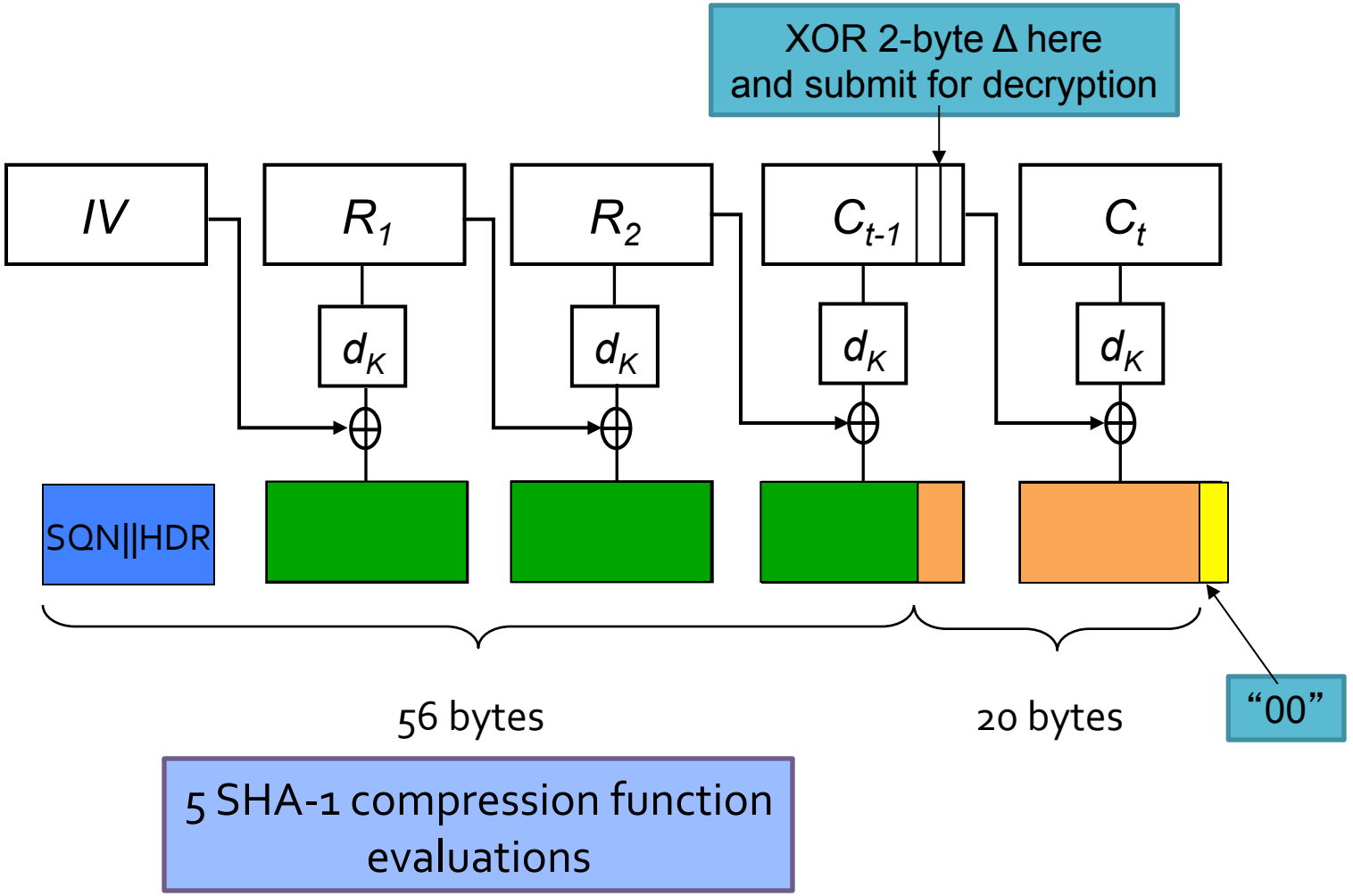


(HMAC-SHA-1 + AES-CBC)

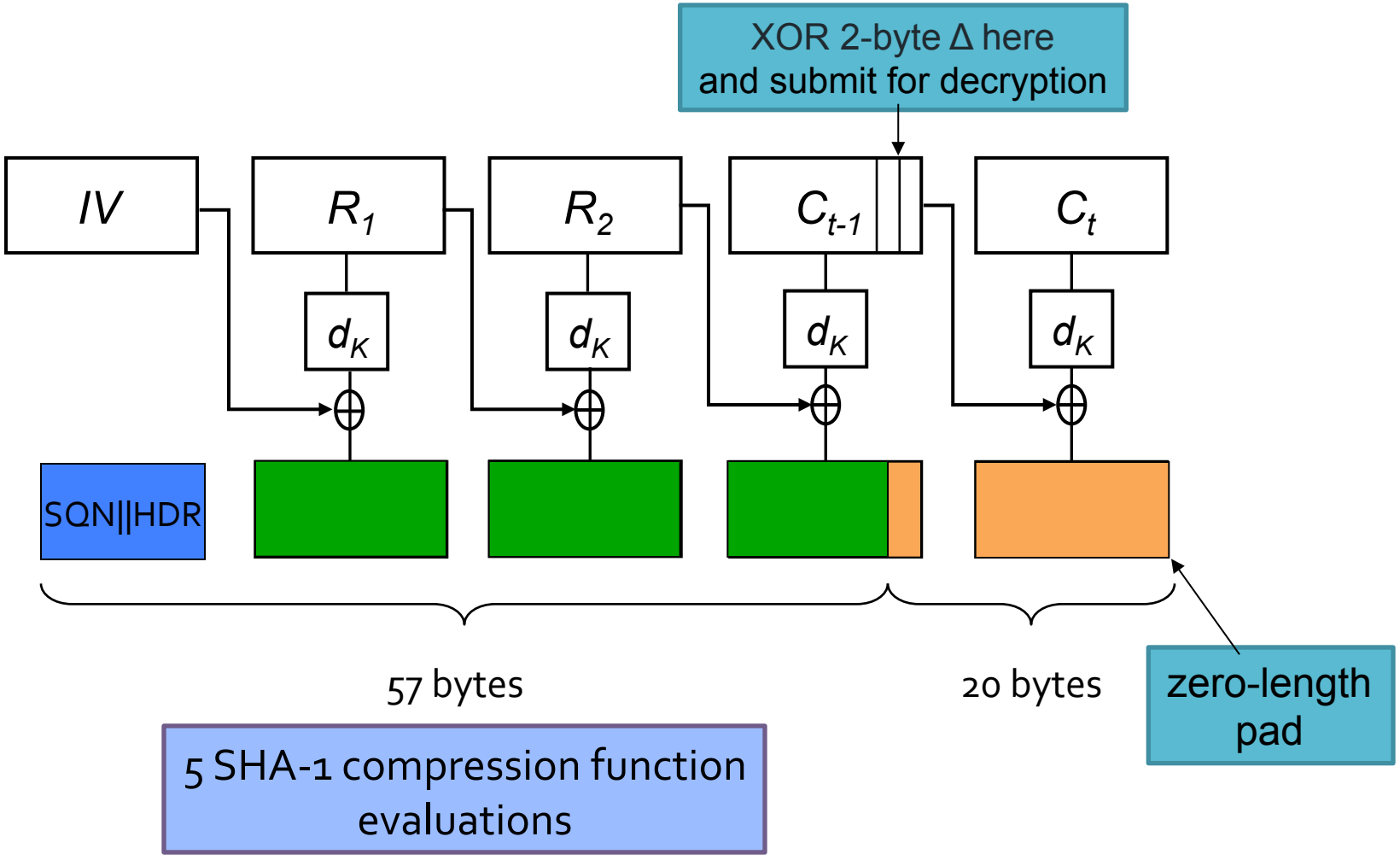
Case 1: "01 01" (or longer valid pad)



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 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 some standard CBC algebra.
- Attack then extends easily 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.

So what's all the fuss about?

Lucky 13 – Improvements (Attacks Get Better!)

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

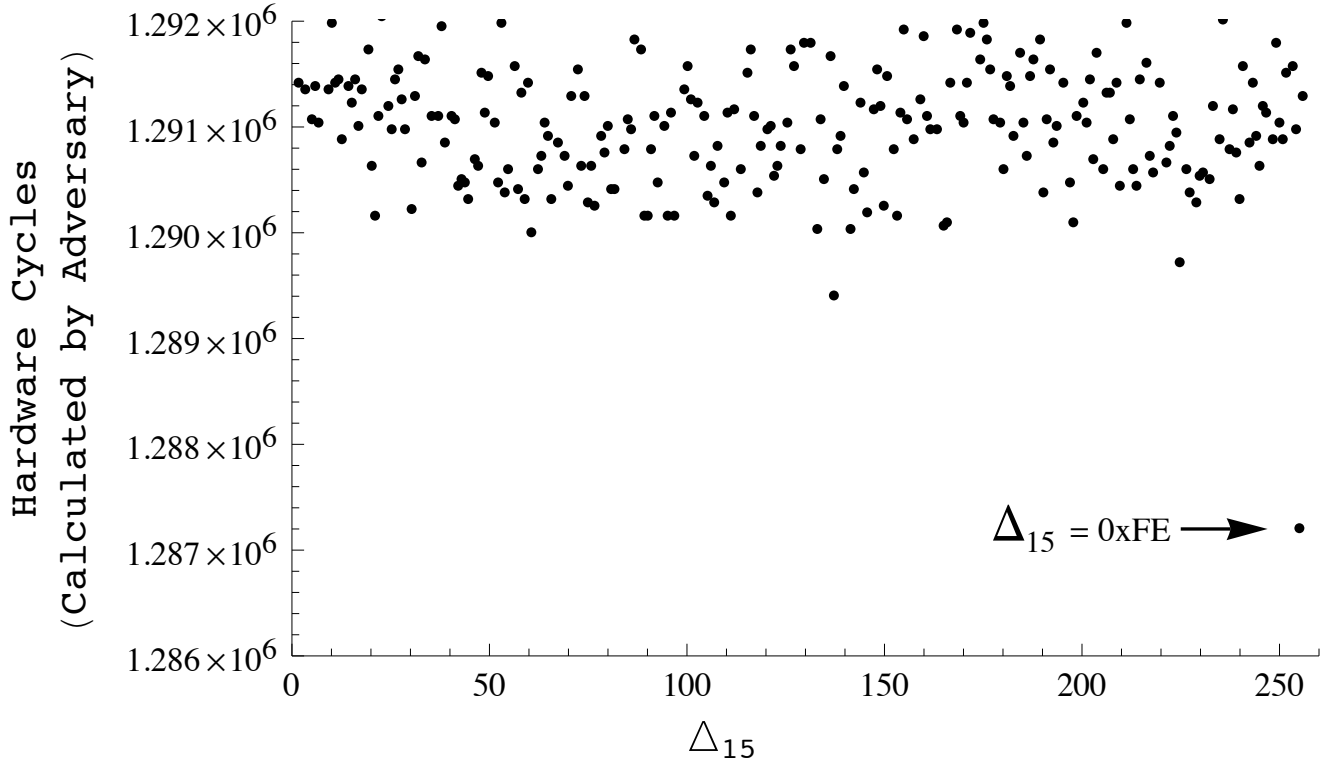
If the plaintext is base64 encoded, then we only need 2^6 attempts per byte.

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

BEAST-style attack targeting HTTP cookies.

- Malicious client-side Javascript makes HTTP GET requests.
- TLS sessions are automatically generated and HTTP cookies attached.
- Pad GET requests so that 1-out-of-2 condition always holds.
- Cost of attack is 2^{13} GET requests per byte of cookie.
- Now a practical attack!

Lucky 13 – Experimental Results



Byte 14 of plaintext set to 01; byte 15 set to FF.

Modify Δ in position 15.

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

Median times (noise not shown).

Lucky 13 – Disclosure

How do you disclose an attack on a protocol that has dozens of different implementations and millions of users?

- Coordination amongst all stakeholders.
- Risk of leakage and panic before agreed time.
- IETF "owns" the specification, so a good place to start?

Working with IETF:

- Good initial communication, contacts into Google, Mozilla, etc.
- Technical idea from ekr to improve the attack!
- But overall "no" to taking responsibility for coordination.
- Became clear that we would have to take the lead and communicate with individual vendors ourselves.

Lucky 13 – Disclosure

We opened up multiple channels of communication.

- OpenSSL, Mozilla, Cisco, Apple, Microsoft, Google, Oracle, Opera, BouncyCastle, F5, and numerous open source projects.
- NOT end users.
- Hundreds of e-mails, December 2012 to February 2013.

Mostly great co-operation, but some not so great.

- Investment of time to build trust.
- No legal spectres were raised; no NDAs were imposed.
- We helped a number of vendors with patch testing.

Also building a website, preparing a press release, priming journalists and bloggers.

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 – Countermeasures

We really need constant-time decryption for TLS-CBC.

Add dummy hash compression function computations when padding is good to ensure total is the same as when padding is bad.

Add dummy padding checks to ensure number of iterations done is independent of padding length and/or correctness of padding.

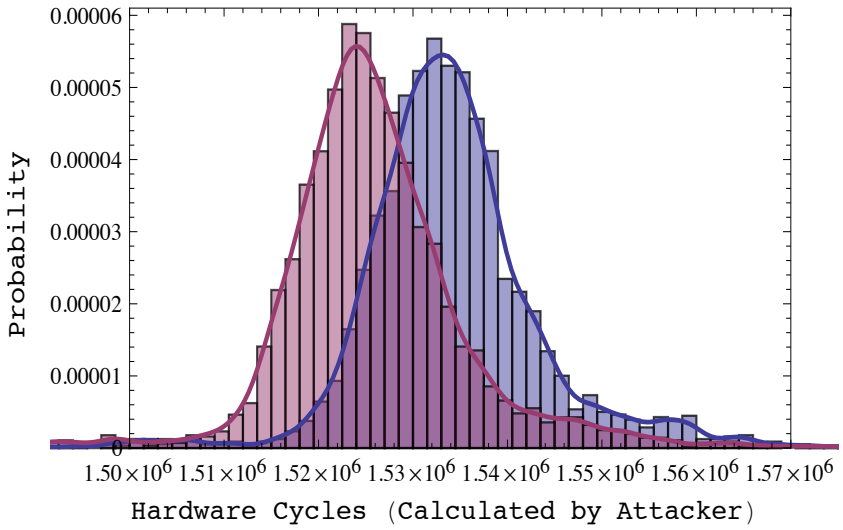
Watch out for length sanity checks too.

> Need to ensure there's enough space for some plaintext after removing padding and MAC, but without leaking any information about amount of padding removed.

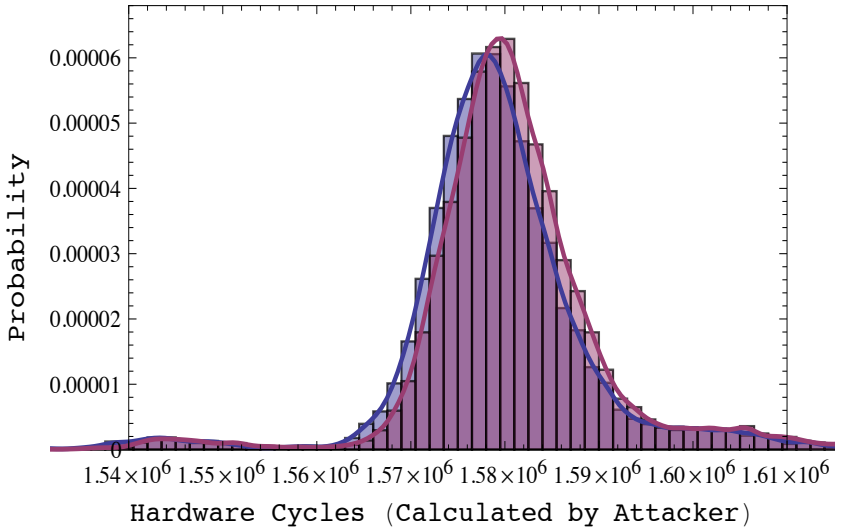
TL;DR: it's a bit of a nightmare.

Performance of Basic Countermeasures

Before



After



Better, but still not perfect.

Distinguishing attack still possible.

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

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

Lucky 13 attack shows that small details matter.

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

Other Lucky 13 Countermeasures?

Introduce random delays during decryption.

- Surprisingly **ineffective,** analysis in our paper.

Redesign TLS:

- Pad-MAC-Encrypt or Pad-Encrypt-MAC?
- Pad-Encrypt-MAC finally being developed by IETF as a TLS extension for TLS 1.1 and higher.
- Will take months/years to deploy.

Switch to TLS 1.2

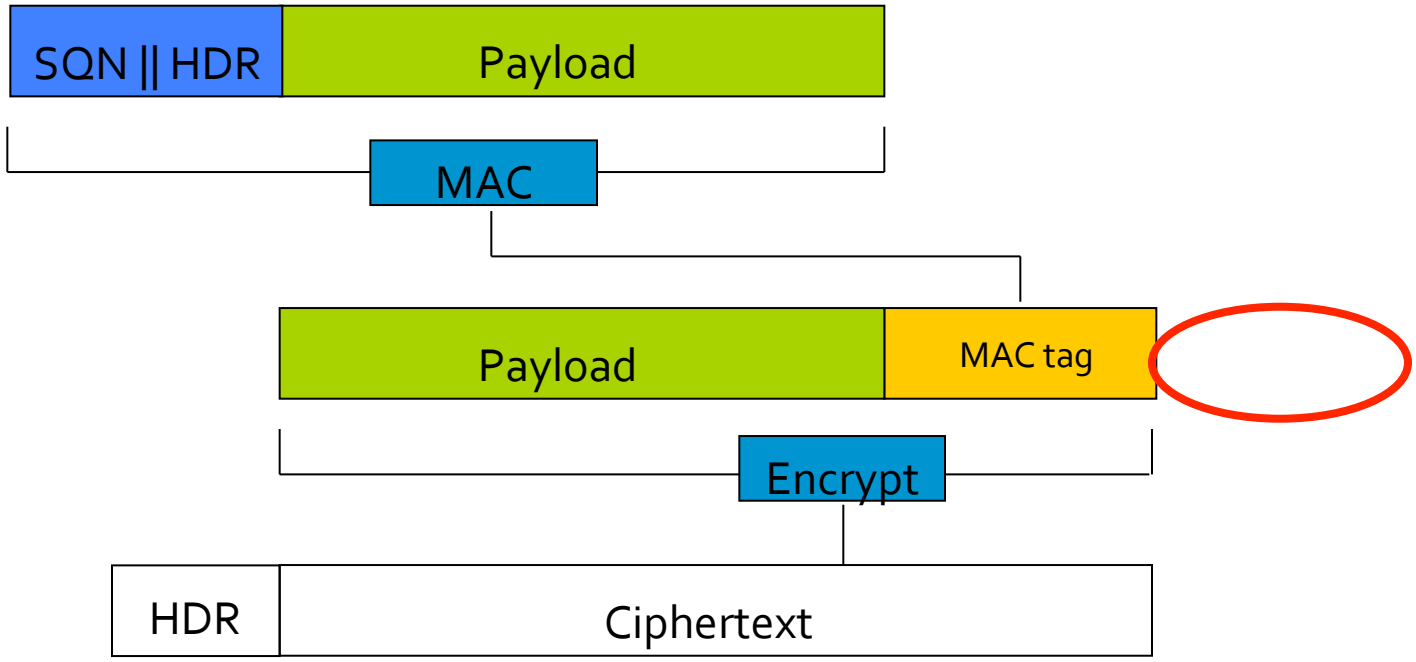
- Has support for AES-GCM and AES-CCM.
- But was not widely supported by browsers or servers at the time Lucky 13 was announced.

Switch to RC4!

- As recommended by many commentators (again!).

Attacking RC4 in TLS

TLS Record Protocol: RC4-128



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

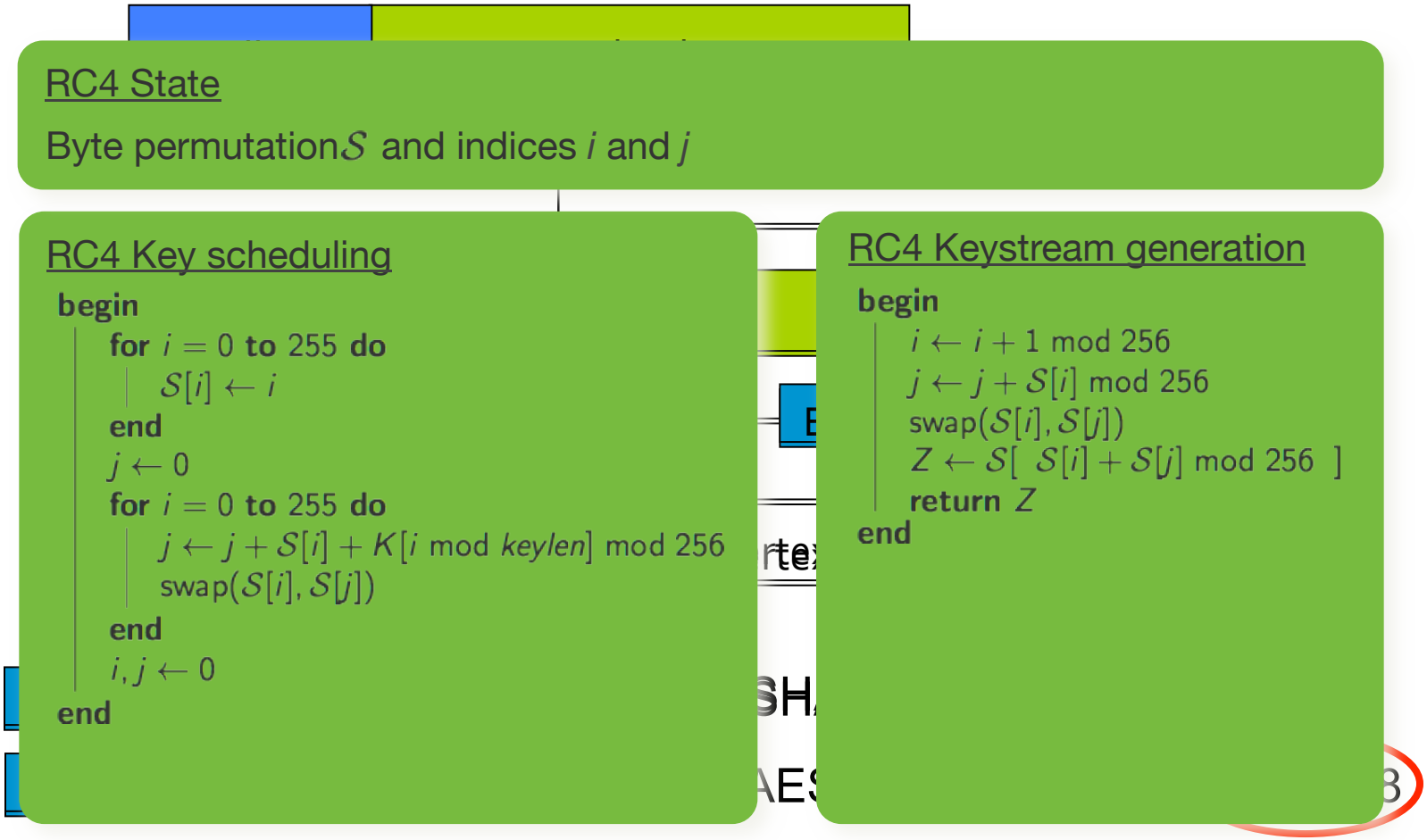
Encrypt — CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

RC4 Attack in Brief

It's also broken.

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

TLS Record Protocol: RC4-128



Use of RC4 in TLS

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

RC4 is also fast when AES hardware not available

Use of RC4 in the wild:

ICSI Certificate Notary

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

Problem: RC4 is known to have statistical weaknesses.

Single-byte Biases in the RC4 Keystream

Z_i = value of i-th keystream byte

[Mantin-Shamir 2001]:

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

[Mironov 2002]:

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

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

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

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

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

l = keylength

What's Going On Here?

Why were we still using RC4 in half of all TLS connections when we 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."

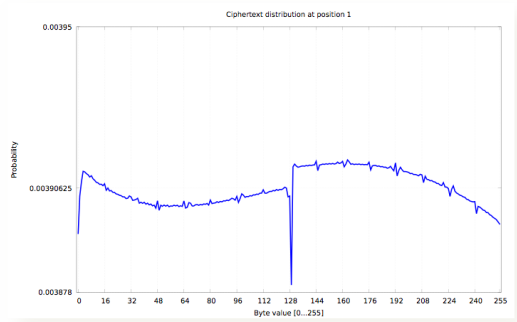
"I'm worried about BEAST on CBC mode. Experts say `use RC4'"

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

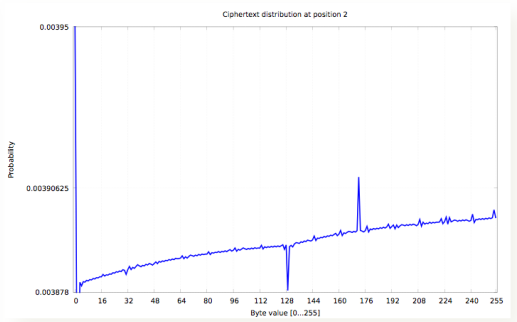
Complete Keystream Byte Distributions

Approach in [ABPPS13]:

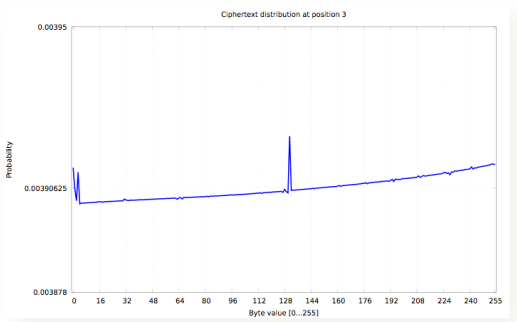
Based on the output from 2^{45} random independent 128-bit RC4 keys, estimate the keystream byte distributions for the first 256 bytes



Z_1



Z_2



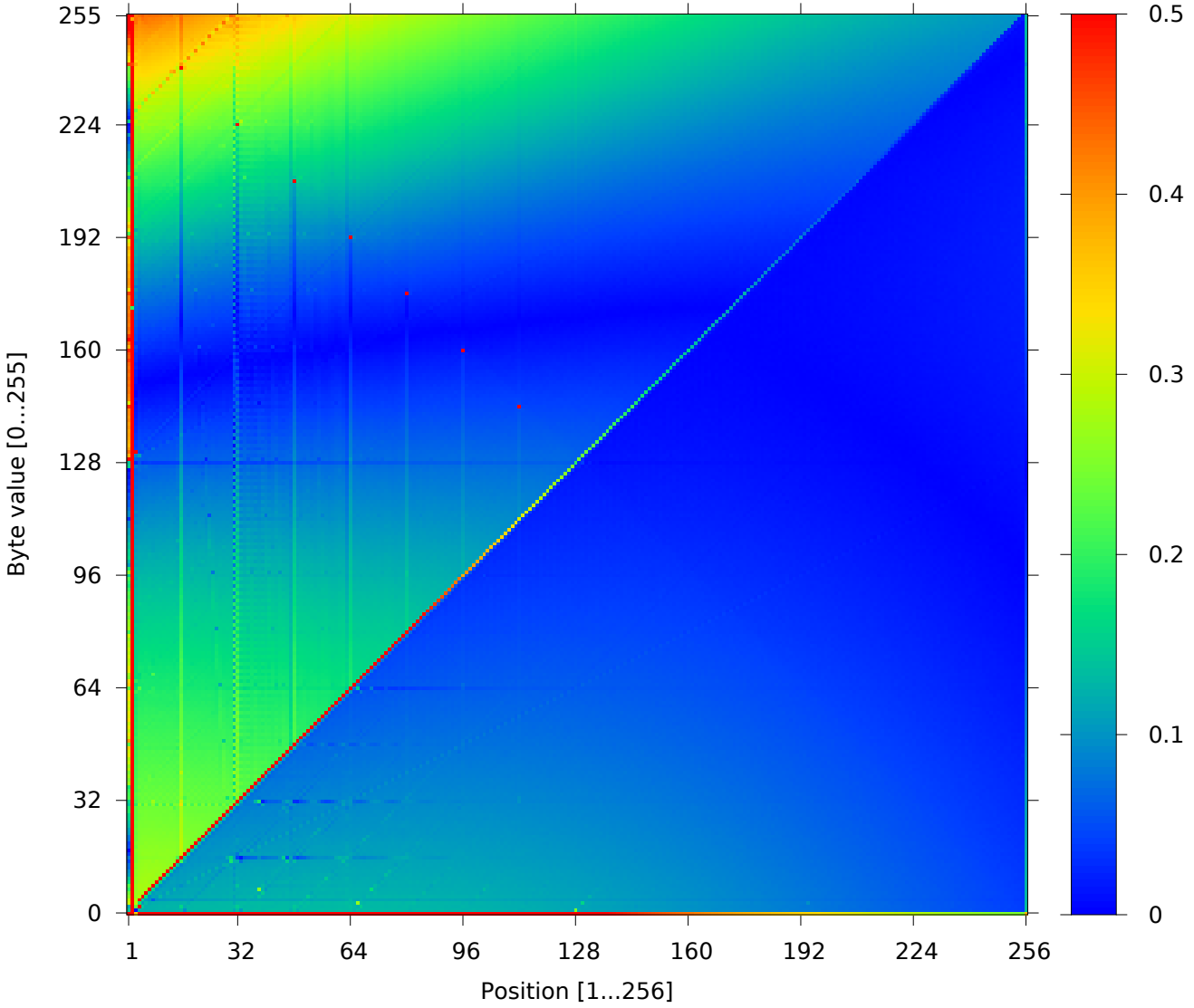
Z_3

... ...

Revealed many new biases in the RC4 keystream.

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

All the Biases



Plaintext Recovery for TLS-RC4

So what?

Using the biased keystream byte distributions, we can construct a plaintext recovery attack against TLS.

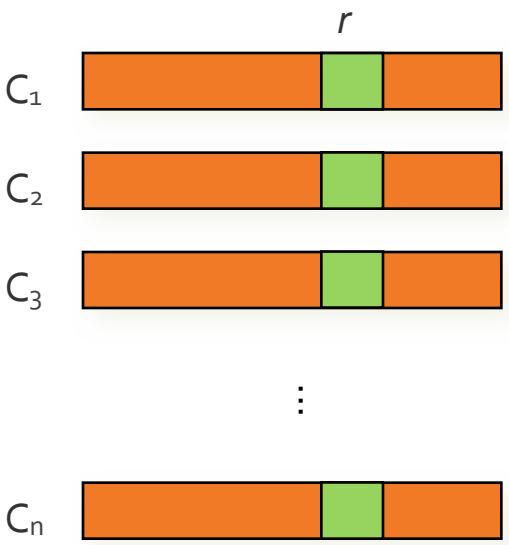
The attack requires the same plaintext to be encrypted under many different keys.

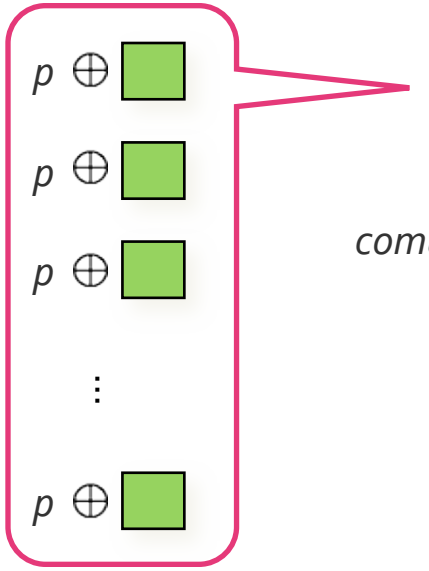
- Use Javascript in browser as mechanism, cookies as target.
- Reusing the BEAST mechanism once more.
- So there is a meaningful attack scenario!

Plaintext Recovery Using Keystream Biases

Encryptions of fixed plaintext under different keys

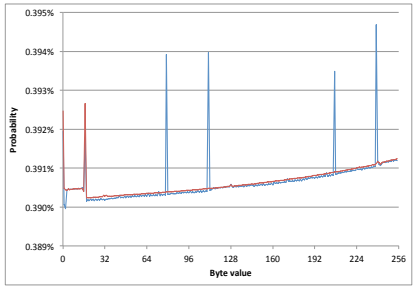
Plaintext candidate byte *p*





yields induced distribution on keystream byte *Z*

combine with known distribution



Likelihood of *p* being correct plaintext byte

Recovery algorithm:
Compute most likely plaintext byte

Details of Statistical Analysis

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

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

Bayes theorem:

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

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

Censored!

Assume Prmaximise Pr[P=p|C=c] by choice of p, we simply need to maximise:

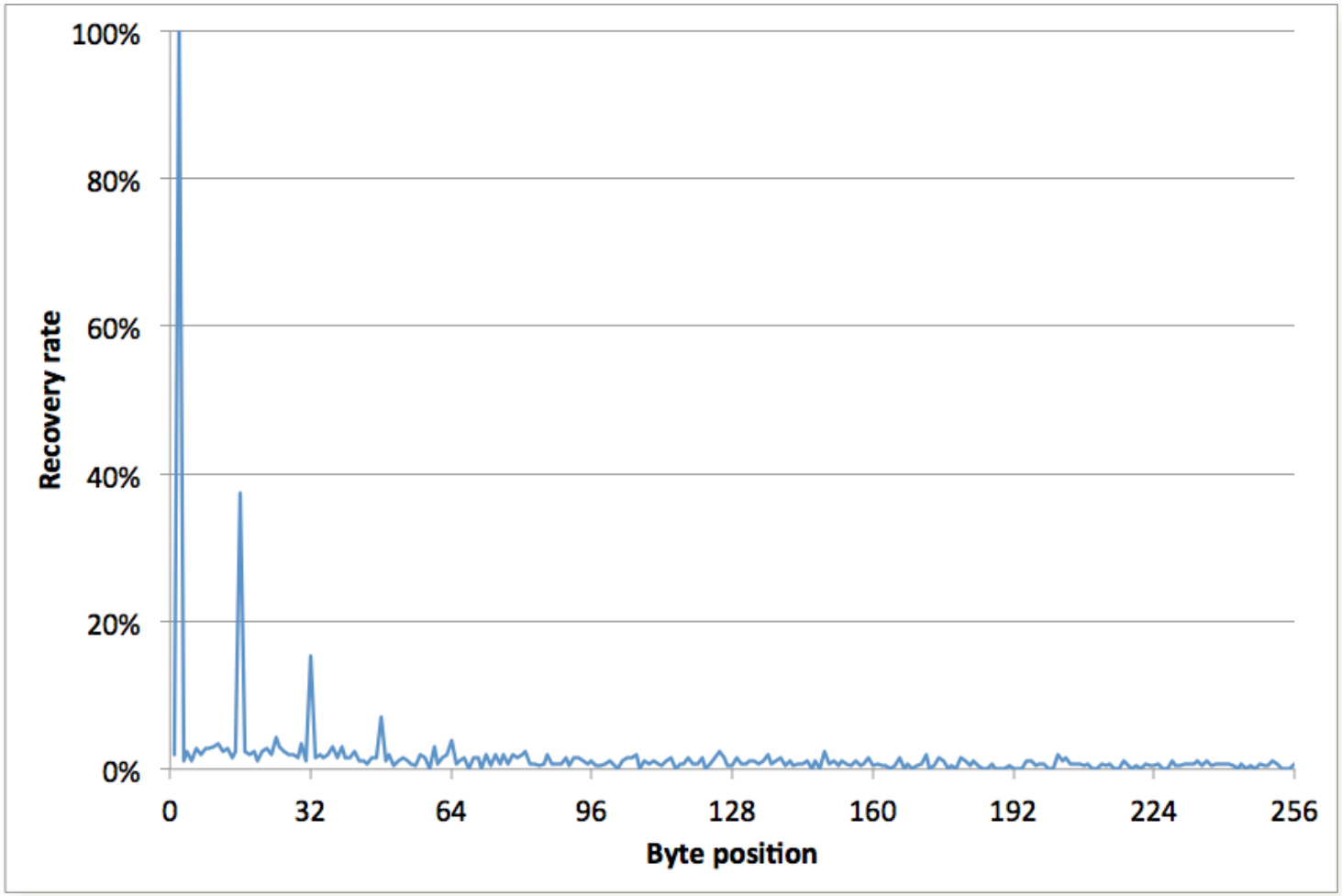
[$P=p$] is constant; Pr[$\boldsymbol{C}=\boldsymbol{c}$] is independent of the choice of p.

Then to

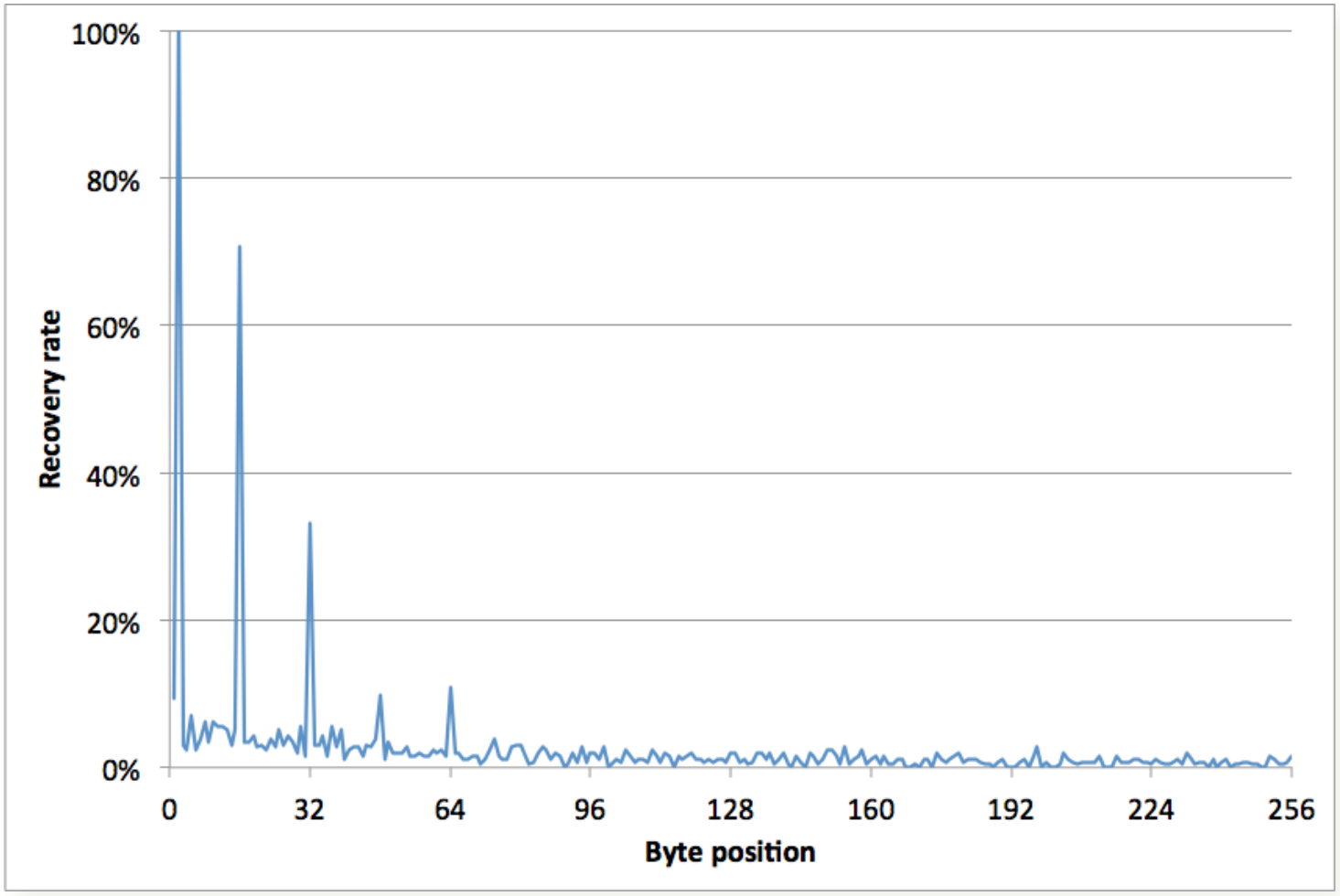
$$\Pr[\boldsymbol{Z}=\boldsymbol{c} \oplus p \mid P=p] = \frac{n!}{n_{00}!n_{01}! \ldots n_{ff}!} q_{00}^{n_{00}} q_{01}^{n_{01}} \ldots q_{ff}^{n_{ff}}$$

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

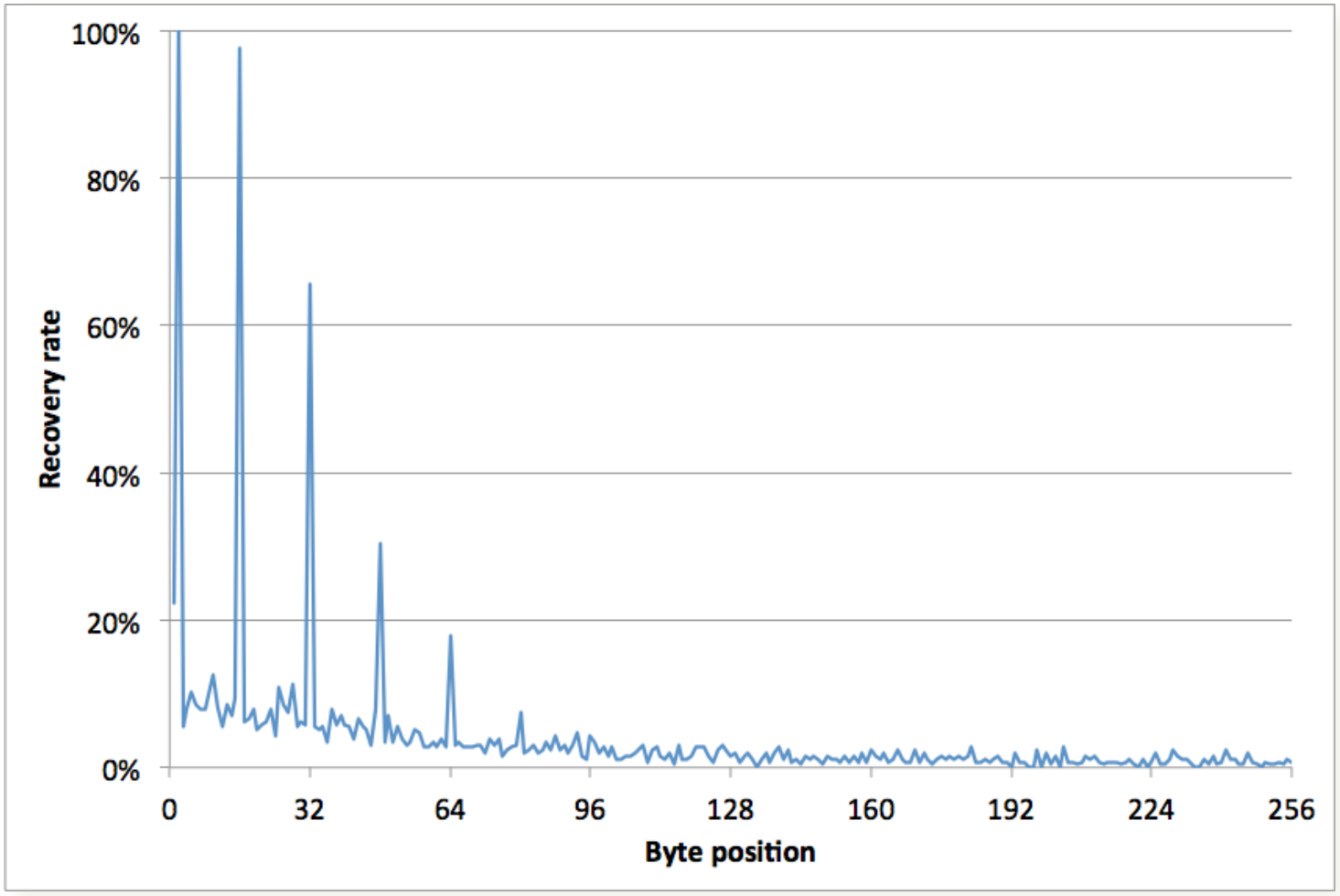
Success Probability 2^{20} Sessions



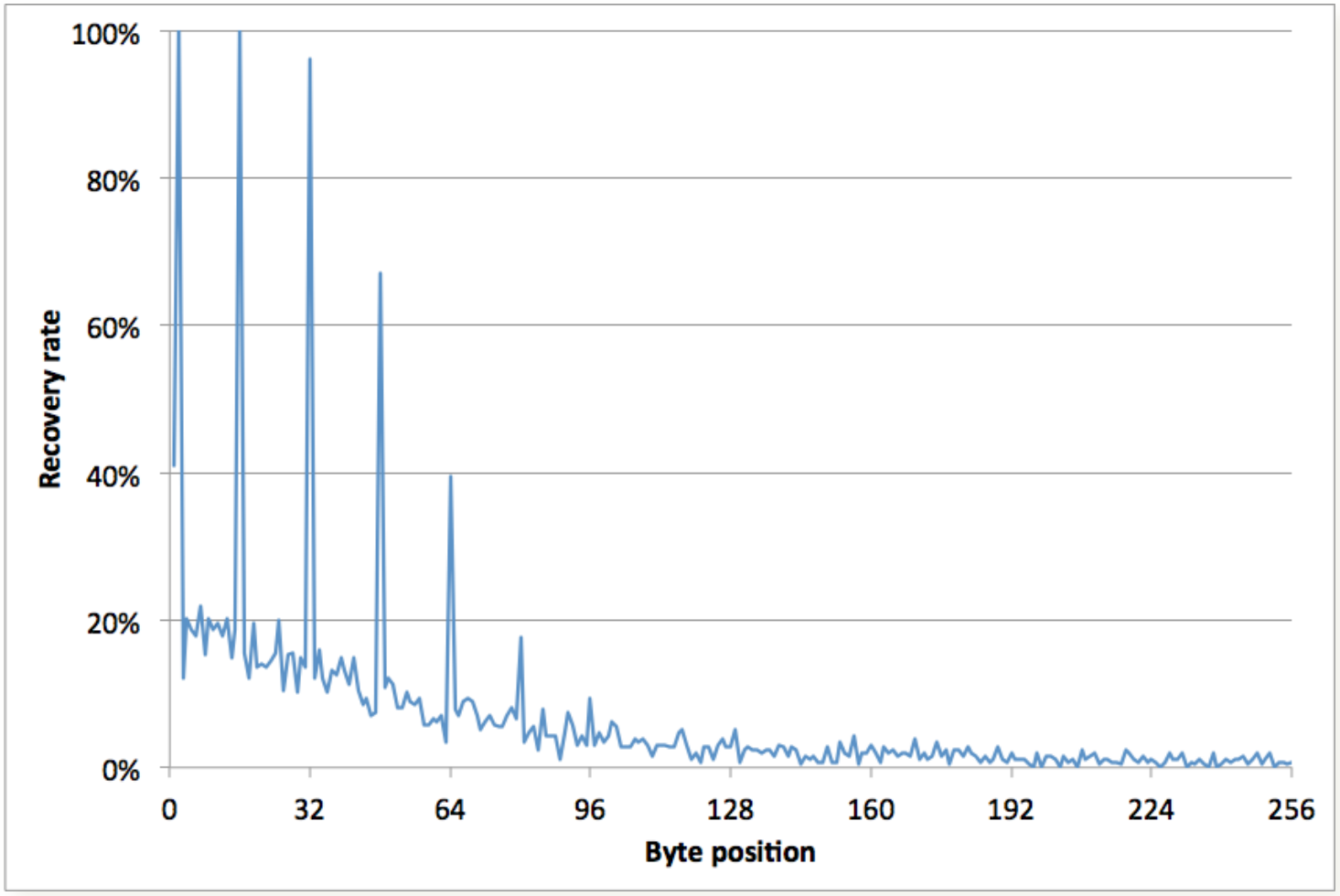
Success Probability 2^{21} Sessions



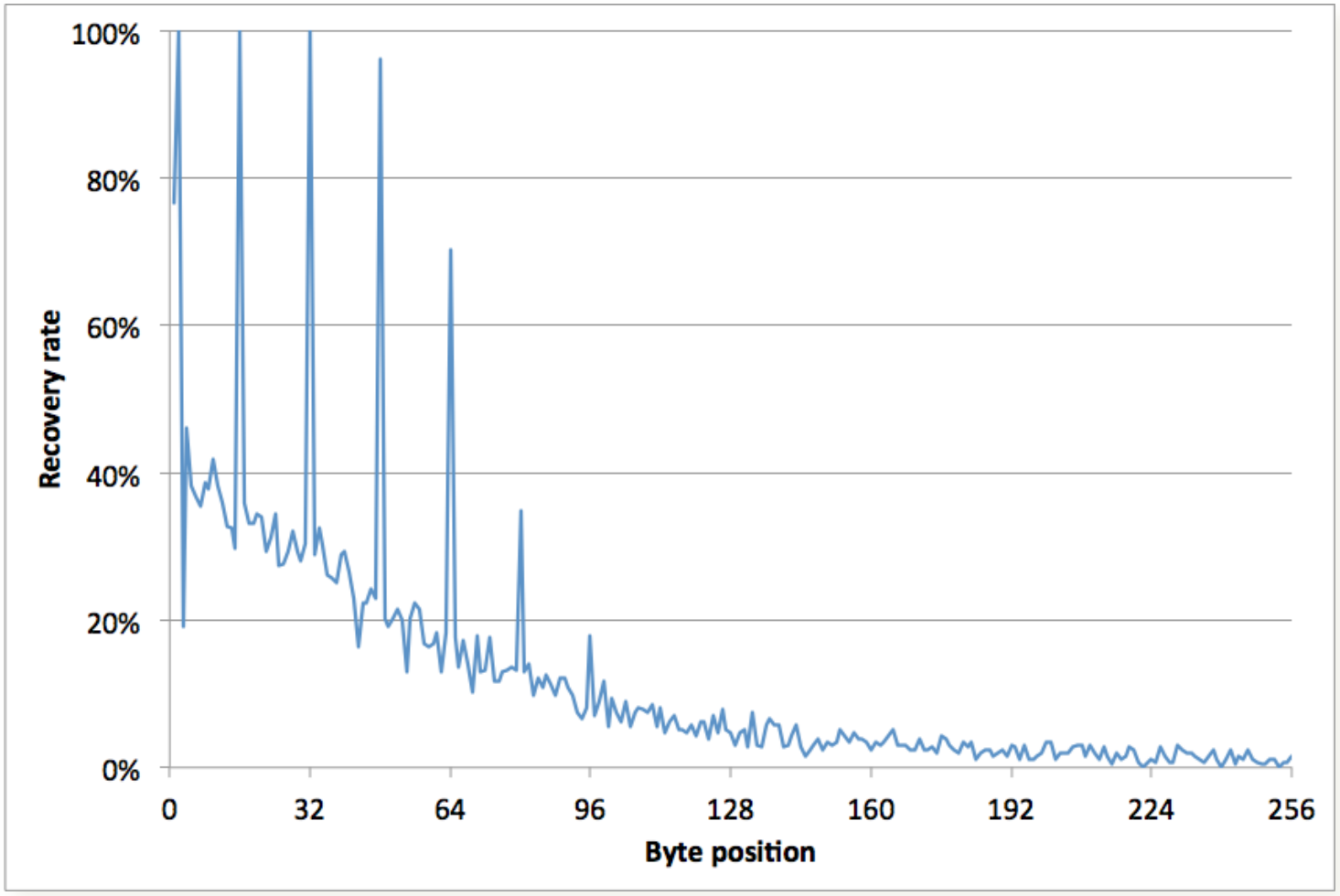
Success Probability 2^{22} Sessions



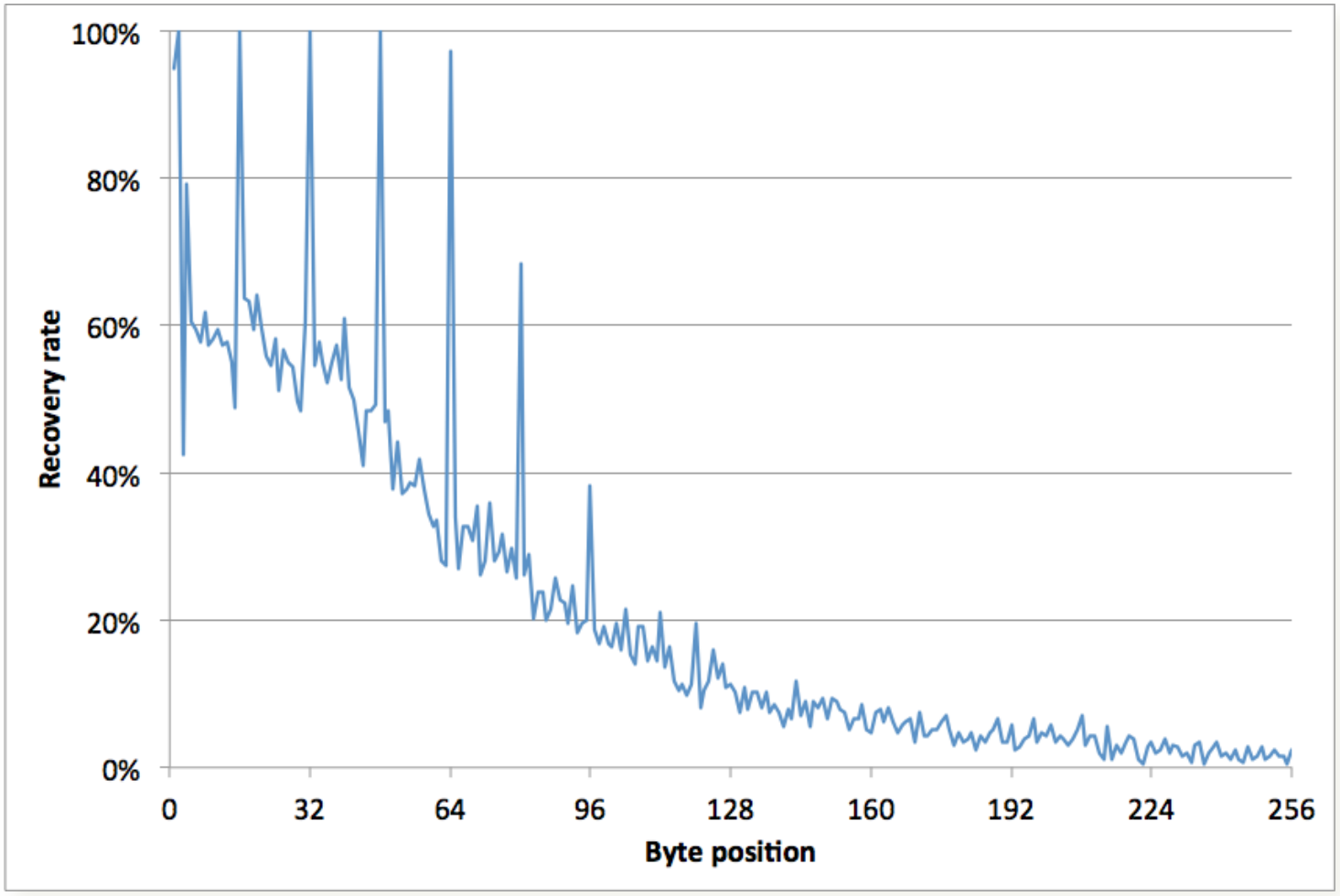
Success Probability 2^{23} Sessions



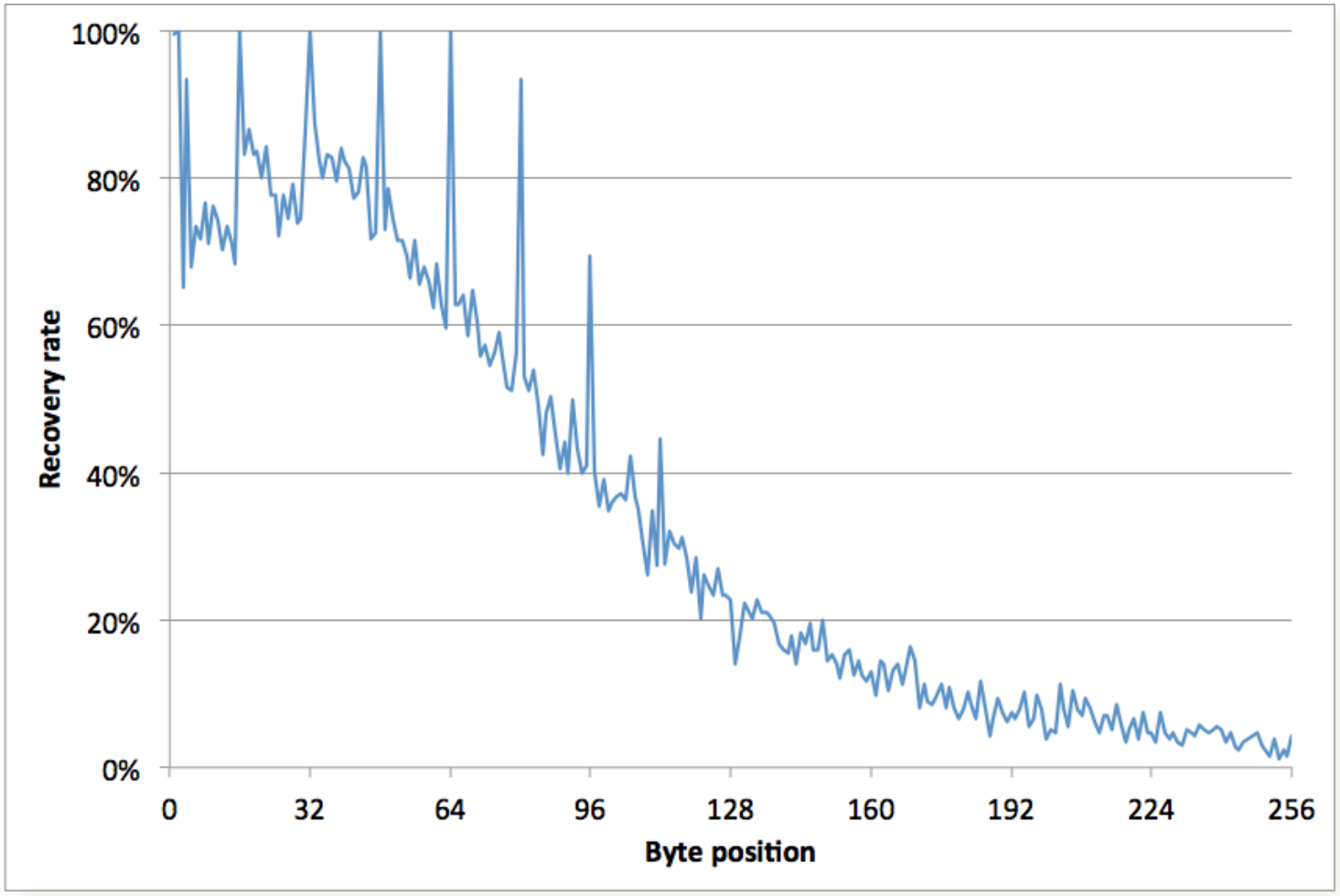
Success Probability 2^{24} Sessions



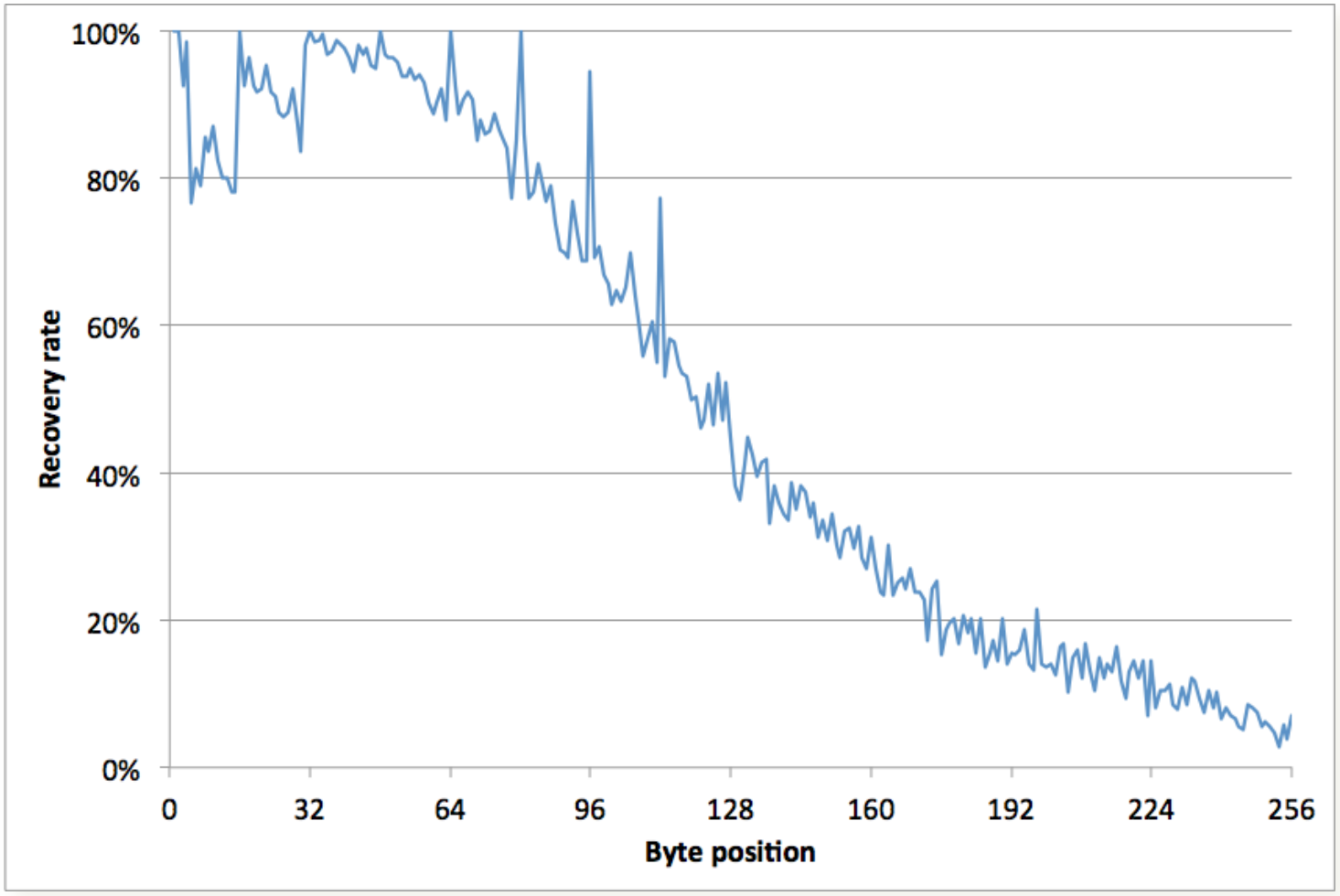
Success Probability 2^{25} Sessions



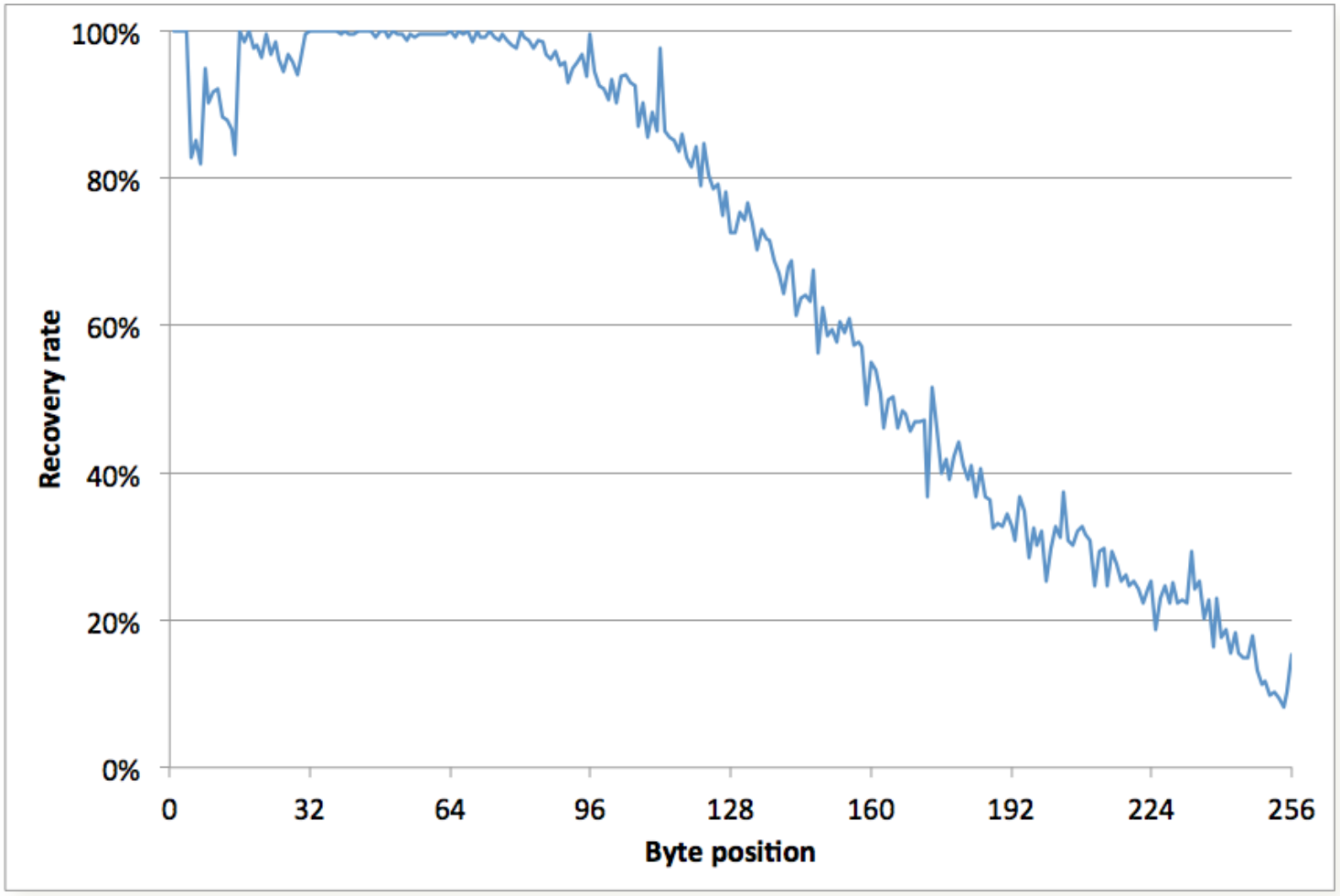
Success Probability 2^{26} Sessions



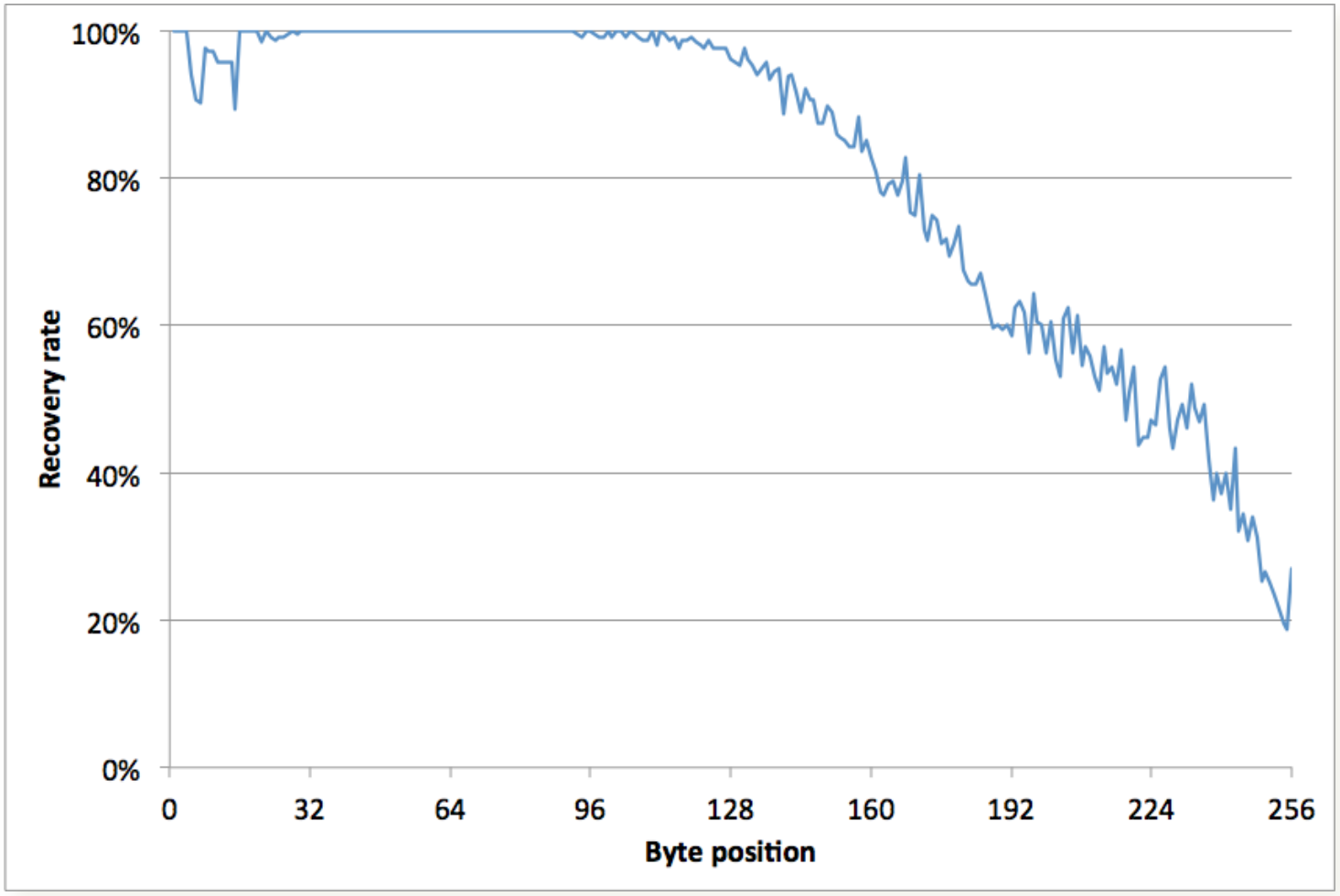
Success Probability 2^{27} Sessions



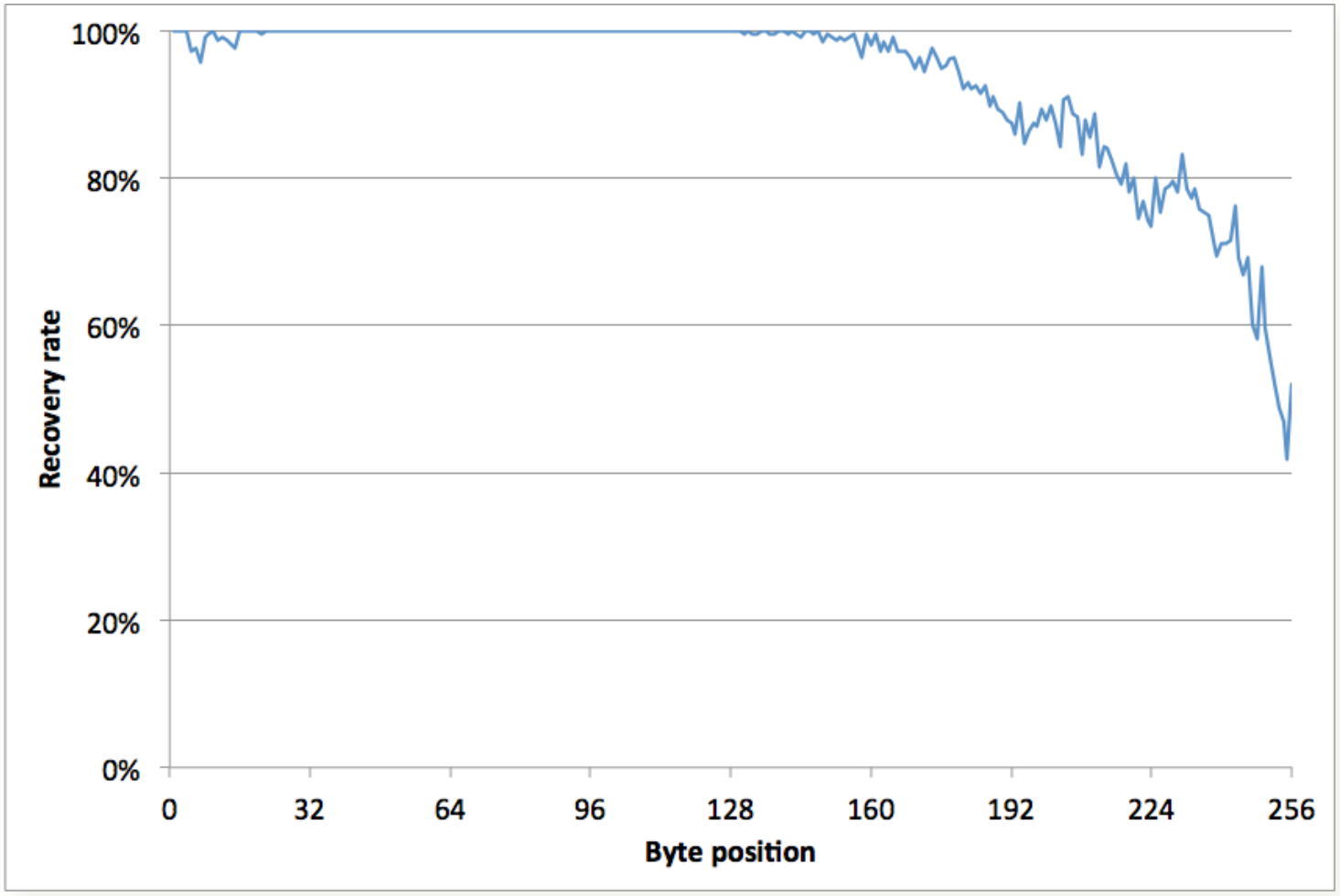
Success Probability 2^{28} Sessions



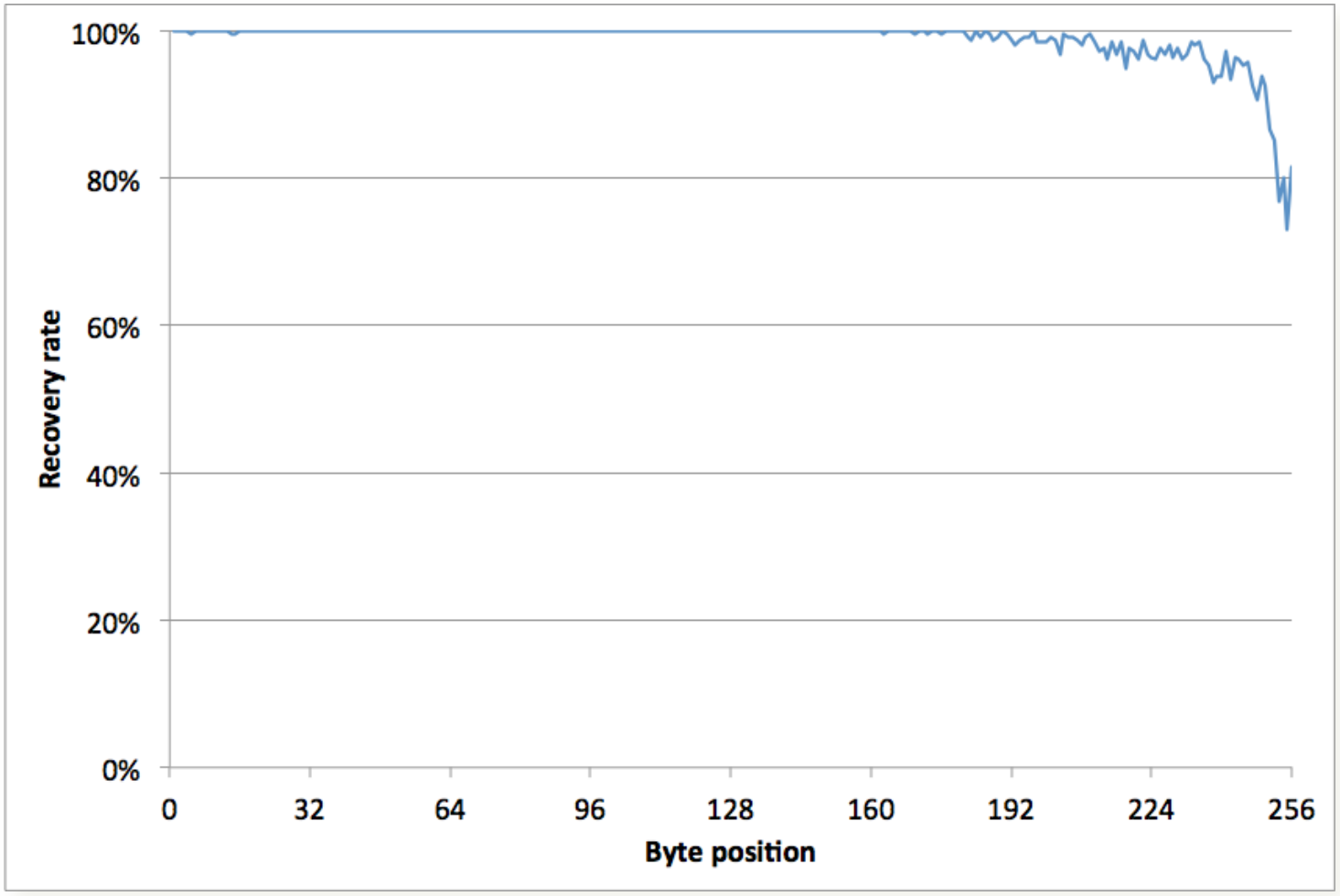
Success Probability 2^{29} Sessions



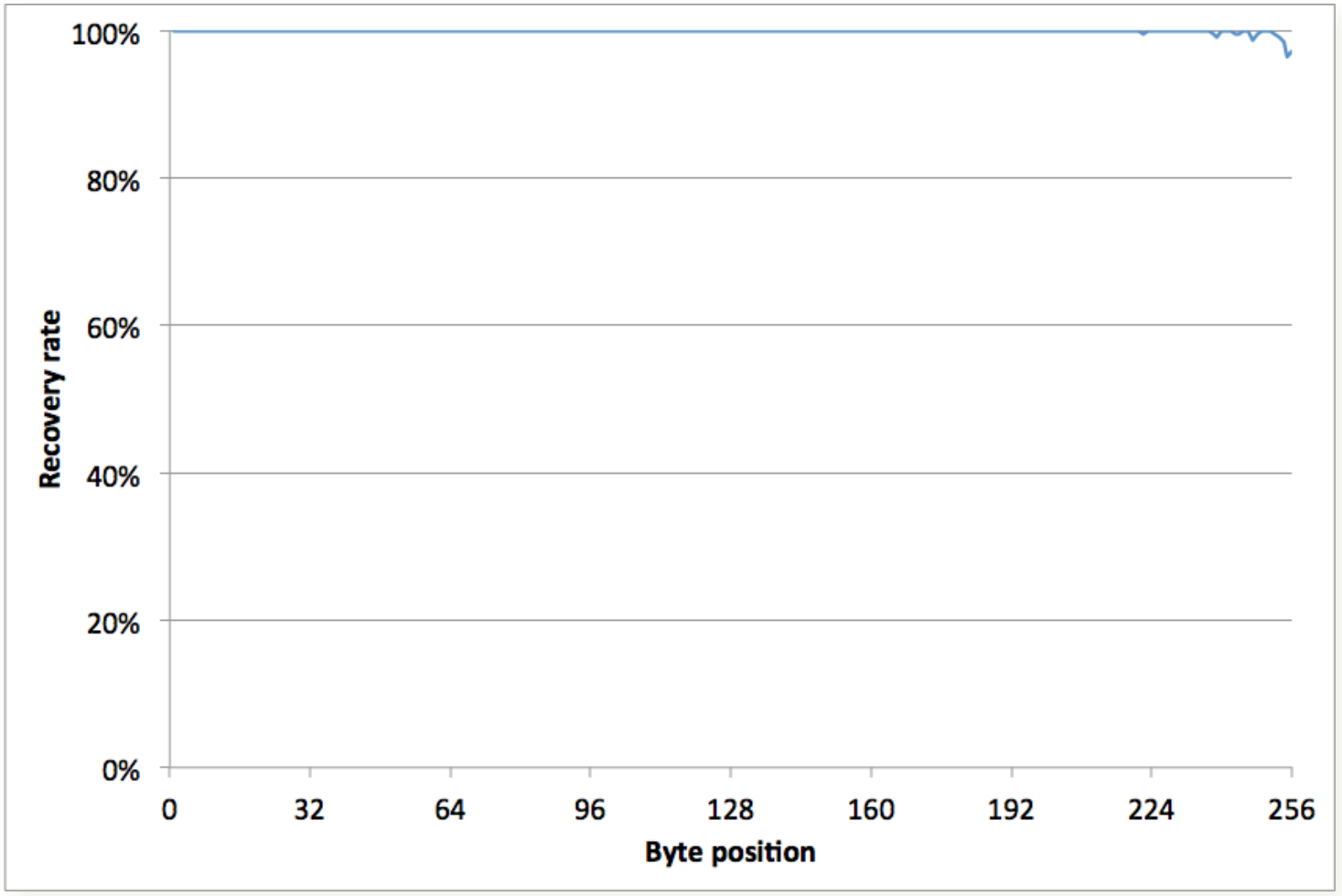
Success Probability 2^{30} Sessions



Success Probability 2^{31} Sessions



Success Probability 2^{32} Sessions



Preferred RC4 Attack

Double-byte bias attack, using Fluhrer-McGrew biases, (known since 2000).

Again using BEAST techniques to generate required plaintexts.

Enables attack beyond the first 256 bytes.

10×2^{30} encryptions for 98% success rate in cookie recovery.

Vendor Responses to RC4 Attack

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.

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

Current/future Developments in TLS

Current/Future Developments

CBC-mode ciphersuites can be patched against BEAST and Lucky 13 , but looking tired.

RC4 pretty much dead.

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

AES-GCM quite hard to implement securely.

Fresh algorithms are under active consideration in IETF TLS WG.

- Some momentum behind Salsa20/ChaCha20 stream ciphers plus Poly1305 MAC.
- But additional review needed.

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

- IETF draft exists and under review.

Current/Future Developments

TLS 1.3 now under active development in TLS WG.

- Simplification of key exchange and authentication methods in Handshake Protocol.
- Server name/identity hiding for improved privacy.
- Reducing latency.
- Reform of symmetric algorithms.

Active review of drafts needed by users and cryptographers.

TLS – Current Status?

"This is a dead parrot."

"He's not dead. He's just resting."

Thanks

Thanks to my co-author Nadhem AlFardan for working with me to make Lucky 13 a reality.

Thanks to Dan Bernstein, Bertram Poettering and Jacob Schuldt for collaboration on analysing RC4 in TLS.

Thanks to Stephen Farrell, IETF Security Area co-director, for the ANRP nomination.

Thank you for listening.