Authenticated Encryption in SSH

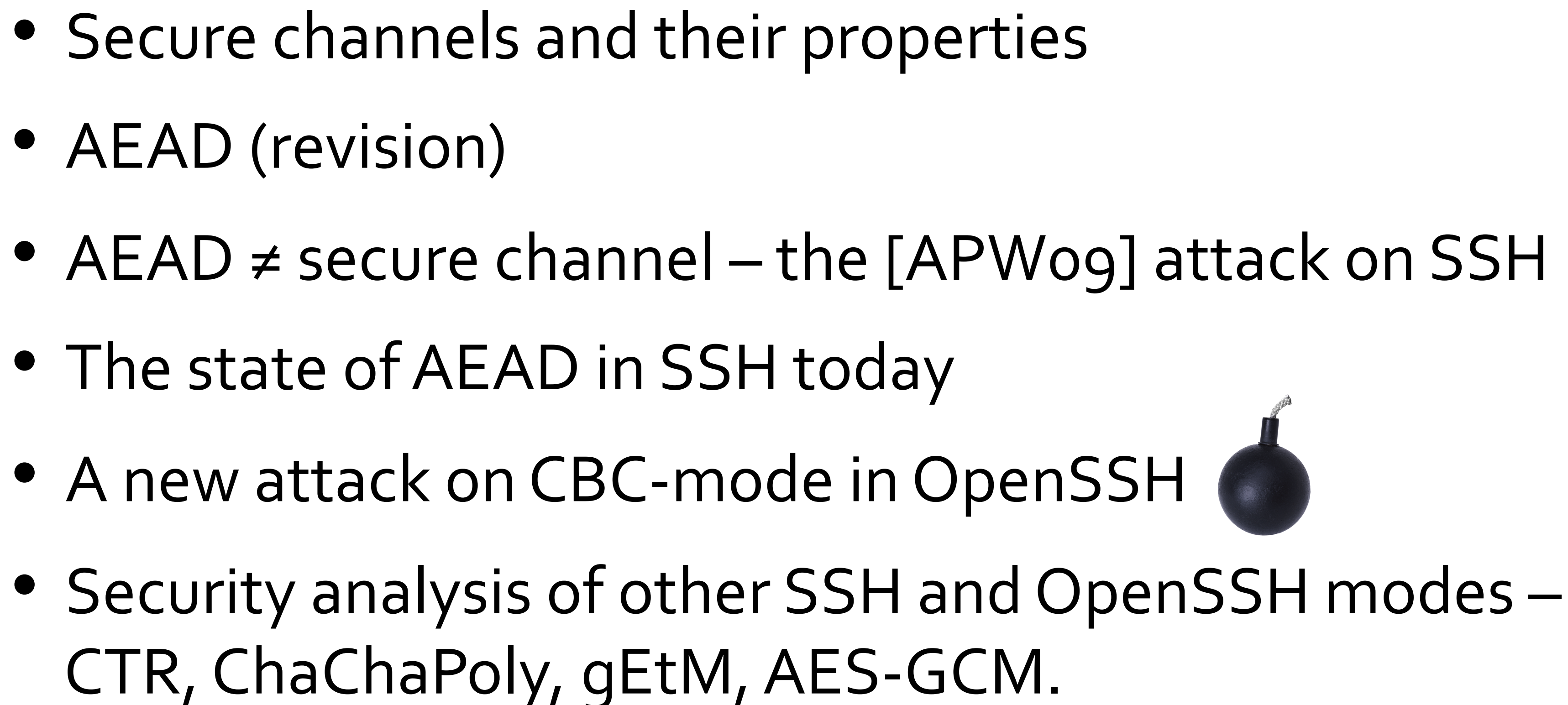
Kenny Paterson

Information Security Group

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

ROYAL HOLLOWAY UNIVERSITY OF LONDON

Overview (both lectures)



- Secure channels and their properties
- AEAD (revision)
- AEAD ≠ secure channel – the [APW09] attack on SSH
- The state of AEAD in SSH today
- A new attack on CBC-mode in OpenSSH
- Security analysis of other SSH and OpenSSH modes – CTR, ChaChaPoly, gEtM, AES-GCM.

Secure channels and their properties

Why do we need secure channels?

- Secure communications **is** the most common real-world application of cryptography today.
- Secure communications systems are extremely widely-deployed in practice:
    - SSL/TLS, DTLS, IPsec, SSH, OpenVPN,…
    - WEP/WPA/WPA2
    - GSM/UMTS/4g/LTE
    - Cryptocat, OTR, SilentCircle
    - OpenPGP, iMessage, Telegram, Signal, and a thousand other messaging apps
    - QUIC, MinimalT, TCPcrypt

Security properties

- **Confidentiality** – privacy for data
- **Integrity** – detection of data modification
- **Authenticity** – assurance concerning the source of data

Some less obvious security properties

- **Anti-replay**
  - Detection that messages have been repeated.
- **Detection of deletion**
  - Detection that messages have been deleted by the adversary or dropped by the network.
- **Detection of re-ordering**
  - Ensuring that the relative order of messages in *each* direction on the secure channel is preserved.
  - Possibly re-ordering the event of violation.
- **Prevention of traffic-analysis.**
  - Using traffic padding and length-hiding techniques.

Possible functionality requirements

- **Speedy**
- **Low-memory**
- **On-line/parallelisable crypto-operations**
  - Performance is heavily hardware-dependent.
  - May have different algorithms for different platforms.
- **IPR-friendly**
  - This issue has slowed down adoption of many otherwise good algorithms, e.g. OCB.
- **Easy to implement**
  - Without introducing any side-channels.

Additional requirements

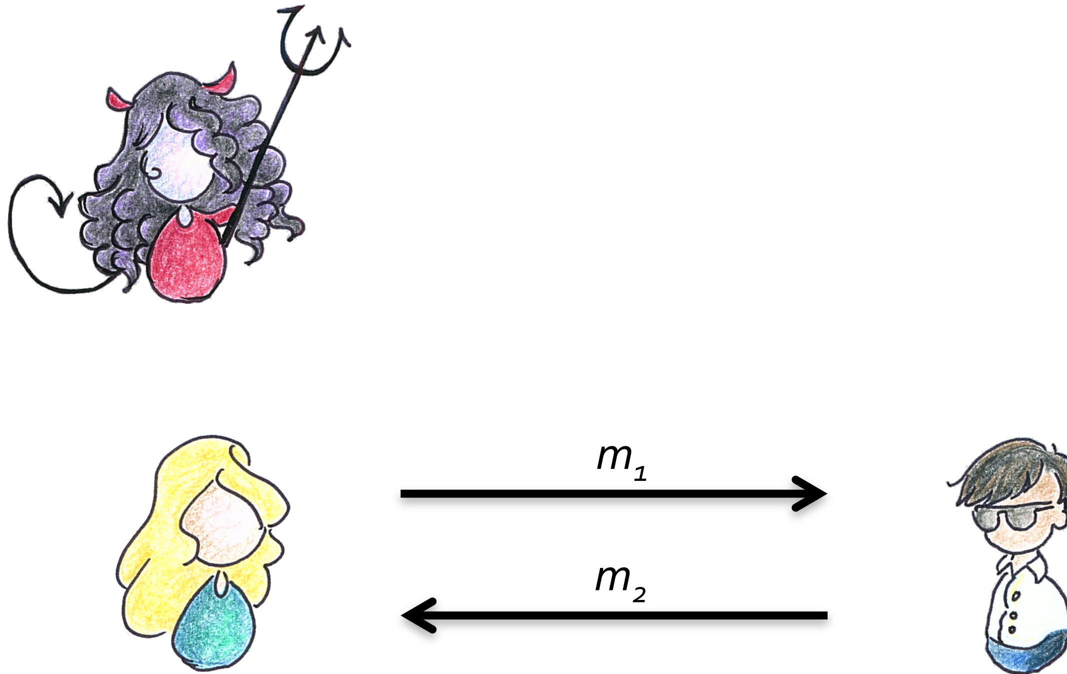
- We need a clean and well-defined API.
- Because the reality is that our secure channel protocol will probably be used blindly by a security-naïve developer.
- Developers want to "open" and "close" secure channels, and issue "send" and "recv" commands.
- They'd like to simply replace TCP with a "secure TCP" having the same API.
- Or to just have a black-box for delivering messages securely.

Additional API-driven requirements

- Does the channel provide a stream-based functionality or a message-oriented functionality?
- Does the channel accept messages of arbitrary length and perform its own fragmentation and reassembly, or is there a maximum message length?
- How is error handling performed? Is a single error fatal, leading to tear-down of channel, or is the channel tolerant of errors?
- How are these errors signalled to the calling application? How should the programmer handle them?
- Does the secure channel itself handle retransmissions? Or is this left to the application? Or is it guaranteed by the underlying network transport?
- Does the channel offer data compression?
- **These are design choices that all impact on security**
- **They are not well-reflected in the security definitions for symmetric encryption**

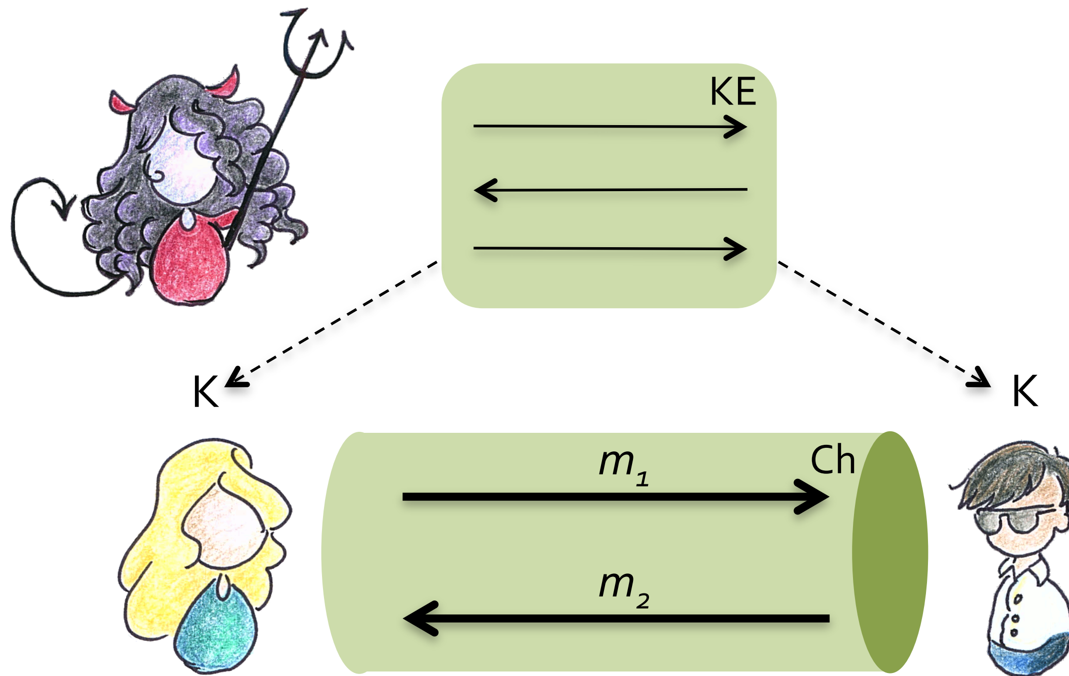
AEAD (Revision)

Security for Symmetric Encryption



Pictures by Giorgia Azzurra Marson

Security for Symmetric Encryption

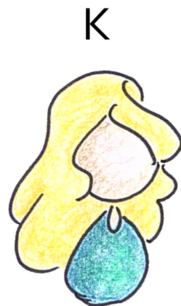


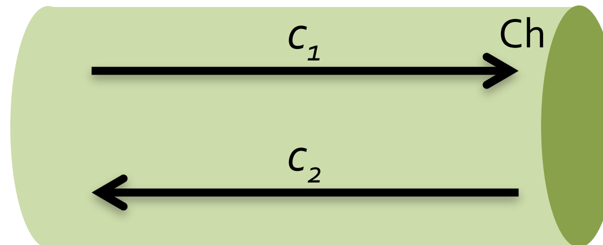
Security for Symmetric Encryption

$c_1 = \text{Enc}_K(m_1)$

$m_2 = \text{Dec}_K(c_2)$

K



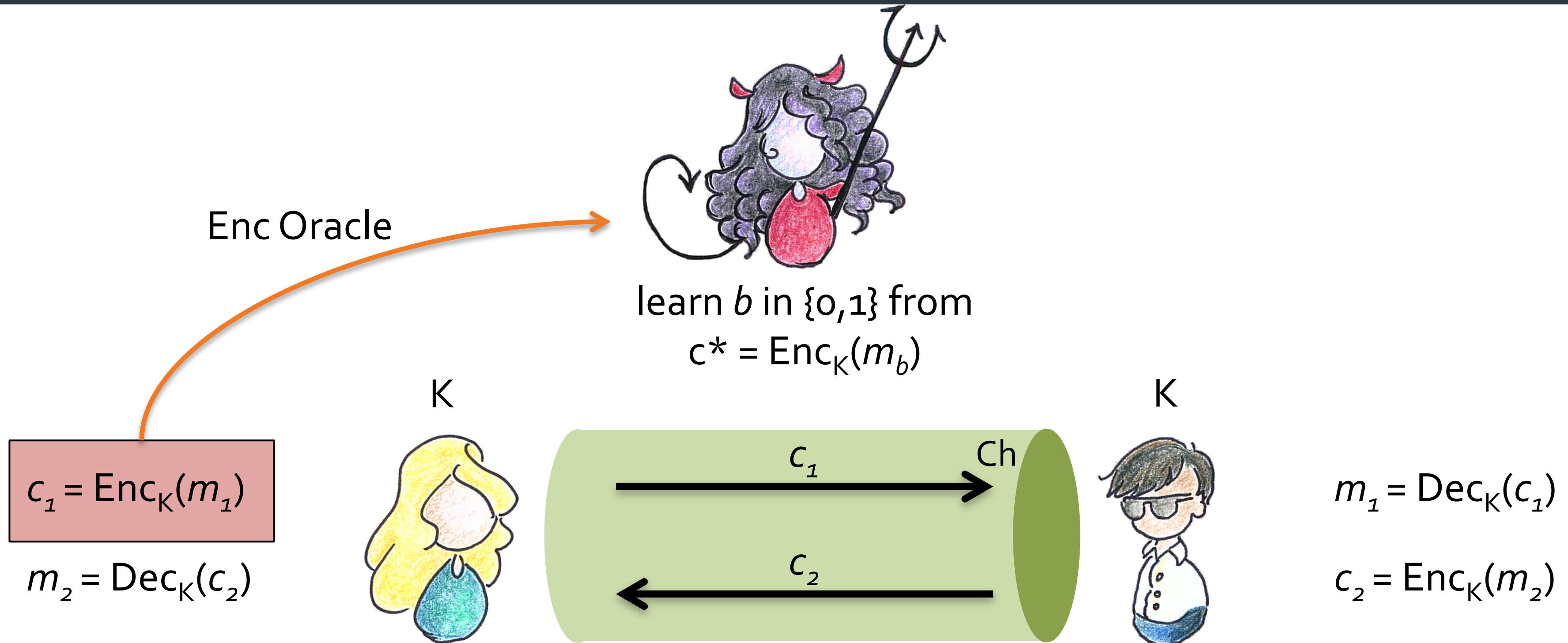


K

$m_1 = \text{Dec}_K(c_1)$

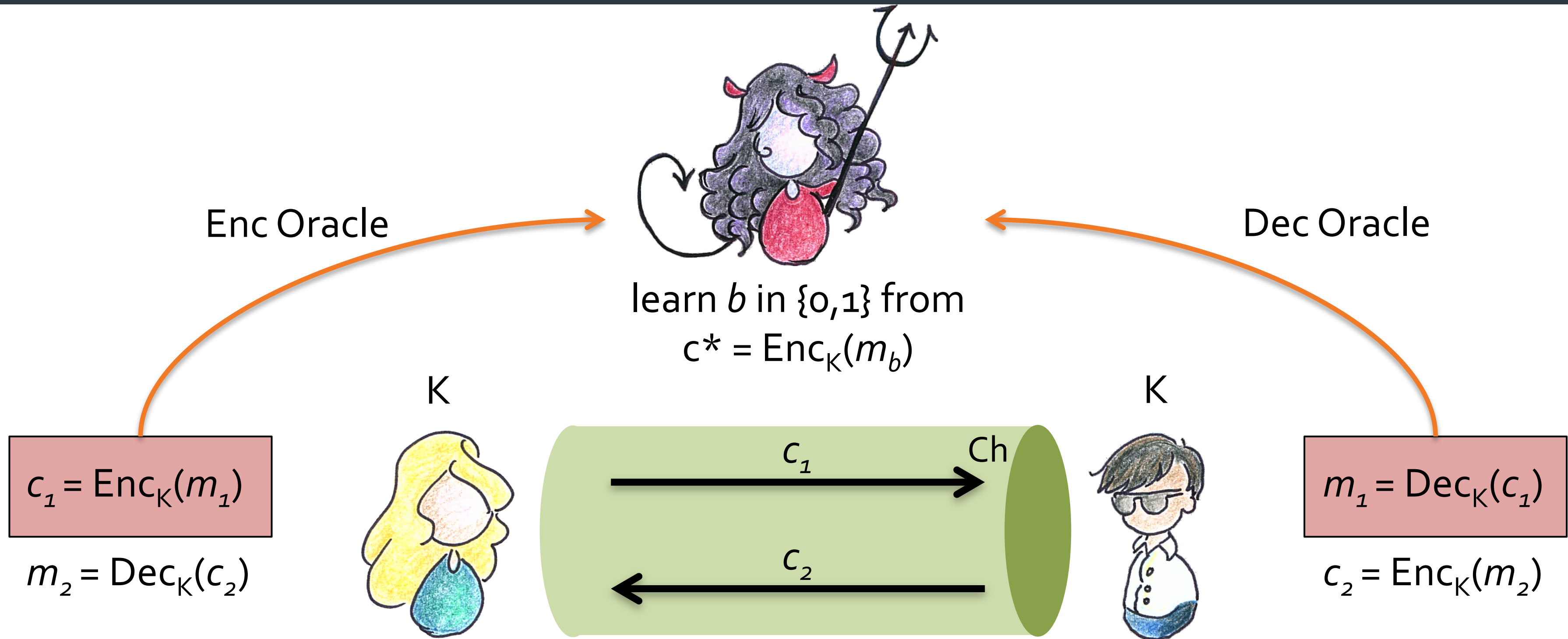
$c_2 = \text{Enc}_K(m_2)$

Security for Symmetric Encryption – Confidentiality



IND-CPA
(Goldwasser-Micali, 1984;
Bellare-Desai-Jokipii-Rogaway, 1997).

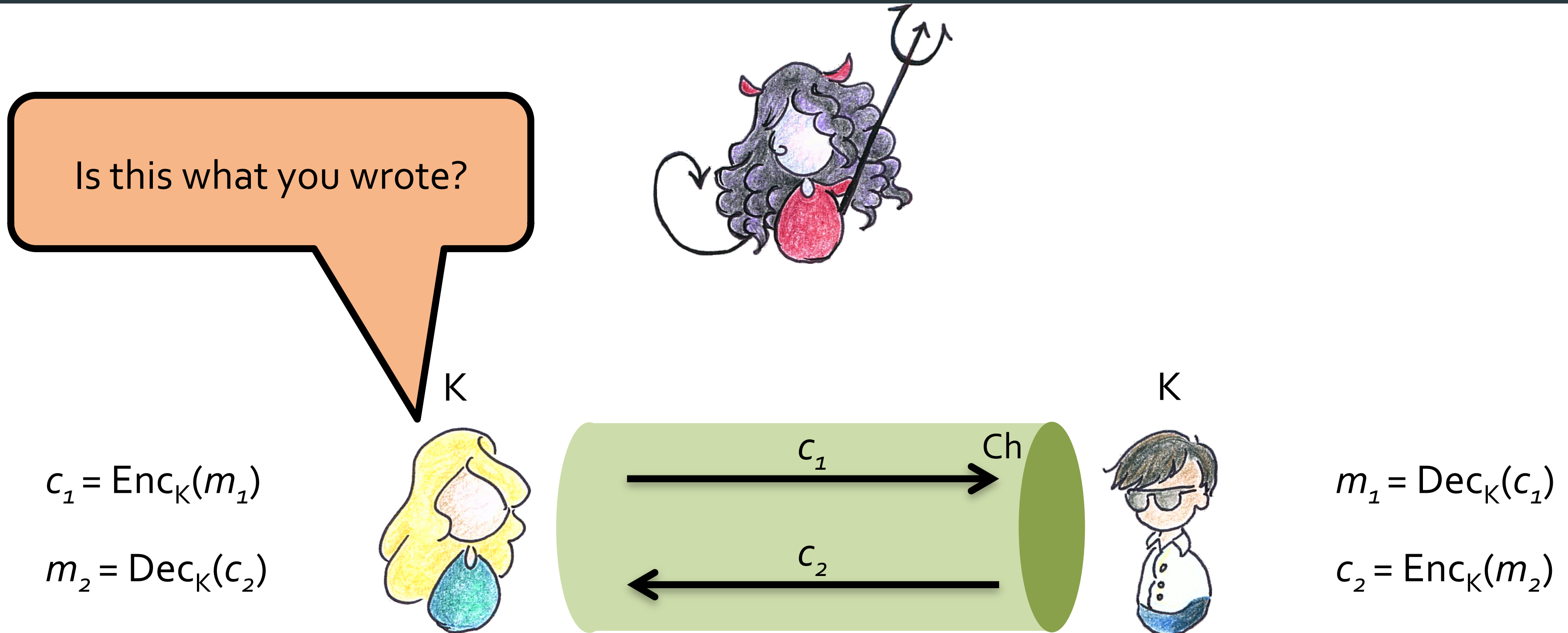
Security for Symmetric Encryption – Confidentiality



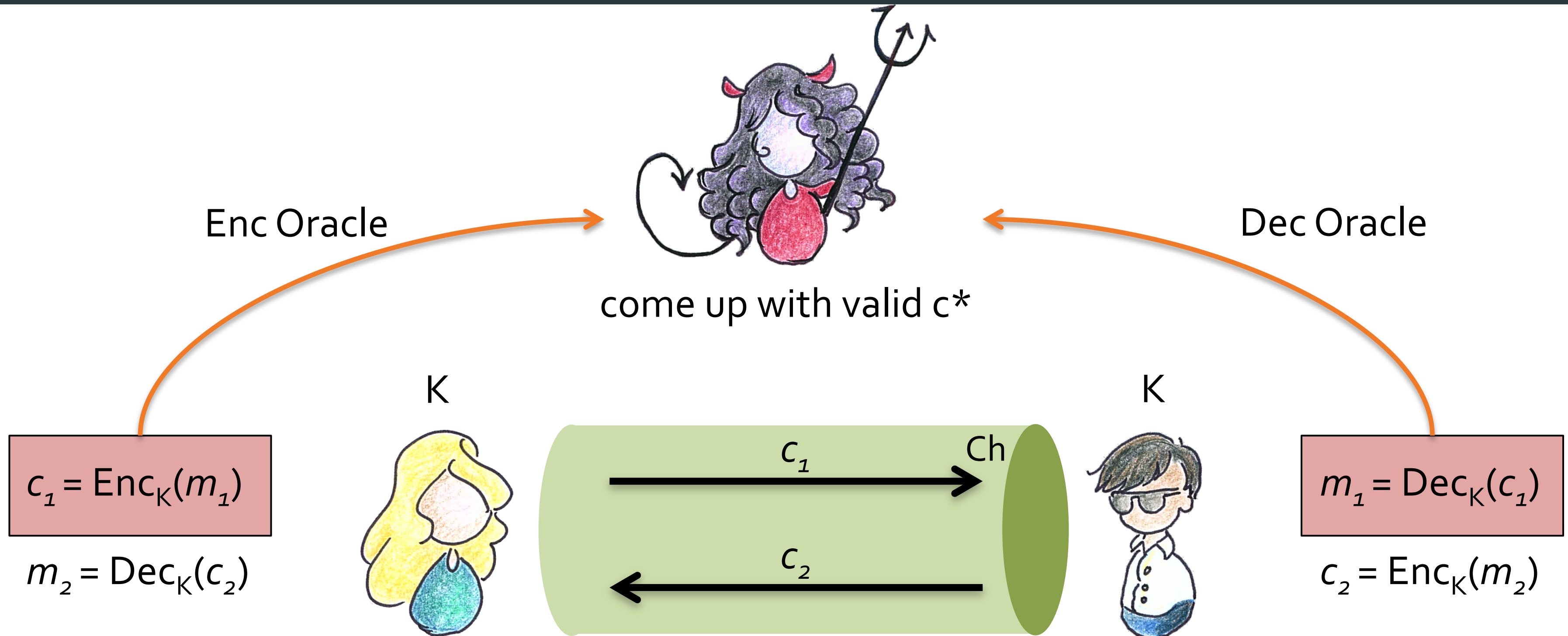
IND-CPA
(Goldwasser-Micali, 1984;
Bellare-Desai-Jokipii-Rogaway, 1997).

IND-CCA
(Naor-Yung, 1990;
Rackoff-Simon, 1997).

Security for Symmetric Encryption – Integrity

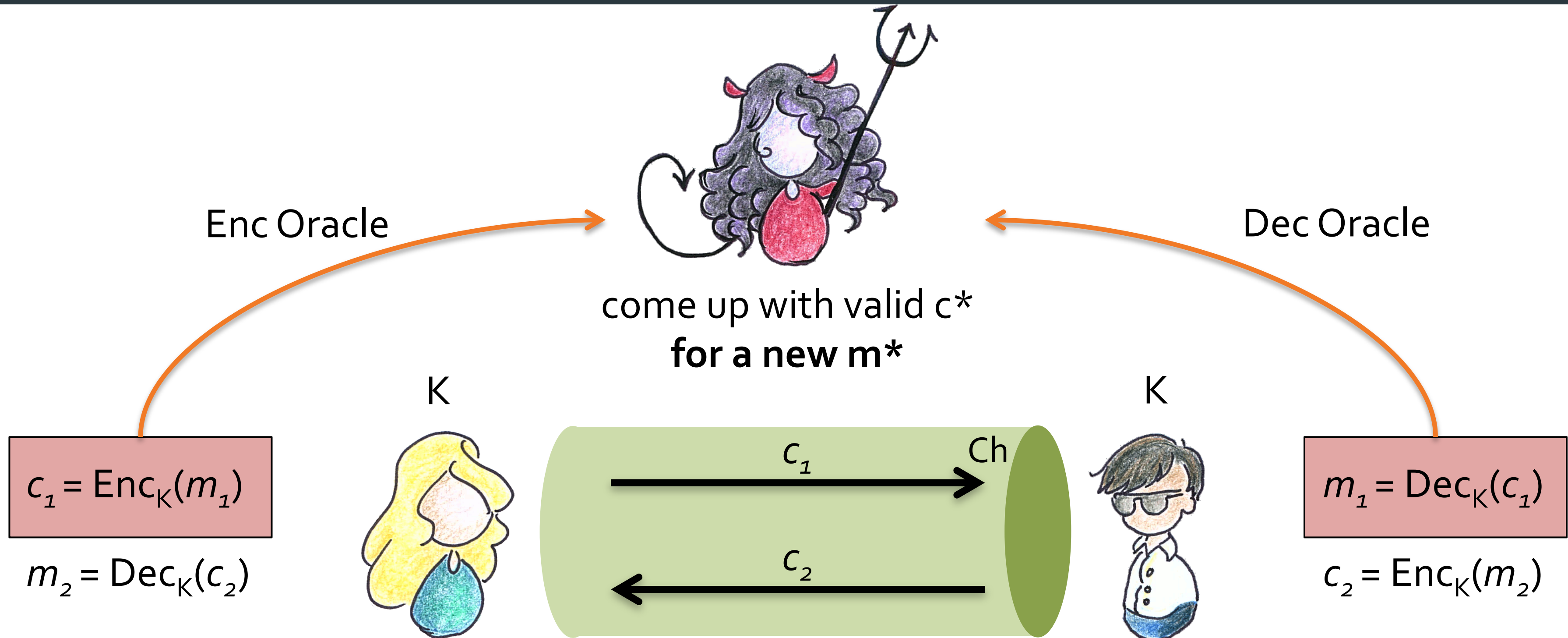


Security for Symmetric Encryption – Integrity



INT-CTXT
(Bellare, Rogaway, 2000)

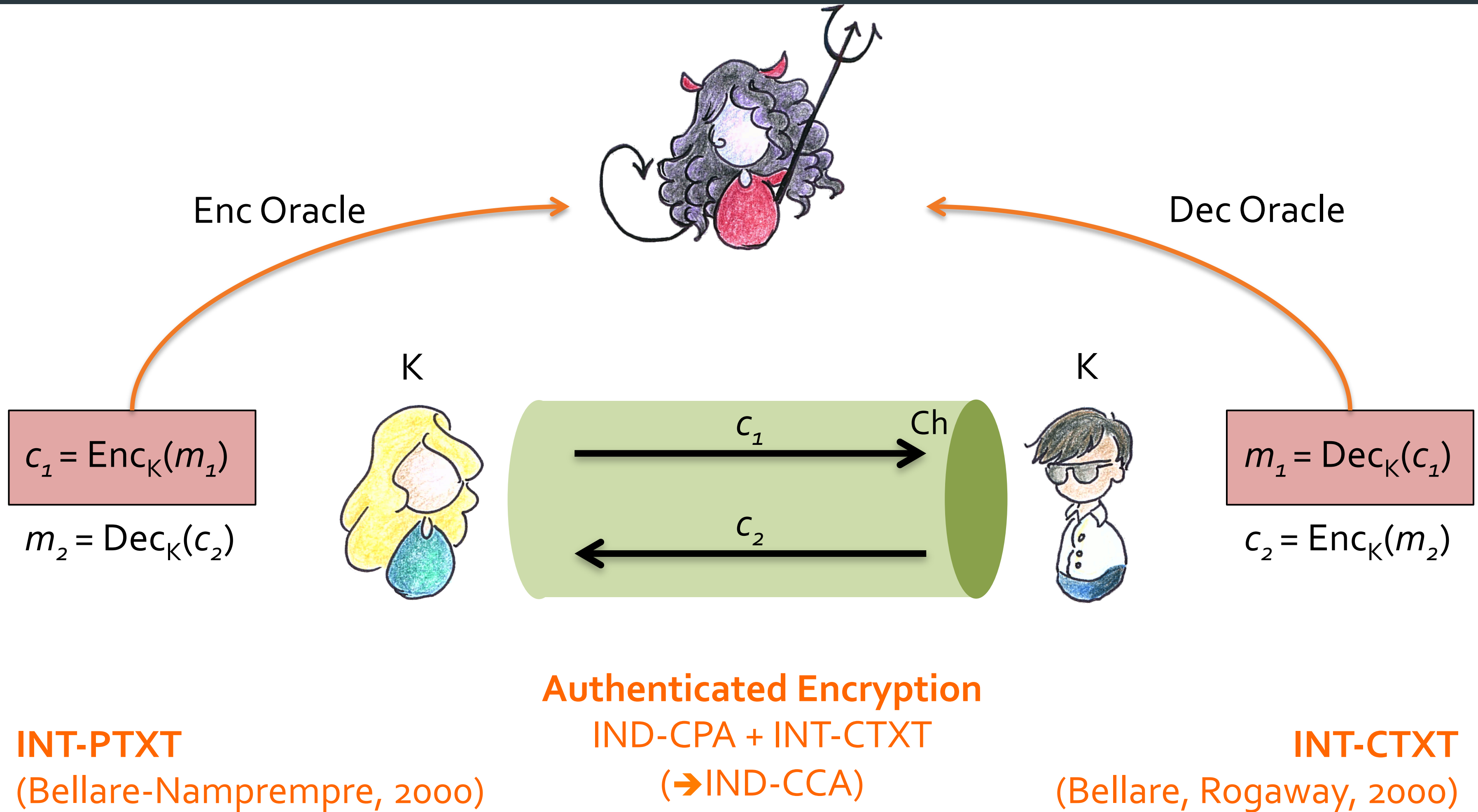
Security for Symmetric Encryption – Integrity



INT-PTXT
(Bellare-Namprempre, 2000)

INT-CTXT
(Bellare, Rogaway, 2000)

Security for Symmetric Encryption – AE

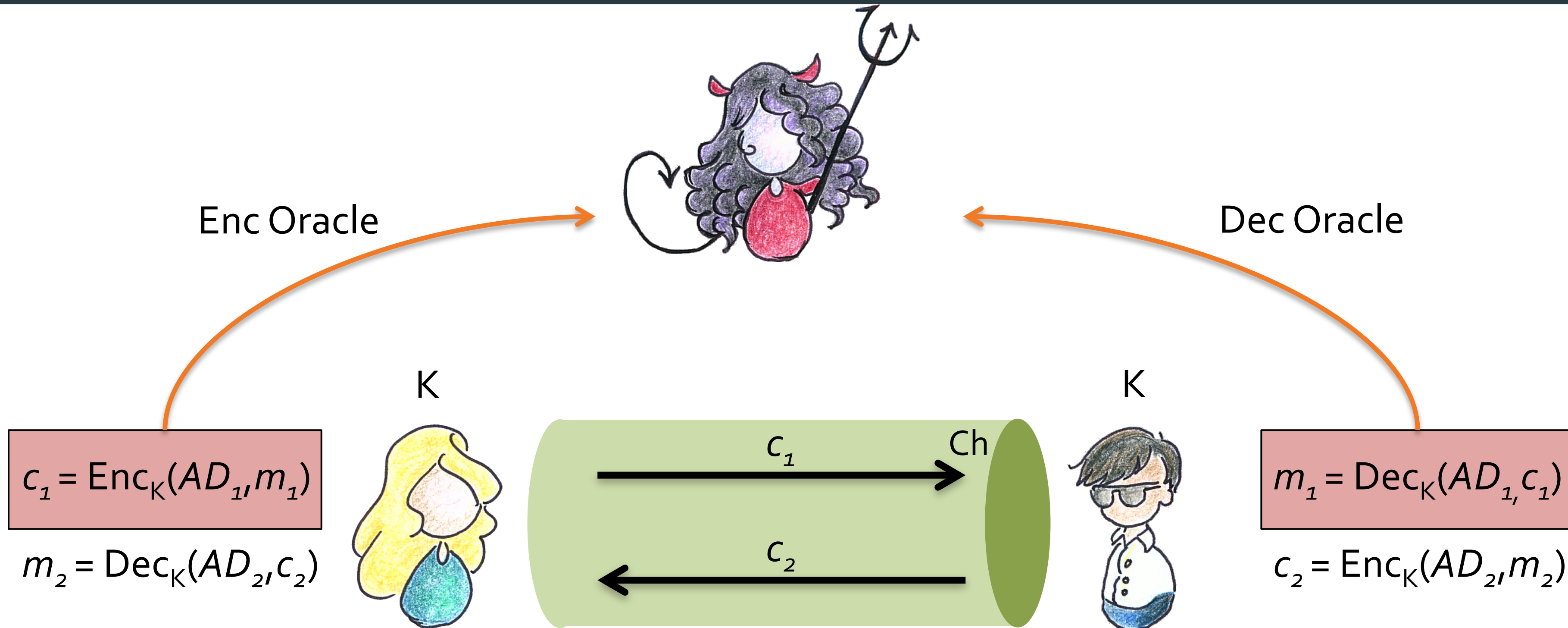


Authenticated Encryption
IND-CPA + INT-CTXT
(→IND-CCA)

INT-PTXT
(Bellare-Namprempre, 2000)

INT-CTXT
(Bellare, Rogaway, 2000)

Security for Symmetric Encryption – AEAD



Authenticated Encryption with Associated Data

AE security for message *m*

Integrity for associated data *AD*

Strong binding between *c* and *AD*

(Rogaway 2002)

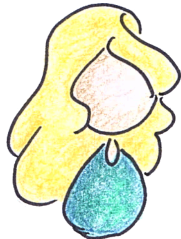
Security for Symmetric Encryption – stateful AEAD

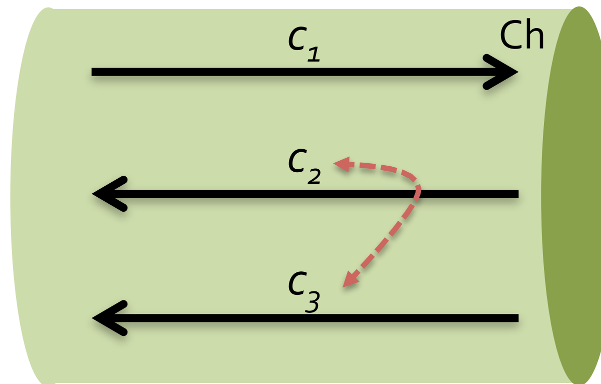
Which came first?

$c_1 = \mathrm{Enc}_K(AD_1, m_1)$

$m_2 = \mathrm{Dec}_K(AD_2, c_2)$

$m_3 = \mathrm{Dec}_K(AD_3, c_3)$





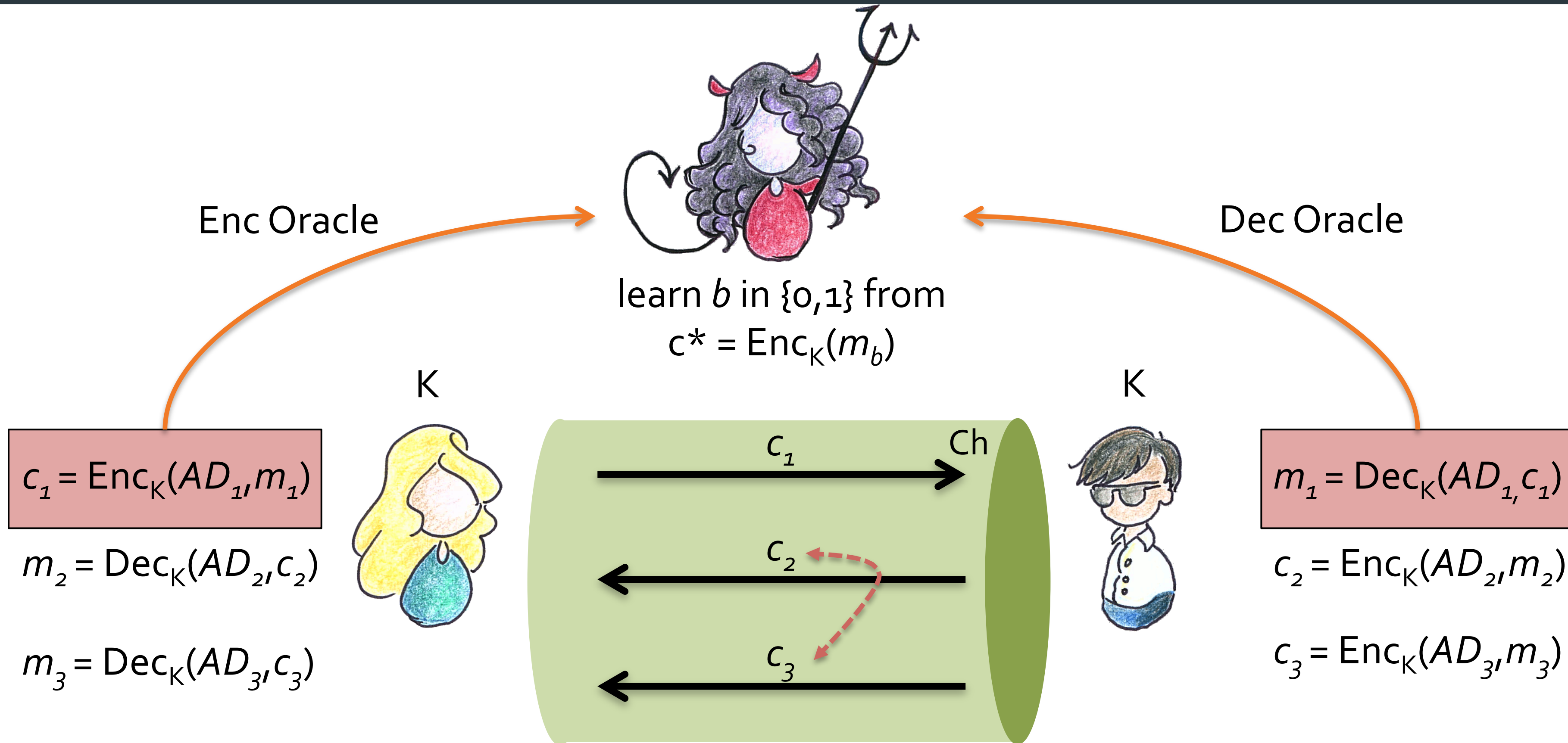
K

$m_1 = \mathrm{Dec}_K(AD_1, c_1)$

$c_2 = \mathrm{Enc}_K(AD_2, m_2)$

$c_3 = \mathrm{Enc}_K(AD_3, m_3)$

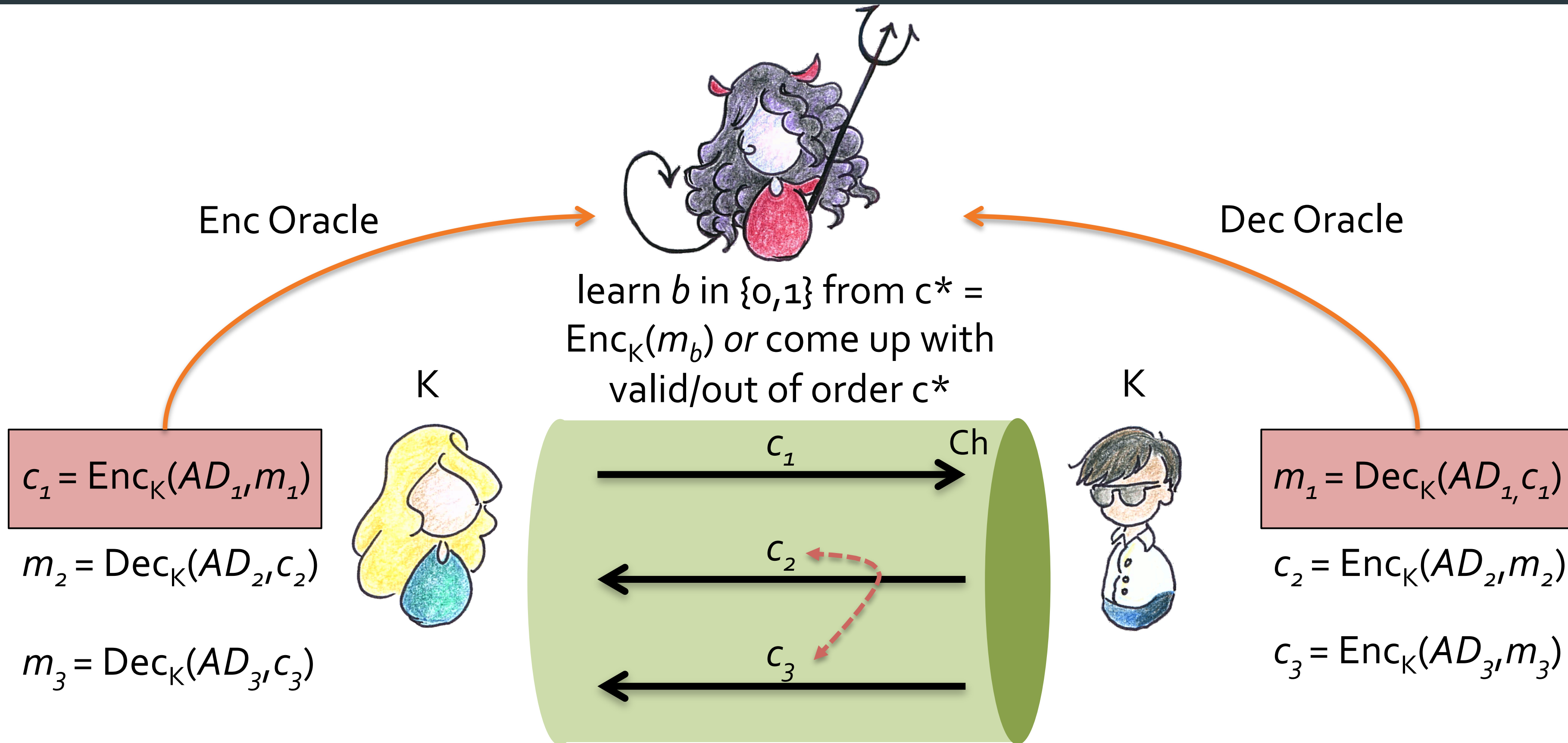
Security for Symmetric Encryption – stateful AEAD



IND-sfCCA

(Bellare-Kohno-Namprempre, 2002)

Security for Symmetric Encryption – stateful AEAD



Stateful AEAD

IND-sfCCA

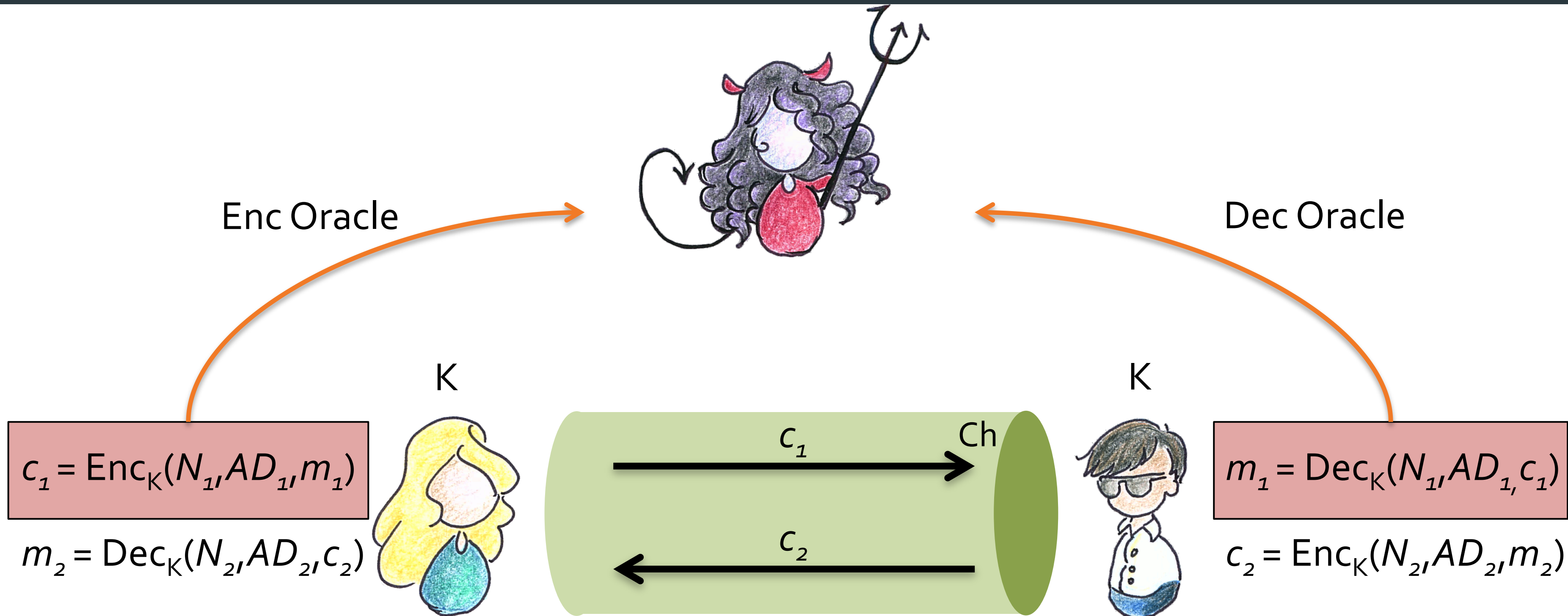
INT-sfCTXT

(Bellare-Kohno-Namprempre, 2002)

INT-sfPTXT

(Brzuska-Smart-Warinschi-Watson, 2013)

Security for Symmetric Encryption – nonce-based AEAD



Nonce-based Authenticated Encryption with Associated Data

As per AEAD, but with additional input N to Enc and Dec algorithms

Adversary may arbitrarily specify N, but "no repeats" rule

Enc and Dec can now be *stateless* and *deterministic*

(Rogaway 2004)

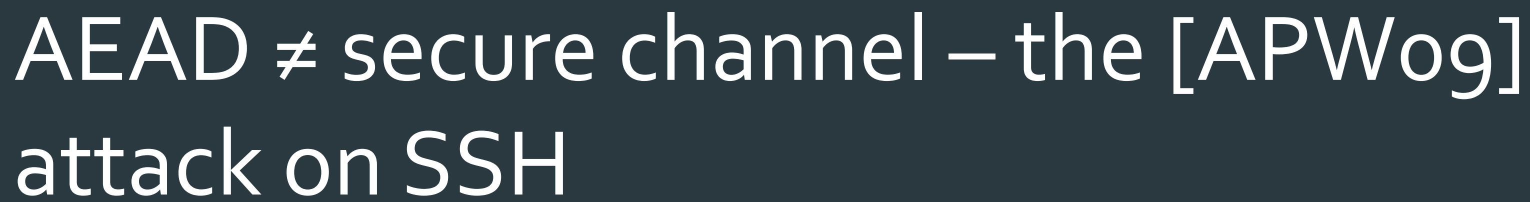
Security for Symmetric Encryption – further notions

- LH-(stateful)AE(AD)
  - On top of everything else, ciphertexts provide a modicum of hiding of plaintext lengths.
  - cf variable length padding in SSL/TLS.
  - Introduced by Paterson-Ristenpart-Shrimpton, 2011.
  - Incorporated into ACCE framework for analysis of TLS by Jager-Kohlar-Schage-Schwenk, 2012.

CAESAR

- CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness.
- Initiated by Dan Bernstein, supported by committee of experts.
- Main goal is the design of a *portfolio* of **AE schemes.**
- CAESAR has involved dozens of person-years of effort and led to a major uptick in research activity.

- **It seems that most of the cryptographic community has settled on nonce-based AE/AEAD as their working abstraction.**


AEAD ≠ secure channel – the [APW09]
attack on SSH

AEAD ≠ secure channel

- Recall our application developer:
  - He wants a drop-in replacement for TCP that's secure.
  - Actually, he might *just* want to send and receive some atomic messages and not a TCP-like stream.

- To what extent does AEAD meet these requirements?

- It doesn't…

AEAD ≠ secure channel

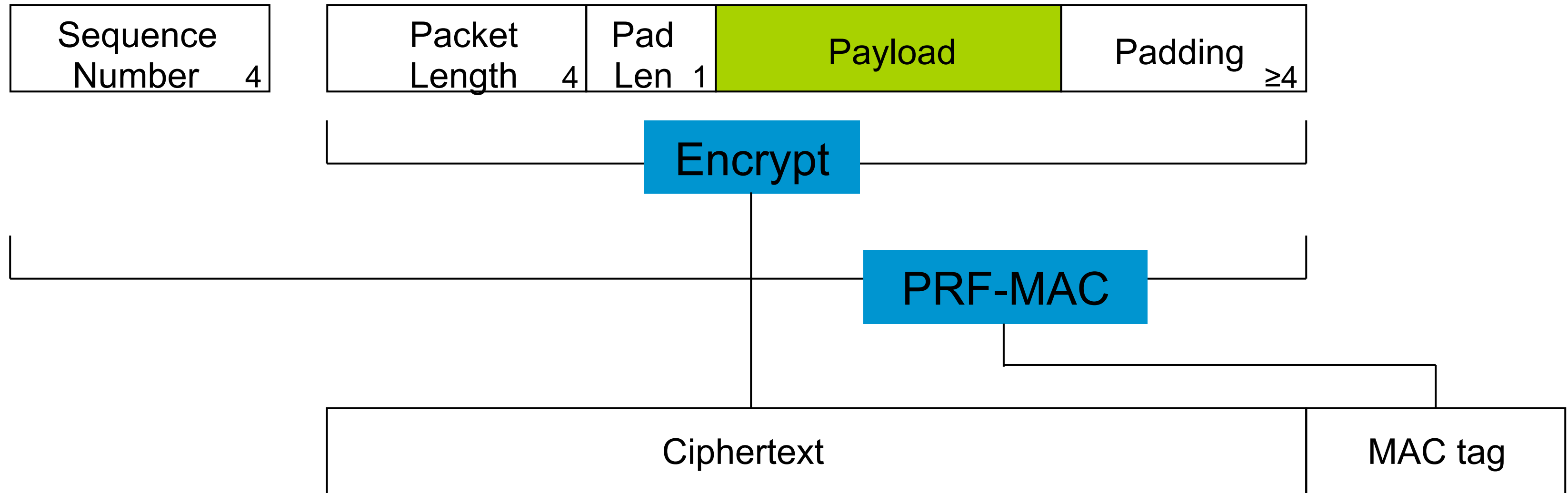
There's a significant semantic gap between AEAD's functionality and raw security guarantees, and all the things a developer expects a secure channel to provide.

Introduction to SSH

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

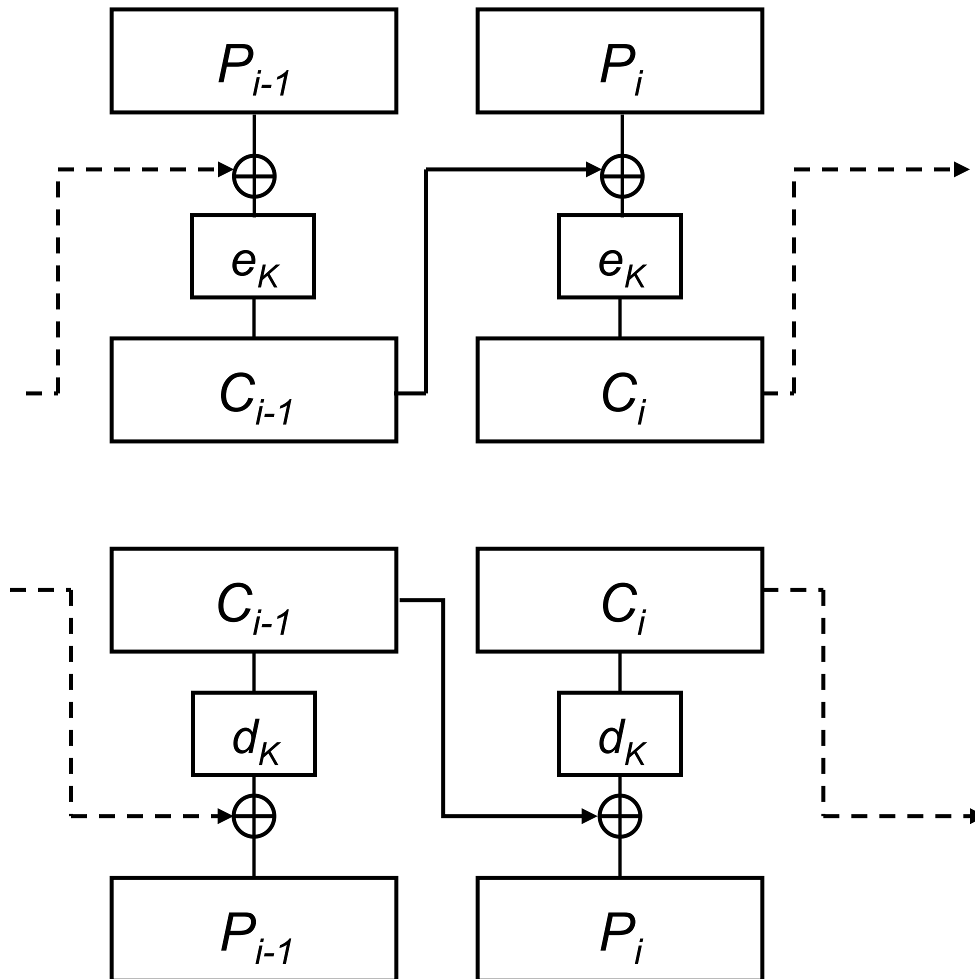
– Wikipedia

SSH Binary Packet Protocol (RFC 4253)



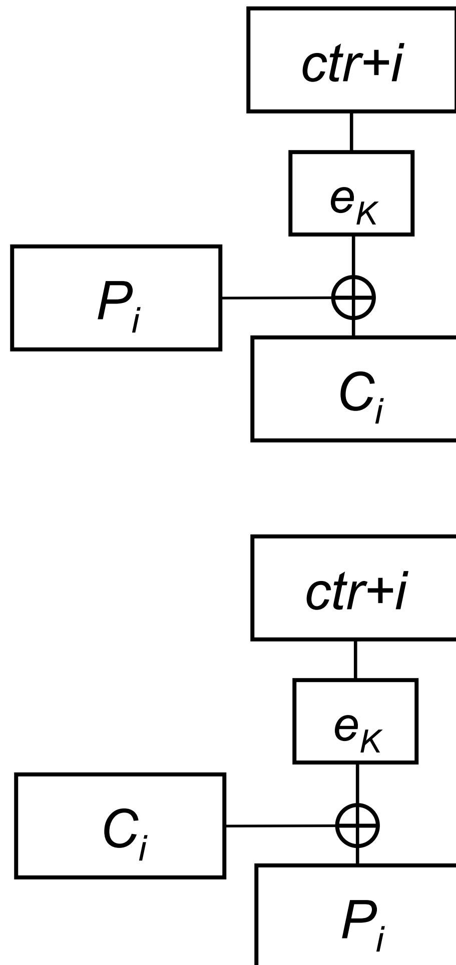
- Encode-then-E&M construction, stateful because of inclusion of 4-byte sequence number.
- Packet length field measures the size of the packet: |PadLen|+ |Payload| + |Padding|.
  - Encrypted, so sequence of encrypted packets looks like a long string of random bytes.
- Encryption options in RFC 4253: CBC mode; RC4.
- AES-CTR defined in RFC 4344.

CBC mode in SSH



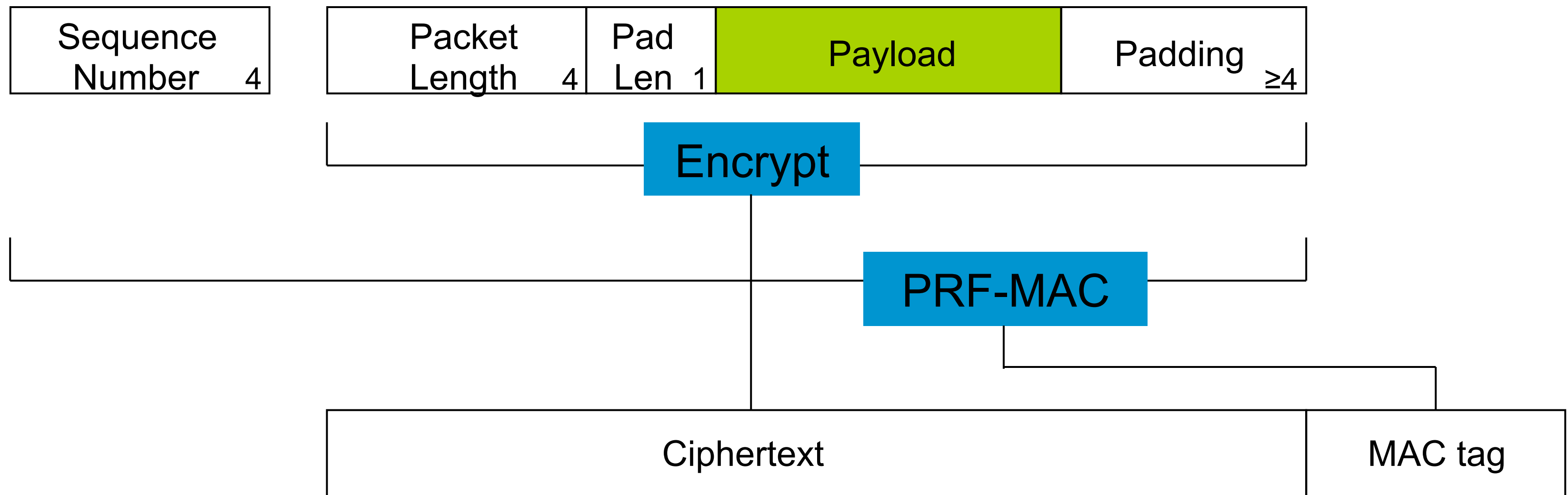
- RFC 4253 mandates 3DES-CBC and recommends AES-CBC.
- SSH uses a chained IV in CBC mode:
  - IV for current packet is the last ciphertext block from the previous packet.
  - Effectively creates a single stream of data from multiple SSH packets.

CTR mode in SSH



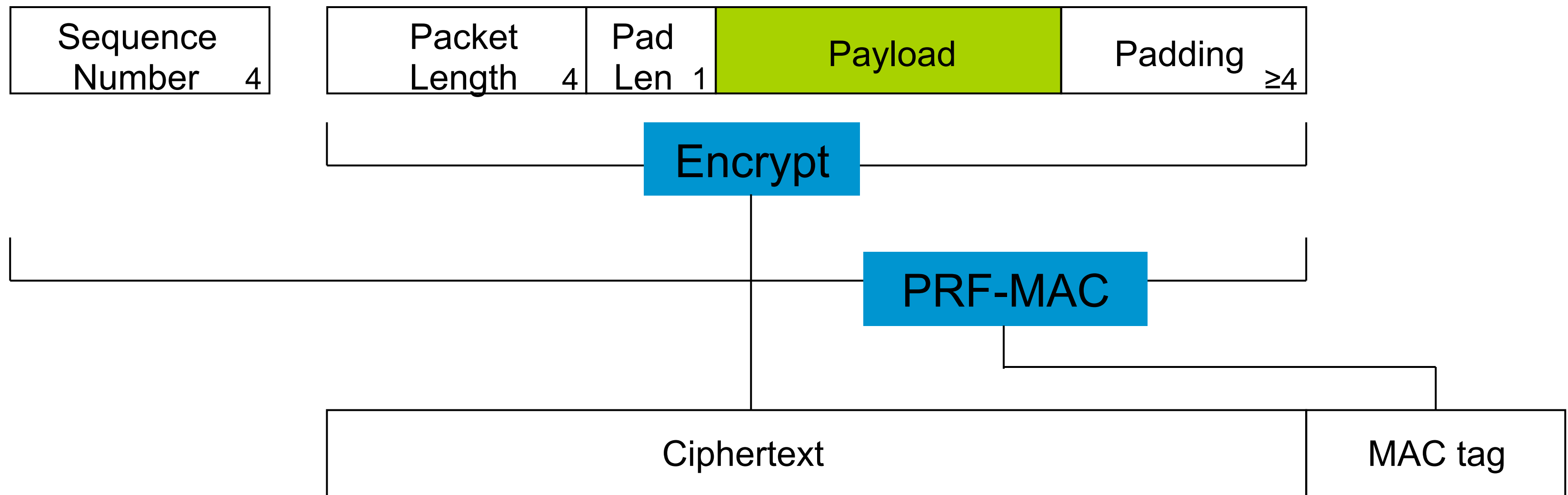
- CTR mode uses block cipher to build a stream cipher.
- CTR mode for SSH is standardised in RFC 4344.
  - Initial value of counter is obtained from handshake protocol.
  - Counter runs across packets.
  - Packet format is preserved from CBC case.
  - RFC recommends use of AES-CTR with 128, 192 and 256-bit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

SSH Binary Packet Protocol (RFC 4253)



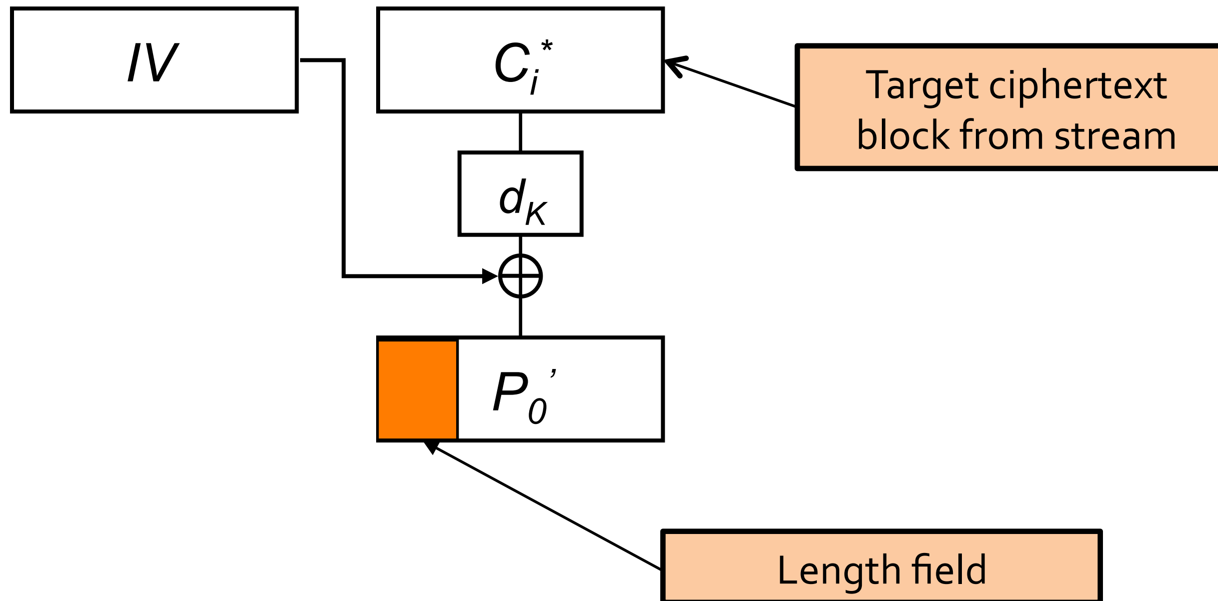
- IV chaining attack:
  - Chosen plaintext, distinguishing attack due to Rogaway, applied to SSH in [BKN02].
  - Not fully realistic for SSH because of format requirements on the first block of plaintext and because of chosen plaintext requirement.
- Construction with random IVs **is** IND-sfCCA secure [BKN02].

SSH Binary Packet Protocol (RFC 4253)



- How does decryption work?
- Recall: receiver gets a stream of bytes, and a single ciphertext can be fragmented over several TCP messages.

Breaking CBC mode in SSH [APW09]

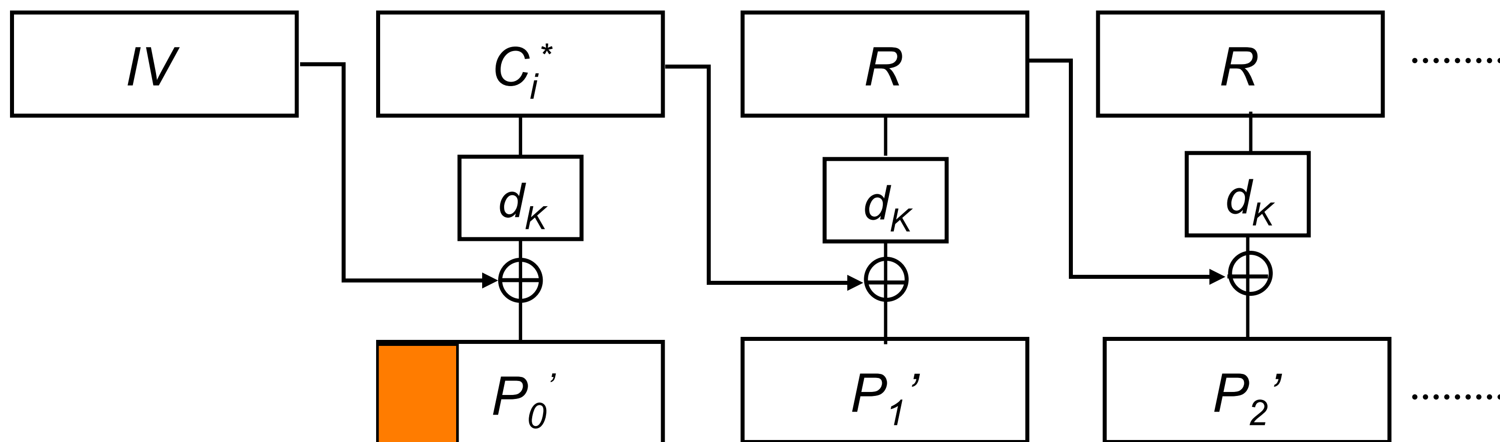


- The receiver will treat the first 32 bits of the calculated plaintext block as the packet length field for the new packet.
- Here:

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

where *IV* is known.

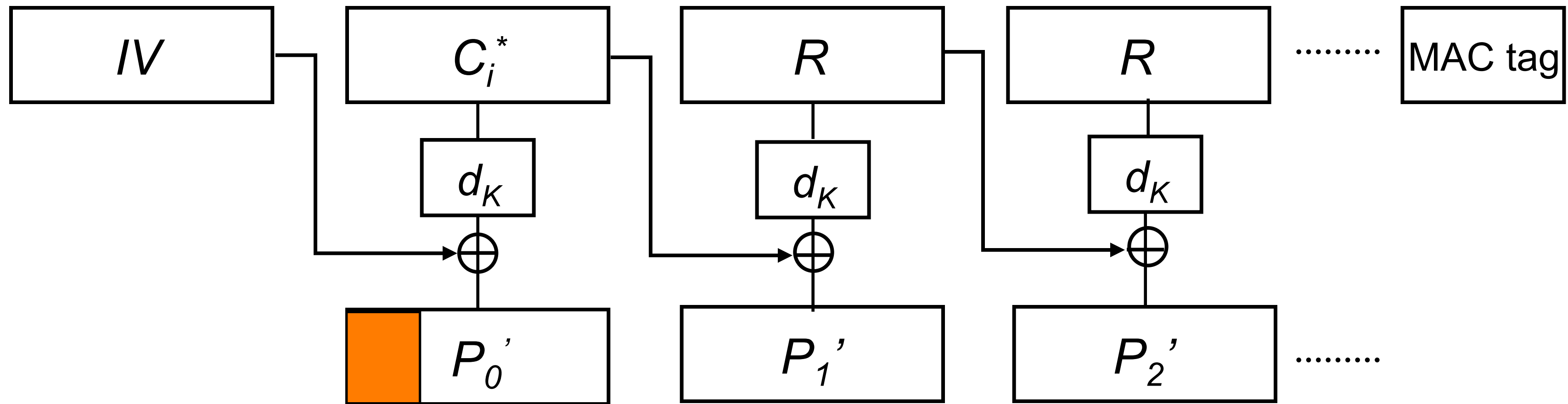
Breaking CBC mode in SSH [APW09]

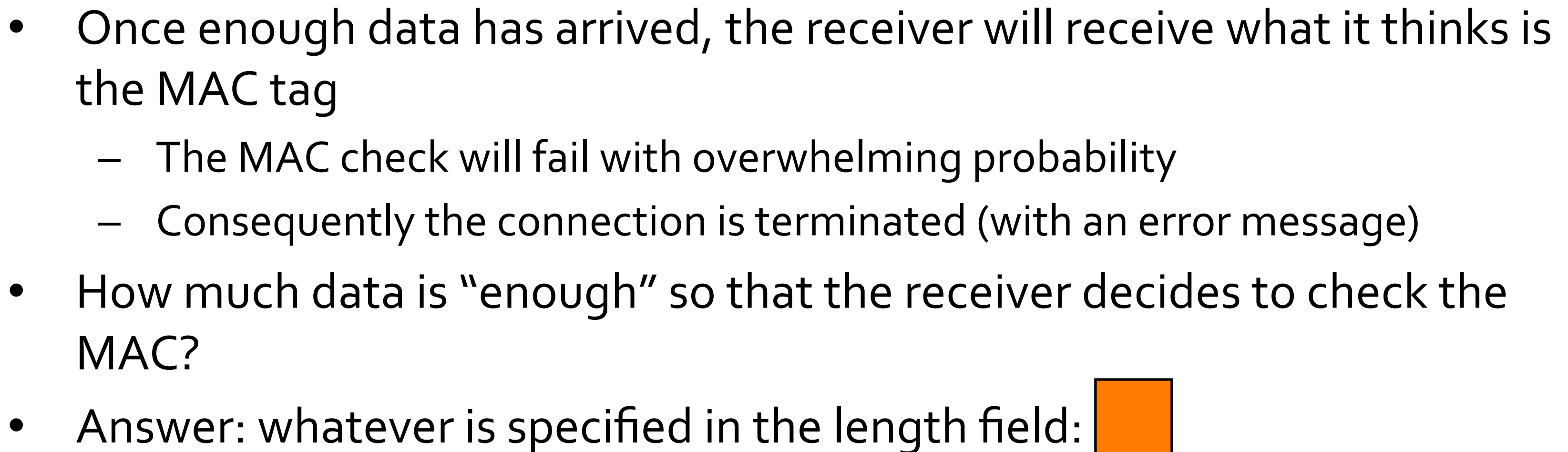


The attacker then feeds random blocks to the receiver

- One block at a time, waiting to see what happens at the server when each new block is processed
- This is possible because SSH runs over TCP and tries to do online processing of incoming blocks

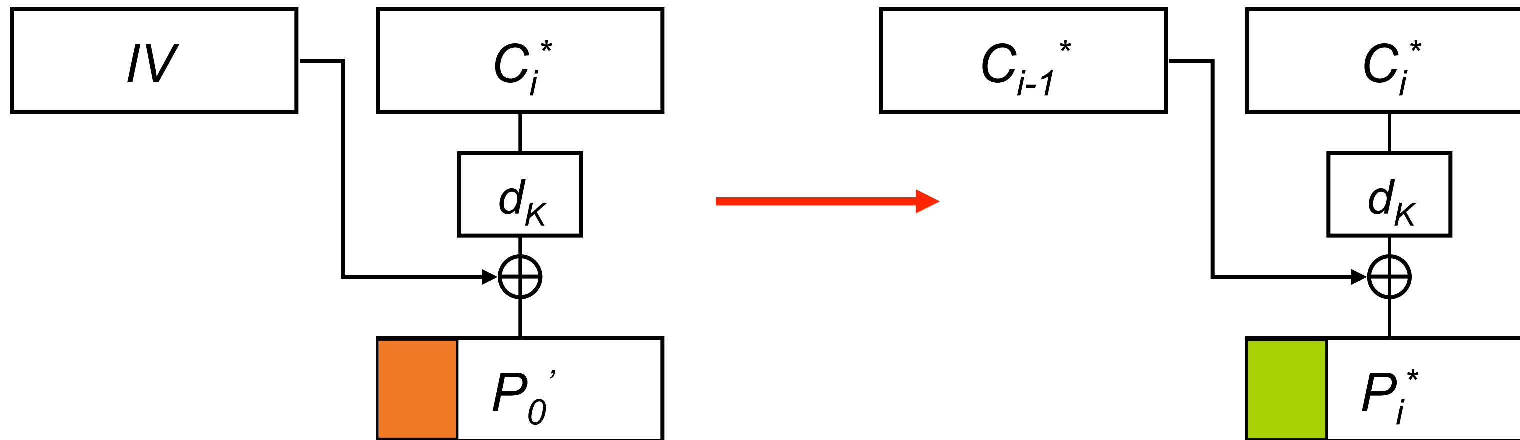
Breaking CBC mode in SSH [APW09]





- Once enough data has arrived, the receiver will receive what it thinks is the MAC tag
  - The MAC check will fail with overwhelming probability
  - Consequently the connection is terminated (with an error message)
- How much data is "enough" so that the receiver decides to check the MAC?
- Answer: whatever is specified in the length field:

Breaking CBC mode in SSH [APW09]



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

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

Further details

- The attack as described requires the injection of *circa* 2^{31} bytes of ciphertext (expected value of length field).
- It recovers 32 bits of plaintext with probability 1.
- And leads to an SSH connection teardown (on MAC failure).
- The attack works with random IVs too, breaking the scheme that was proven secure in [BKN02].
- Something went wrong somewhere!

Further details – Length checking

- RFC 4253 requires implementations to check that length field is “reasonable”.
- Details are implementation-specific.
- Back in 2009, the leading implementation was OpenSSH, then at version 5.1.
    - According to SSH webpage, 80% of servers on the Internet were using OpenSSH around that time.

Further details – Length checking in OpenSSH

- OpenSSH5.1 performs two length checks on the length field (LF) when decrypting the first ciphertext block:
  - Check 1: $5 \leq LF \leq 2^{18}$.
  - Check 2: total length (4+LF) is a multiple of the block size:

$$LF + 4 \bmod BL = 0.$$

- Each check produces a *different* error message on the network, distinguishable by attacker.
- If both checks pass, then OpenSSH waits for more bytes, then performs MAC check, resulting in a third kind of error message.

Further details – Length checking in OpenSSH

- Check 1 ($5 \leq LF \leq 2^{18}$) passes with probability approx. 2^{-14}.
- If it passes, then with high probability, 14 MSBs of LF are "0".
  - Pass/fail detectable via error message.
  - Hence attack with success prob. 2^{-14} recovering 14 bits of confirmed plaintext.

- Check 2 ($LF + 4 \bmod BL = 0$) passes with probability 1/BL, typically 2^{-3} or 2^{-4}.
  - If it passes, then some (3 or 4) LSBs of LF are revealed.
  - Pass/fail detectable via error message/connection entering wait state.
  - If wait state is entered, then the attack proceeds as before.

- Overall, the attack on OpenSSH5.1 recovers 32 bits of plaintext with prob. 2^{-18} (for BL= 16) and requires injection of at most 2^{18} bytes of data.

Does the attack matter?

- On the one hand, the attack has low success rate, only recovers 32 bits of plaintext, and causes the SSH connection to abort.
- On the other hand, an attacker can apply the attack to many connections, boosting his overall success rate.
- Can also iterate the attack against clients that perform auto-reconnects.
- Think about what kinds of data SSH might be protecting.
- SSH was meant to be bullet-proof; the attack showed it was not.
- It left the provable security of the SSH BPP unresolved.

Countermeasures to the attack

- **Abandon CBC-mode?**
  - Alternatives available at that time: CTR, RC4.
  - Dropbear implemented CTR and relegated CBC mode in version 0.53.
- **Develop new modes?**
  - Modes based on Generic EtM, AES-GCM, ChaCha20-Poly1305 were subsequently added to OpenSSH.
- **Patch CBC-mode?**
  - OpenSSH5.2 also introduced a patch to stop the specific attack on CBC mode.

The OpenSSH patch

- Basic idea: hide the errors from the adversary.
  - If the length checks fail, do not send an error message, but wait until 2^{18} bytes have arrived, then check the MAC.
  - If the length checks pass, but the MAC check eventually fails, then wait until 2^{18} bytes have arrived, then check the MAC.
- No error message is ever sent until 2^{18} bytes of ciphertext have arrived.
- Can no longer count bytes to see how many are required to trigger MAC failure.

Theory lesson from the SSH attack

- Model used for security proof in [BKN02] was inadequate.
  - It assumed length was known and atomic processing of ciphertexts.
  - But fragmented adversarial delivery of ciphertext over TCP is possible.
  - Implementations have to decrypt first block to find out how long plaintext is meant to be, and act on it before performing any authentication.
- **That's not reflected in any of the AE/AEAD security models!**
- **And there's no CAESAR requirement that looks like this!**

The State of AEAD in SSH Today

The state of AEAD in SSH today

- In [ADHP16], we performed a measurement study of SSH deployment.
- We conducted two IPv4 address space scans in Nov/Dec 2015 and Jan 2016 using ZGrab/ZMap.
- Grabbing banners and SSH servers' preferred ciphers.
  - Actual cipher used in a given SSH connection depends on client and server preferences.
- Roughly 2^{24} servers found in each scan.
- Nmap fingerprinting suggests mostly embedded routers, firewalls.

The state of AEAD in SSH today: SSH versions

software	scan 2015–12		scan 2016–01	
dropbear_2014.66	7,229,491	(42.0%)	8,334,758	(47.0%)
OpenSSH_5.3	2,108,738	(12.3%)	2,133,772	(12.0%)
OpenSSH_6.6.1p1	1,198,987	(7.0%)	1,124,914	(6.3%)
OpenSSH_6.0p1	554,295	(3.2%)	573,634	(3.2%)
OpenSSH_5.9p1	467,899	(2.7%)	500,975	(2.8%)
dropbear_2014.63	422,764	(2.5%)	197,353	(1.1%)
dropbear_0.51	403,923	(2.3%)	434,839	(2.5%)
dropbear_2011.54	383,575	(2.2%)	64,666	(0.4%)
ROSSSH	345,916	(2.0%)	333,992	(1.9%)
OpenSSH_6.6.1	338,787	(2.0%)	252,856	(1.4%)
dropbear_0.46	301,913	(1.8%)	335,425	(1.9%)
OpenSSH_5.5p1	262,367	(1.5%)	272,990	(1.5%)
OpenSSH_6.7p1	261,867	(1.5%)	213,843	(1.2%)
OpenSSH_6.2	255,088	(1.5%)	288,710	(1.6%)
dropbear_2013.58	236,409	(1.4%)	249,284	(1.4%)
dropbear_0.53	217,970	(1.3%)	213,670	(1.2%)
dropbear_0.52	132,668	(0.8%)	136,196	(0.8%)
OpenSSH	110,602	(0.6%)	108,520	(0.6%)
OpenSSH_5.8	88,258	(0.5%)	[illegible]	[illegible]
OpenSSH_5.1	86,338	(0.5%)	[illegible]	[illegible]
OpenSSH_5.3p1	84,559	(0.5%)	0	[illegible]
OpenSSH_7.1	83,793	(0.5%)	0	[illegible]

Mostly OpenSSH and dropbear; others less than 5%.

The state of AEAD in SSH today: SSH versions

software	scan 2015–12		scan 2016–01	
dropbear_2014.66	7,229,491	(42.0%)	8,334,758	(47.0%)
OpenSSH_5.3	2,108,738	(12.3%)	2,133,772	(12.0%)
OpenSSH_6.6.1p1	1,198,987	(7.0%)	1,124,914	(6.3%)
OpenSSH_6.0p1	554,295	(3.2%)	573,634	(3.2%)
OpenSSH_5.9p1	467,899	(2.7%)	500,975	(2.8%)
dropbear_2014.63	422,764	(2.5%)	197,353	(1.1%)
dropbear_0.51	403,923	(2.3%)	434,839	(2.5%)
dropbear_2011.54	383,575	(2.2%)	64,666	(0.4%)
ROSSSH	345,916	(2.0%)	333,992	(1.9%)
OpenSSH_6.6.1	338,787	(2.0%)	252,856	(1.4%)
dropbear_0.46	301,913	(1.8%)	335,425	(1.9%)
OpenSSH_5.5p1	262,367	(1.5%)	272,990	(1.5%)
OpenSSH_6.7p1	261,867	(1.5%)	213,843	(1.2%)
OpenSSH_6.2	255,088	(1.5%)	288,710	(1.6%)
dropbear_2013.58	236,409	(1.4%)	249,284	(1.4%)
dropbear_0.53	217,970	(1.3%)	213,670	(1.2%)
dropbear_0.52	132,668	(0.8%)	136,1[illegible]	[illegible]
OpenSSH	110,602	(0.6%)	[illegible]	[illegible]
OpenSSH_5.8	88,258	(0.5%)	[illegible]	[illegible]
OpenSSH_5.1	86,338	(0.5%)	44,1[illegible]	[illegible]
OpenSSH_5.3p1	84,559	(0.5%)	[illegible]	[illegible]
OpenSSH_7.1	83,793	(0.5%)	[illegible]	[illegible]

Dropbear at 56-58%. 886k older then version 0.53, so vulnerable to variant of 2009 CBC-mode attack!

The state of AEAD in SSH today: SSH versions

software	scan 2015–12		scan 2016–01	
dropbear_2014.66	7,229,491	(42.0%)	8,334,758	(47.0%)
OpenSSH_5.3	2,108,738	(12.3%)	2,133,772	(12.0%)
OpenSSH_6.6.1p1	1,198,987	(7.0%)	1,124,914	(6.3%)
OpenSSH_6.0p1	554,295	(3.2%)	573,634	(3.2%)
OpenSSH_5.9p1	467,899	(2.7%)	500,975	(2.8%)
dropbear_2014.63	422,764	(2.5%)	197,353	(1.1%)
dropbear_0.51	403,923	(2.3%)	434,839	(2.5%)
dropbear_2011.54	383,575	(2.2%)	64,666	(0.4%)
ROSSSH	345,916	(2.0%)	333,992	(1.9%)
OpenSSH_6.6.1	338,787	(2.0%)	252,856	(1.4%)
dropbear_0.46	301,913	(1.8%)	335,425	(1.9%)
OpenSSH_5.5p1	262,367	(1.5%)	272,990	(1.5%)
OpenSSH_6.7p1	261,867	(1.5%)	213,843	(1.2%)
OpenSSH_6.2	255,088	(1.5%)	288,710	(1.6%)
dropbear_2013.58	236,409	(1.4%)	249,2[illegible]	[illegible]
dropbear_0.53	217,970	(1.3%)	213,[illegible]	[illegible]
dropbear_0.52	132,668	(0.8%)	136,1[illegible]	[illegible]
OpenSSH	110,602	(0.6%)	108,[illegible]	[illegible]
OpenSSH_5.8	88,258	(0.5%)	89,1[illegible]	[illegible]
OpenSSH_5.1	86,338	[illegible]	[illegible]	[illegible]
OpenSSH_5.3p1	84,559	(0.5%)	[illegible]	[illegible]
OpenSSH_7.1	83,793	(0.5%)	[illegible]	[illegible]

OpenSSH at 37-39%.
130-166k older then version 5.2 and prefer CBC mode, so vulnerable to 2009 attack!

The state of AEAD in SSH today: SSH versions

- Dropbear now dominates OpenSSH.
  - But may be switching back again: Comcast cable IP address range dropped from 2M+ devices running Dropbear (Feb 2016) to 83k (May 2016).
- Long tail of old software versions.
  - Most popular version of OpenSSH is version 5.3, released Oct 2009 (current version is 7.2).
  - Determined by major Linux distros?
- Significant percentage of Dropbear and OpenSSH servers are potentially still vulnerable to the 2009 attack.
  - 8.4% for Dropbear.

The state of AEAD in SSH today: preferred algorithms

encryption and mac algorithm	count	
aes128-ctr + hmac-md5	3,877,790	(57.65%)
aes128-ctr + hmac-md5-etm@	2,010,936	(29.90%)
aes128-ctr + umac-64-etm@	331,014	(4.92%)
aes128-cbc + hmac-md5	161,624	(2.40%)
chacha20-poly1305@	115,526	(1.72%)
aes128-ctr + hmac-sha1	68,027	(1.01%)
des + hmac-md5	40,418	(0.60%)
aes256-gcm@	28,019	(0.42%)
aes256-ctr + hmac-sha2-512	17,897	(0.27%)
aes128-cbc + hmac-sha1	11,082	(0.16%)
aes128-ctr + hmac-ripemd160	10,621	(0.16%)

OpenSSH preferred algorithms

- Lots of diversity, surprising amount of "generic EtM" (gEtM).
- CTR dominates, followed by CBC.
- ChaCha20-Poly1305 on the rise? (became default in OpenSSH 6.9).
- Small amount of GCM.

The state of AEAD in SSH today: preferred algorithms

encryption and mac algorithm	count	
aes128-ctr + hmac-sha1-96	8,724,863	(90.44%)
aes128-cbc + hmac-sha1-96	478,181	(4.96%)
3des-cbc + hmac-sha1	321,492	(3.33%)
aes128-ctr + hmac-sha1	62,465	(0.65%)
aes128-ctr + hmac-sha2-256	36,150	(0.37%)
aes128-cbc + hmac-sha1	14,477	(0.15%)

Dropbear preferred algorithms

- Less diversity than OpenSSH.
- CTR dominates, followed at a long distance by CBC.
- No “exotic” options.

A New Attack on CBC mode in OpenSSH

The OpenSSH patch

- Version 5.2 + CBC mode preferred by roughly 20k OpenSSH servers.
- Recall the OpenSSH patch, in version 5.2 and up:
  - If the length checks fail, do not send an error message, but wait until 2^{18} bytes have arrived, then check the MAC.
  - If the length checks pass, but the MAC check eventually fails, then wait until 2^{18} bytes have arrived, then check the MAC.

- One MAC check is done if length checks fail: on 2^{18} bytes.
- Two MAC checks are done if length checks pass: one on roughly LF bytes, the other on 2^{18} bytes.

Attacking the OpenSSH patch [ADHP16]

- This leads to a timing attack on CBC mode in OpenSSH5.2 and up:
  1. Inject target ciphertext block C_i^*.
  2. Then send 2^{18} bytes as quickly as possible to server.
  3. Time the arrival of the MAC failure message.
- Fast arrival indicates that length checks failed and one MAC computation.
- Slow arrival indicates that the length checks passed and two MAC computations.
- This leaks 18 bits of information about the length field, and hence 18 bits about the target block.

Attacking the OpenSSH patch [ADHP16]

- Size of timing difference:
  - A MAC computation on roughly 2^{17} bytes (the expected value of LF).
  - For HMAC-SHA1, this requires 2^{11} hash compression function evaluations.
  - cf. Lucky 13 timing difference: a single hash compression function!
  - Remote attacker can easily detect difference.
- Success probability of the attack:
  - Need to pass both length checks, so 2^{-18}.
  - Can increase success rate given partial plaintext knowledge in target block.
  - (Idea: wait for the right IV before mounting the attack; more severe attack for random, explicit IV version.)

Attacking the OpenSSH patch [ADHP16]

- Increase number of plaintext bits recovered by using finer-grained timing information.
  - Because the timing delay is proportional to the value of LF.
  - If timing granularity = 1 compression function evaluation, then we can recover up to 30 bits of plaintext from target block.
  - Challenging, but not impossible in co-resident attacker scenario.

- Possible countermeasure to the attack: if MAC fails, then compute second MAC on 2^{18} – LF bytes instead of on all 2^{18} bytes.
- Still leaves residual timing difference because of fine details of HMAC.
- Really need constant time implementation of decryption algorithm to eliminate this class of attack.

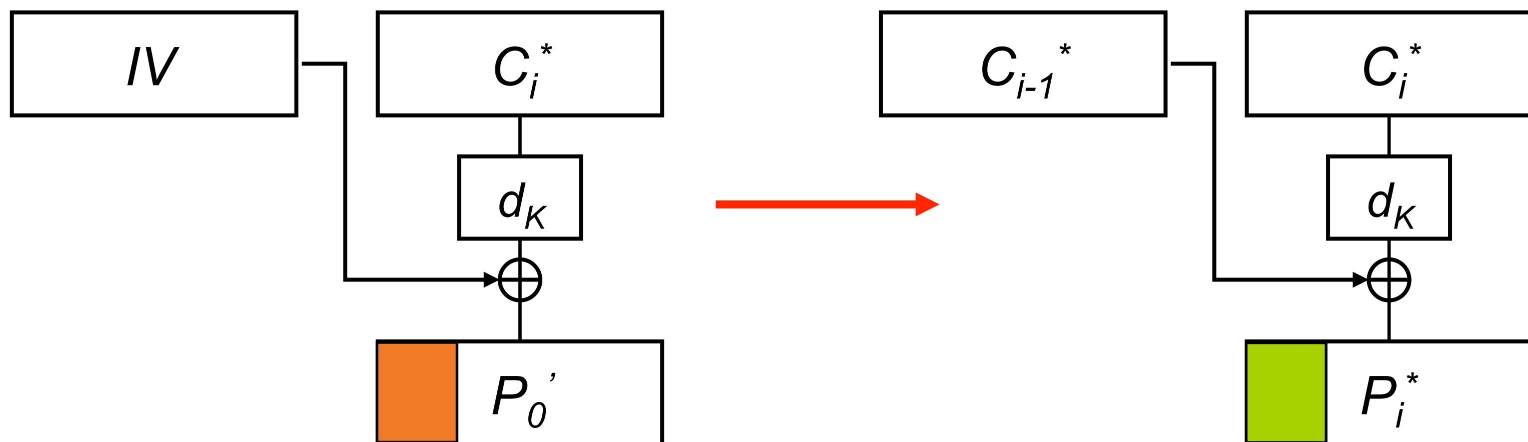
Disclosure of the attack

- We notified the OpenSSH team of the attack on 5th May 2016.
- They are considering adding countermeasures for the next release of OpenSSH (7.3).
- "...we do not feel that an emergency release is necessary, nor that the attack remain secret ahead of such a release."
- OpenSSH has steadily been deprecating old algorithms and modes.
- CBC mode was already disabled by default in OpenSSH 6.7 (but can be re-enabled).
- But OpenSSH cannot force people to stop using old versions of the software.

Security analysis of other SSH and OpenSSH modes –
CTR, gEtM, AES-GCM, ChaCha20Poly1305

Security analysis of other encryption modes

- The [APW09] attacks exploits the attacker's ability to deliver ciphertext fragments and the "cut-and-paste" properties of CBC-mode:
  - Decryption of target block in wrong position is meaningfully related to its decryption in true position:



Security analysis of other encryption modes

- The cut and paste property does not hold for CTR mode.
- Inserting C_i^* in the stream results in *unrelated* plaintext:

$$P_o' = C_i^* \oplus e_K(\mathrm{ctr_o}) = P_i^* \oplus e_K(\mathrm{ctr_i}) \oplus e_K(\mathrm{ctr_o})$$

- But is CTR mode secure against an adversary who can deliver ciphertext in a fragmented fashion?
- Classical security models for symmetric encryption cannot tell us the answer.
- And what about the other modes that have been added to OpenSSH since 2009?
  - gEtM, AES-GCM, ChaCha20Poly1305.

Security analysis of CTR mode in SSH

- [PW10] developed a bespoke security model for CTR mode in SSH and proved it secure (assuming block cipher is a PRP).
- The model allows the attacker to deliver ciphertexts to decryption oracle in a byte-by-byte fashion.
- Decryption oracle intended to accurately model OpenSSH's CTR mode implementation.
  - Sanity checking of length field, with related error messages, MAC failures, etc.
  - Complex pseudo-code descriptions of algorithms and oracles.

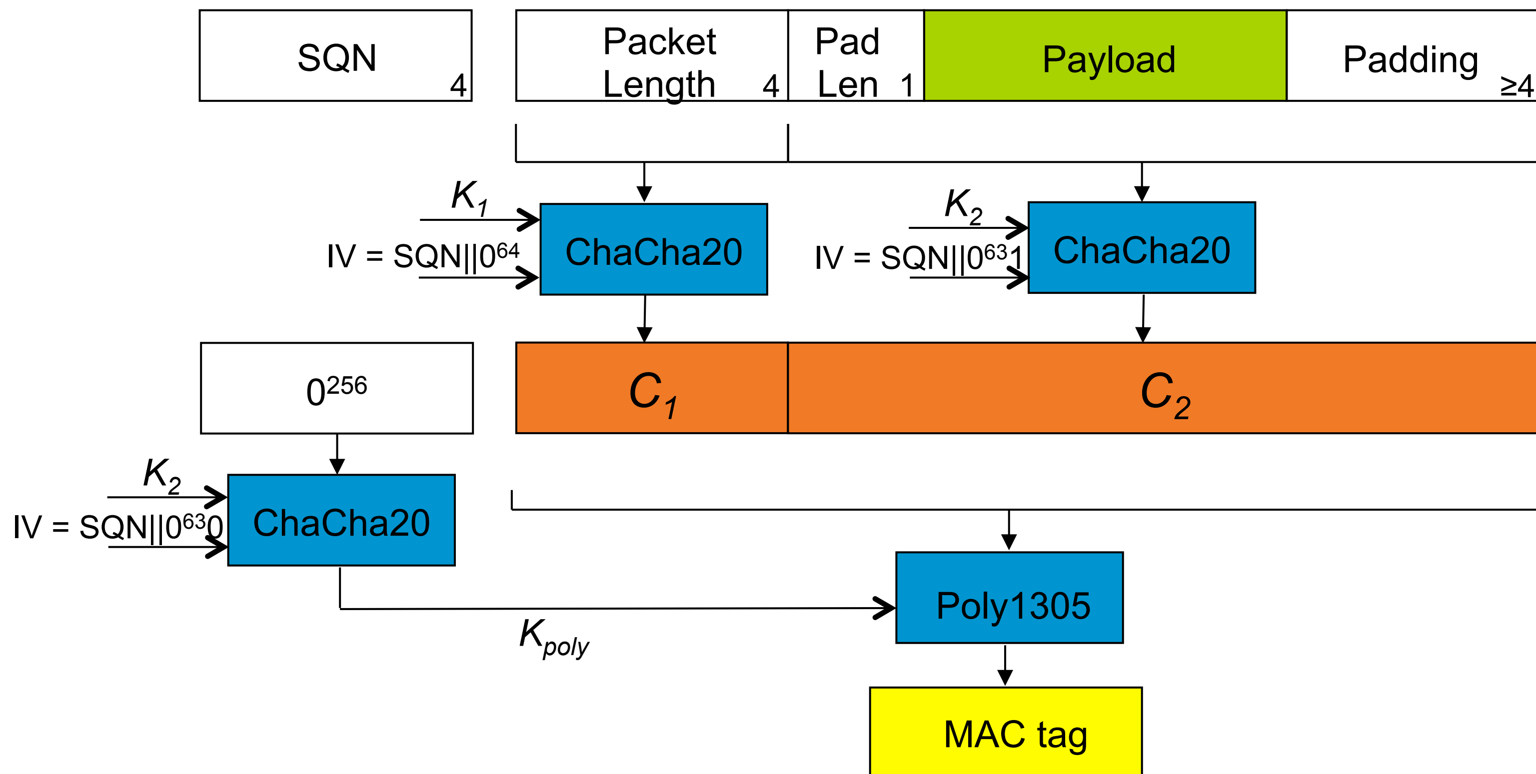
Security analysis of other OpenSSH encryption modes

- [BDPS12] developed a general framework for studying "Symmetric Encryption schemes supporting fragmented decryption".
- The IND-CFA model allows the attacker to deliver ciphertext to a decryption oracle in a symbol-by-symbol fashion and observe any errors/message outputs.
- [BDPS12] also identified additional security properties that SSH attempts to provide:
  - Boundary Hiding (BH) and Denial-of-Service resistance.

Security analysis of other OpenSSH encryption modes

- [ADHP16] used the framework of [BDPS12] to study gEtM, AES-GCM, and ChaCha20-Poly1305 in OpenSSH.
- gEtM and AES-GCM:
  - Derived from AEAD schemes with AD = length field (now unencrypted).
  - Hence sanity checking of length field cannot reveal anything useful to adversary.
  - **Issue in OpenSSH code for gEtM**: because of shared path with legacy E&M code, the MAC is computed once the ciphertext has arrived but is not compared to received MAC until *after* decryption.
  - Hence any errors arising during decryption step will be signalled to attacker.
  - Not a security threat for any currently specified encryption schemes.
  - Both (fixed) gEtM and AES-GCM are provably secure.

ChaCha20-Poly1305 in OpenSSH



Security analysis of ChaCha20-Poly1305 in OpenSSH

- ChaCha20-Poly1305 in OpenSSH:
  - 64-byte key is split into two halves, K_1, K_2.
  - K_1 used to encrypt SSH length field using ChaCha20.
  - K_2 used to encrypt everything else, also using ChaCha20.
  - Poly1305 MAC key is obtained as:

    $$\text{ChaCha20}(K_2, IV = SQN \| 0^{63}0, M = 0^{256}).$$

  - MAC applied to both ciphertext components.
  - Analysis more complex because of encrypted length field.
  - Idea is that using separate keys for encrypting length field and the rest stops attacks.
  - CTR mode analysis shows this to be unnecessary.

Closing remarks

Closing remarks

- Simple security models for symmetric encryption versus complex security properties desired of secure channels.
- In fact, our models for secure channels are still evolving…
- There is much yet to be done here, but community's focus is currently mostly on AEAD.
- Key take-aways:
  - Take cryptographer's abstractions with a pinch of salt.
  - Think top-down, and only bottom-up (from API to crypto, not the reverse).

Closing remarks