

# ARM

# Linux Kernel Cryptographic API and use cases

Gilad Ben-Yossef

Principal Software Engineer

Security IP - Systems & Software Group

Kernel Tel-Aviv Meetup

15 March 2017

©ARM 2017

# About me

- I'm Gilad Ben-Yossef
- I work on upstream Linux kernel cryptography and security in general and Arm® TrustZone® CryptoCell® support in particular.
- I'm working in various forms on the Linux kernel and other Open Source project for close to twenty years – i.e. I'm an old bugger... 😊
- I co-authored “Building Embedded Linux Systems” 2<sup>nd</sup> edition from O'Reilly.
- I'm a co-founder of HaMakor, an Israeli NPO for free and open source software and of August Penguin.
- During my B.A. I once took a course in Cryptography but never finished it because the home assignments were too time consuming – i.e. I am not a cryptographer.



# Agenda

	Topic
19:15	About me
19:20	Alice, meet Bob - a (very) short introduction to cryptography
19:35	Here be Dragons - the Linux crypto API
21:00	Tea break
21:15	DM-Crypt and DM-Verity – protecting your file systems from the scum of the universe
21:50	Q&A



AES SHA-1 SHA-2 SHA-3  
SHA-256 MD5 DES 3DES  
CBC OFB MAC HMAC  
GMAC ECB CFB CTR RSA  
DSA DH ECDH ECC FIPS  
ECDSA EDDSA NIST NSA

# Alice, meet Bob – A (very) short introduction to Cryptography

# Symmetric Encryption

**P**: Plain text  
**C**: Cipher text  
**K**: Key  
**E**: Encryption  
**D**: Decryption



$$C = E(k, P)$$

$$P = D(k, C)$$

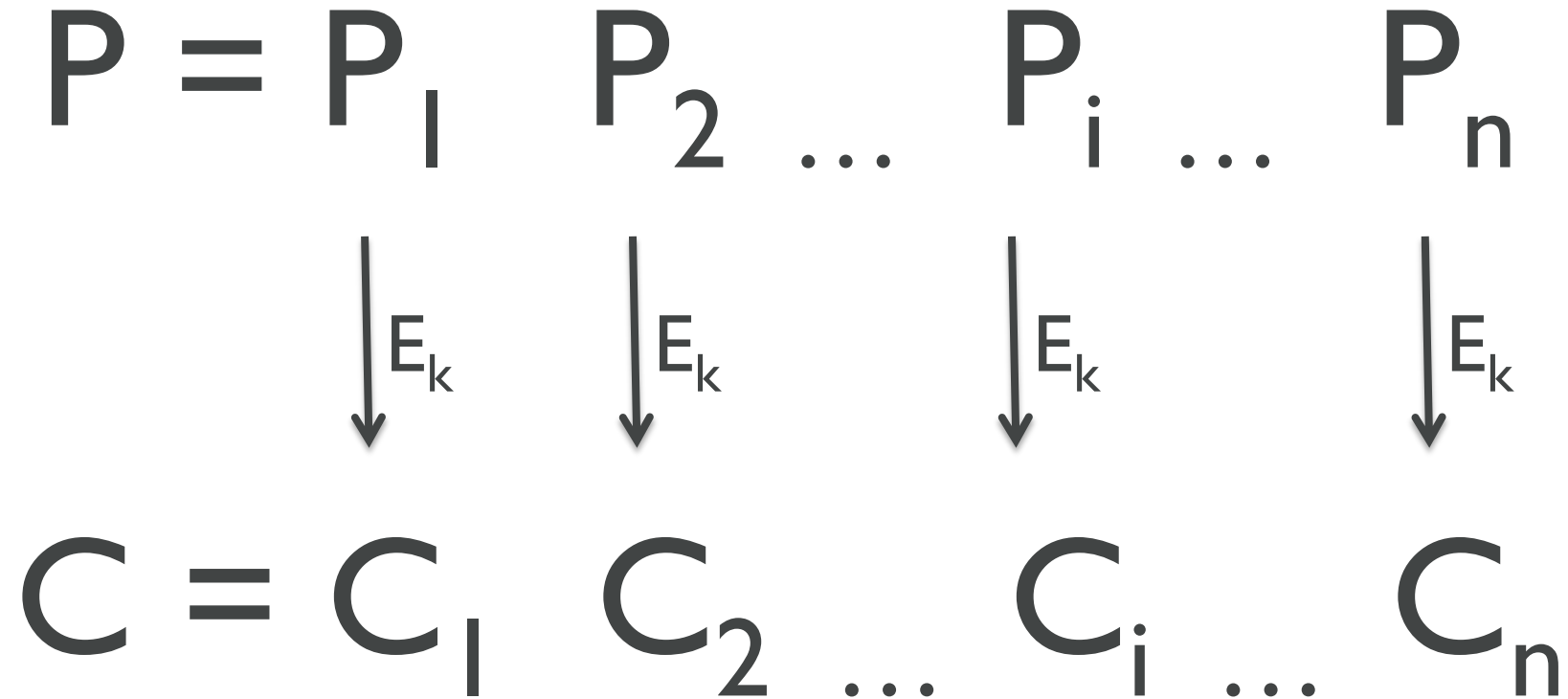
E is a function that takes a key and a block of plain text\* and outputs a block of random looking cipher text\*.

D is the inverse function to D which takes the random looking cipher text\* and the same key as input and outputs the original plain text\*.

# Block Ciphers

**P**: Plain text  
**C**: Cipher text  
**K**: Key  
**E**: Encryption  
**D**: Decryption

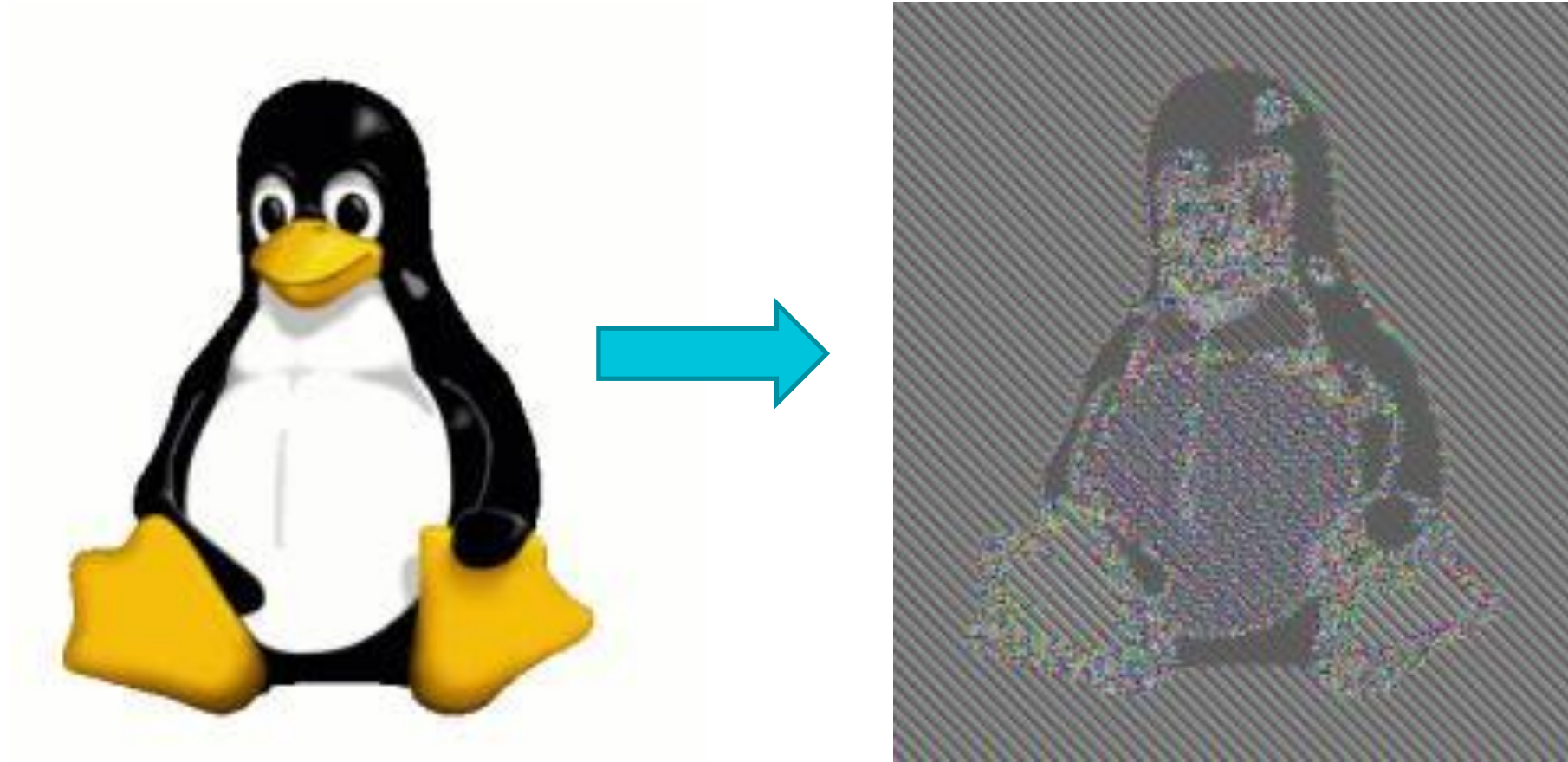
Break down message to blocks, encrypt each block using the key



Examples: ~~DES~~, ~~3DES~~, AES



# The problem with block ciphers



**Identical plain text block yield identical cipher text blocks**

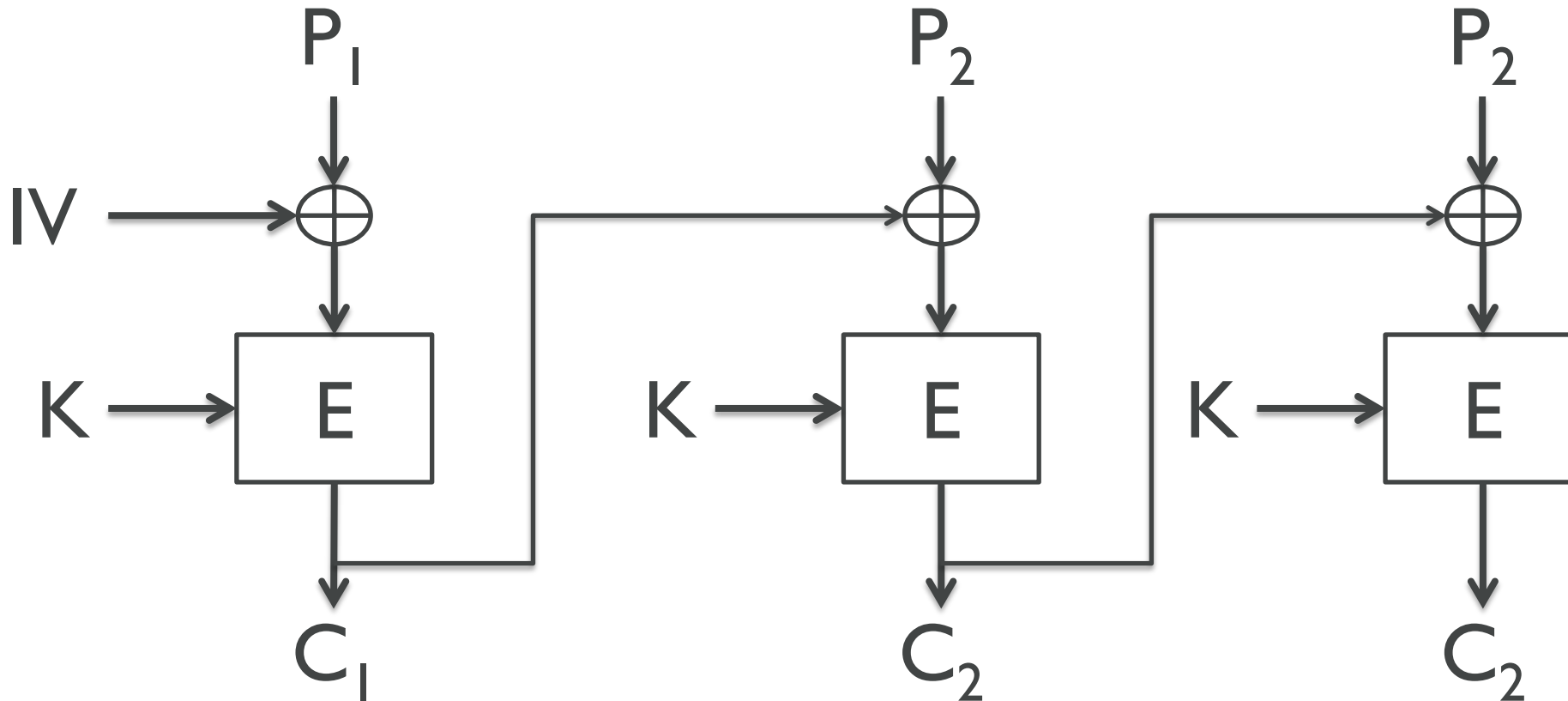
# Modes of Operation

- Symmetric cryptographic functions E and D can be treated as black boxes.
- We can define different ways to use these black boxes to encrypt/decrypt our messages.
- These are referred to as the **Modes of Operation** of the symmetric cipher – how we choose to use it.
- We previously saw the most basic mode of operation called **Electronic Code Book** mode, or **ECB** mode for short.
- Other modes of operations are defined that can, among other things, turn a block cipher into a stream cipher, thus making it more secure.

# Cipher Chaining

**P**: Plain text  
**C**: Cipher text  
**K**: Key  
**E**: Encryption  
**D**: Decryption  
**IV**: Initial Vector

This is the **C**ipher **B**lock **C**haining mode, or **CBC** mode for short:



$$C_0 = IV$$
$$C_i = Ek(P_i \oplus C_{i-1})$$
$$P_i = Dk(C_i) \oplus C_{i-1}$$

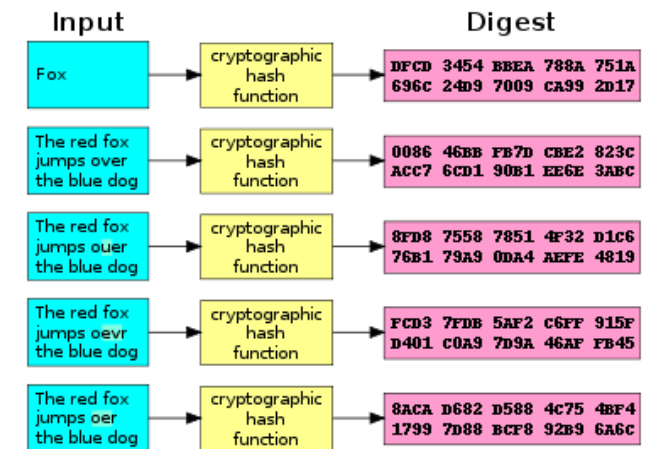
Each cipher block is dependent on the key and the previous blocks.

Other operation modes include **O**utput **F**eed**B**ack, **C**ipher **F**eed**B**ack and **C**oun**T**er modes.

# Cryptographic Hashes (Digests)

- The ideal cryptographic hash function is:
  - Deterministic
  - Quick to compute
  - Infeasible to generate a message from its hash value except by brute force
- A small change to a message changes the hash value so extensively that the new hash value appears uncorrelated with the old hash value
- Infeasible to find two different messages with the same hash value

“Regular” hash function requirements



Examples: ~~MD5~~, ~~SHA1~~, SHA2, SHA3

# Message Authentication Codes

- **Message Authentication Code**, or **MAC** for short, is a method for generating a value that is computed from a message and a secret (key) that can tell the recipient:
  - That the message has not been changed from the way it was originally sent
  - That the sender knows the secret (key)
- For example, this can be a (bad!) MAC built on top of a cryptographic hash function:

$$\text{MAC}_k(M) = H(K \parallel M)$$

Not secure!  
Do not use!

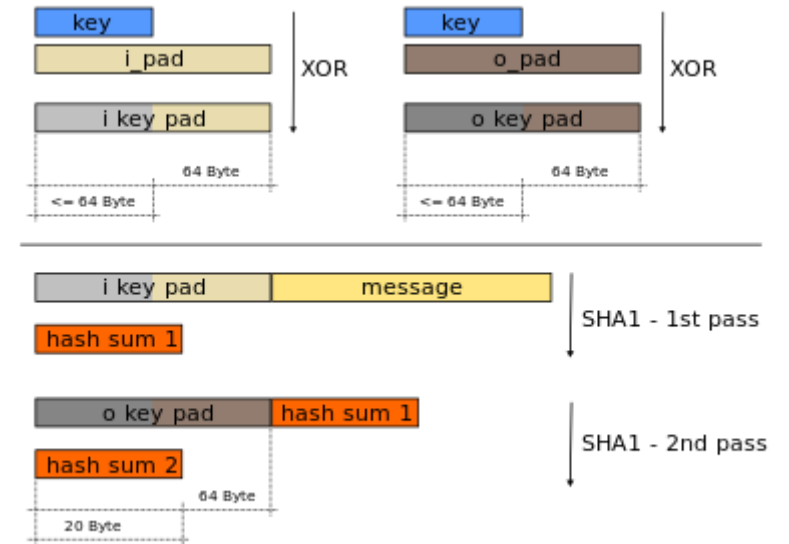
- Example MACs: CBC-MAC, HMAC, GMAC

# Hash Based Message Authentication Codes

$$\text{HMAC}(K, m) = H((K' \oplus \text{opad}) \parallel H(K' \oplus \text{ipad} \parallel m))$$

- Where

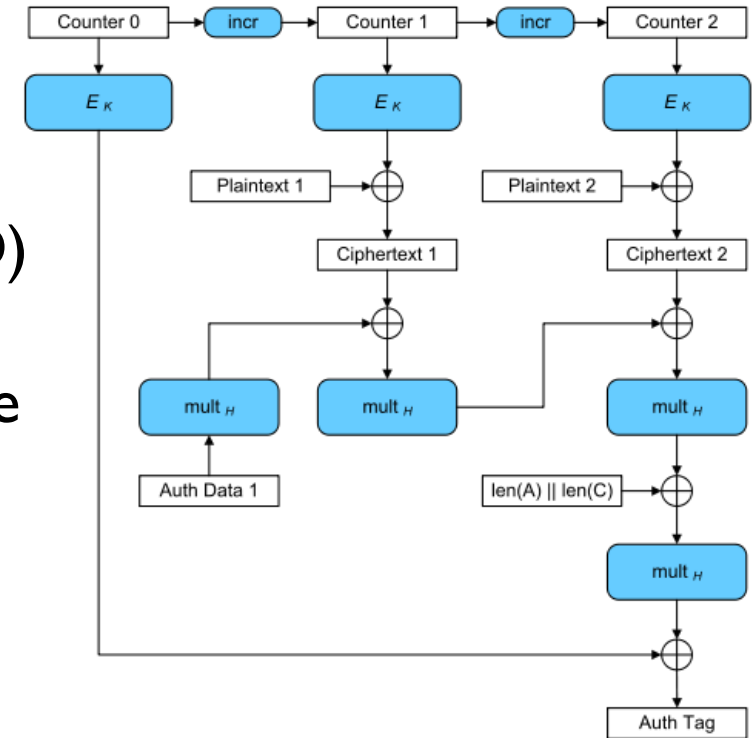
- H is a cryptographic hash function,
- K is the secret key,
- m is the message to be authenticated,
- K' is another secret key, derived from the original key K (by padding K to the right with extra zeroes to the input block size of the hash function, or by hashing K if it is longer than that block size),
- $\parallel$  denotes concatenation,
- $\oplus$  denotes exclusive or (XOR),
- opad is the outer padding (0x5c5c5c...5c5c)
- ipad is the inner padding (0x363636...3636).





# Authenticated Encryption

- **Authenticated Encryption with Associated Data (AEAD)** is a form of encryption which simultaneously provides confidentiality, integrity, and authenticity assurances on the data; decryption is combined in single step with integrity verification.
- These attributes are provided under a single, easy to use programming interface.
- An example of an AEAD in common use is **Galois/Counter Mode**, or **GCM** for short, shown in the picture above.
  - GCM is considered a very efficient AEAD since it can produce both the cipher text and the MAC in a single pass on the data.



# Here be Dragons - The Linux kernel Crypto API

# Transformations

- A **transformation** is an algorithm that transforms data.
  - A transformation can be a cipher (i.e. encryption and decryption), whether symmetric or asymmetric, a hash, compression of data, IV generation and random number generation
  - This is why handle to crypto API object is typically named “**tfm**”.
- A transformation implementation is a code module that registers with the kernel that can provide a transformation.
  - It can be just code running on the CPU, code that uses a special instruction as part of the CPU or a driver to some piece of hardware off of the CPU.
- Multiple transformation implementation can exist that provide that same transformation.
  - E.g. in x86\_64 there are at least implementation of AES: AES-NI, assembler implementation, and plain C.

# Templates

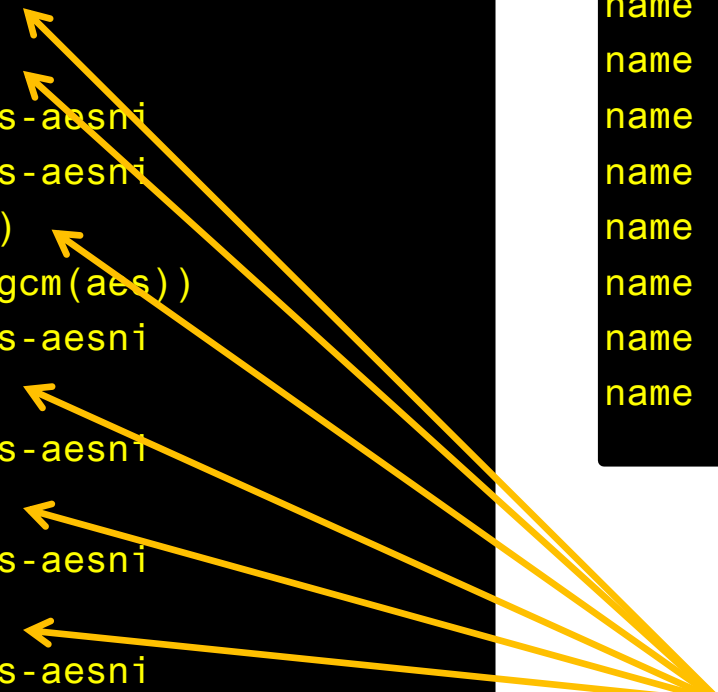
- A transformation can be self contained (.e.g AES or SHA-2) but it can also be a **template** that works on another transformation:
  - CTS(AES) is AES working in Counter operation mode.
  - HMAC(SHA256) is SHA 256 being used as an HMAC.
- Templates can be cascaded
  - E.g. RFC4106(GCM(AES))) is AES in GCM operation mode as used in IPsec.
- Templates can also express generic implementation augmentation.
  - For example CRYPTD(AES) refers to code that runs an AES implementation in parallel inside kernel threads on all the system CPUs in parallel.
- The same template works on multiple underlying transformations.
  - E.g. CTS(AES) and CTS(3DES)

# Priorities and true names


- There can be multiple transformation implementation, including template implementation, which provided the exact same transformation
  - E.g. aes-ni, aes-asm and aes-generic are three different AES implementation on x86\_64
- When you use a **generic name** (e.g. aes) to refer to a transformation, the Crypto API will provide you the implementation with the highest priority that matches your request, based on priority value provided by the implementation when they register with the Crypto API.
  - As we will see ahead you can limit the Crypto API to ignore certain type of providers based on their properties as we will see ahead.
- If you want to refer to a specific implementation, use its **driver name** to refer to it.
  - E.g. “aes-ni” forces the selection of the implementation that uses the AES-NI instruction.

# /proc/crypto

```
gby@gby:~$ cat /proc/crypto | grep name
name      : crc32
name      : xts(aes)
name      : lrw(aes)
name      : __xts-aes-aesni
name      : __lrw-aes-aesni
name      : pcbc(aes)
name      : rfc4106(gcm(aes))
name      : __gcm-aes-aesni
name      : ctr(aes)
name      : __ctr-aes-aesni
name      : cbc(aes)
name      : __ecb-aes-aesni
name      : ecb(aes)
name      : __cbc-aes-aesni
name      : __ecb-aes-aesni
name      : __aes-aesni
```



```
name      : aes
name      : aes
name      : crct10dif
name      : crct10dif
name      : crc32c
name      : stdrng
name      : lzo
name      : aes
name      : sha1
name      : md5
```



We have 3 different implementations of AES

We can use AES in 6 different operation modes



# /proc/crypto

```
gby@gby:~$ cat /proc/crypto
name      : crc32
driver    : crc32-pclmul
module    : crc32_pclmul
priority  : 200
refcnt    : 1
selftest  : passed
type      : shash
blocksize : 1
digestsize : 4
```

```
name      : xts(aes)
driver    : xts-aes-aesni
module    : aesni_intel
priority  : 400
refcnt    : 1
selftest  : passed
type      : ablkcipher
async     : yes
blocksize : 16
min keysize : 32
max keysize : 64
ivsize    : 16
geniv     : <default>
```

Priority

Generic name

Driver name

# Synchronous versus Asynchronous

- A transformation implementation may be either synchronous or asynchronous
- **Synchronous implementations**
  - Run wholly on the CPU itself in a synchronous fashion (take CPU cycles during all the run)
  - Access virtual memory using the CPU MMU mechanism
- **Asynchronous implementations**
  - MAY be implemented using off-CPU resources (i.e. typically dedicated hardware), or not
  - MAY access memory via DMA and not have access to MMU, or not.
  - In fact, synchronous implementation are a sub group of asynchronous ones!
    - E.g. cryptd(aes-ni) turns the synchronous aes-ni implementation into an asynchronous implementation by running it on several cores in parallel.
- You can limit yourself to working only with synchronous implementations by passing the CRYPTO\_ALG\_ASYNC ask flag to allocation calls.

# Backlog

- Transformation implementations have a certain capacity for in flight requests
  - i.e. descriptor ring space, hardware FIFO etc.
- If you pass the `CRYPTO_TFM_REQ_MAY_BACKLOG` flag to callback setting API call, you will allow the implementation provider to queue up back log word to a software queue if it doesn't have enough capacity.
  - If this happens, your callback function will be called twice:
    - Once with err set to `-EINPROGRESS` to indicate it got queued off of the backlog.
    - A second time when it is actually processed
- The kernel now has a generic implementation of this mechanism for transformation providers can use to implement this.
  - However, the author believes it is currently buggy and is working to fix it... ☺

# API Typical Life Cycle

- Allocate a transformation object (tfm)
- Get properties of the transformation chosen by the kernel (e.g. digest size)
- Set global transformation properties (e.g. encryption key)
- Allocate a request object
- Set request specific parameters, such as GFP flags and callback function to call when operation finish
- Initialize the request
- Feed data (e.g. buffers to encrypt or hash)
- Finalize (e.g. read final HMAC value)
- Free request object
- Free transformation object

# API documentation and code examples

- No use copy & paste them here, so just go to:
  - Documentation/cryptoor
  - <http://www.chronox.de/crypto-API/crypto/index.html>

# Pitfalls to look out for

- This call requests only **synchronous** implementations

```
tfm = crypto_alloc_ahash("md5", 0, CRYPTO_ALG_ASYNC);
```

- Yes, it's totally backwards – the mask flag **masks out** the async. Implementations!
- If you see code that passes NULL as a callback like the following, it's probably used the mask flag (see above) to request only synchronous implementations:

```
ahash_request_set_callback(req, 0, NULL, NULL);
```

- Remember that operation may complete synchronously or asynchronously based on the transformation provider discretion only and you need to handle both cases!
- Don't ignore the err parameter of the callback.
- Remember that if you passed the CRYPTO\_TFM\_REQ\_MAY\_BACKLOG flag when setting the callback, your callback **may** be called twice!
- Even the request init function might sometime complete asynchronously.



# DM-Crypt and DM-Verity -

## Protecting your file systems from the scum of the universe

# DM-Crypt

- **dm-crypt** is a transparent disk encryption subsystem in Linux
- It is part of the device mapper infrastructure, and uses cryptographic routines from the kernel's Crypto API.
- dm-crypt was designed to support advanced modes of operation, such as XTS, LRW and ESSIV, in order to avoid watermarking attacks.
- dm-crypt is implemented as a device mapper target and may be stacked on top of other device mapper transformations.
  - It can thus encrypt whole disks (including removable media), partitions, software RAID volumes, logical volumes, as well as files.
  - It appears as a block device, which can be used to back file systems, swap or as an LVM physical volume.
- It is used by Android devices to implement Full Disk Encryption.

# Simple dm-crypt setup example

```
# cryptsetup luksFormat fs3.img
# cryptsetup open --type luks fs3.img croot
# mke2fs /dev/mapper/croot
# mount -t ext2 /dev/mapper/croot /media
# umount /media/
# cryptsetup close croot
```

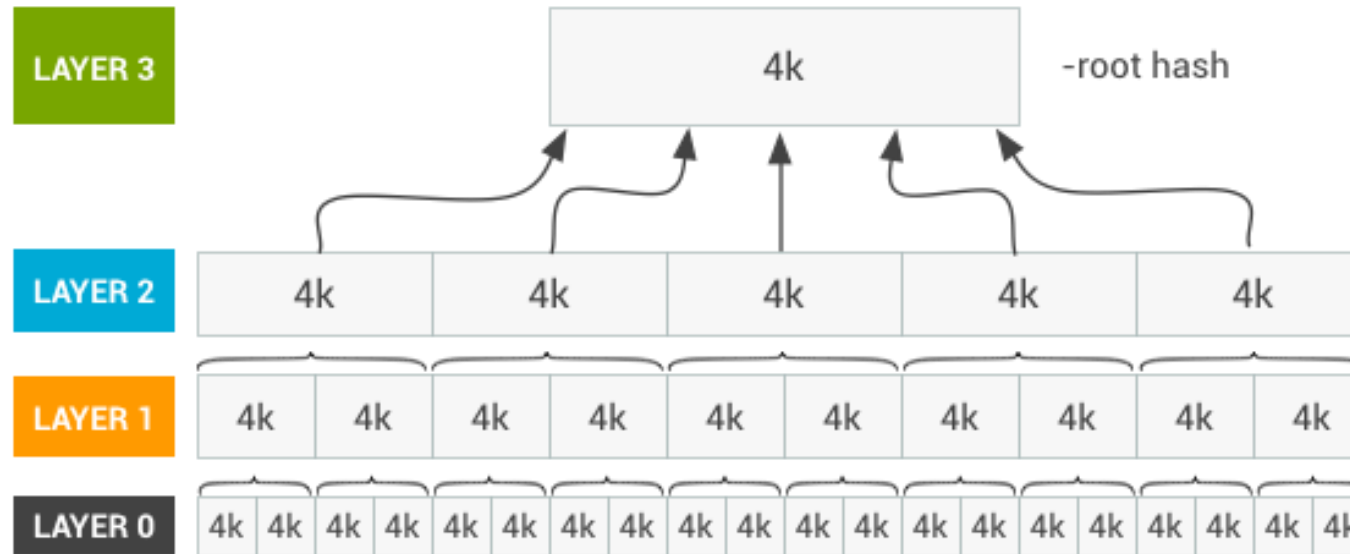
This examples uses AES in XTS operating mode to protect a loopback mounted partition file.

**Make sure to remember your chosen key or your data is lost!**

# DM-Verity

- Device-Mapper's "verity" target provides transparent integrity checking of block devices using a cryptographic digest provided by the kernel crypto API. T
  - This target is read-only.
- dm-verity helps prevent persistent rootkits that can hold onto root privileges and compromise devices.
- The dm-verity feature lets you look at a block device, the underlying storage layer of the file system, and determine if it matches its expected configuration.
- It does this using a cryptographic hash tree. For every block (typically 4k), there is a SHA256 hash.
- Dm-verity can optionally FEC to protect against hash data corruptions.
- DM-Verity is used by Android devices.

# How does DM-Verity work?



- Dm-verity uses a Merkle tree of storage blocks to protect the integrity of the read only data on the storage device, in a way that the integrity can be evaluated in a lazy fashion during runtime instead of pre-mount time.
- It needs a singular trusted root hash to achieve security

# Simple dm-verity setup example

```
# veritysetup format filesystem.img signature.img
# veritysetup create vroot filesystem.img signature.img \
ffa0a985fd78462d95c9b1ae7c9e49...5e3f13c10e4700058b8ed28
# mount -t ext2 /dev/mapper/vroot /media/
# umount /media
# veritysetup remove vroot
```

This examples uses SHA 256 to protect the integrity of the loopback file system provided the root hash is secure.



# Thank you for listening!

## Questions?

**ARM**

@gilad  
gilad@benyossef.com

The trademarks featured in this presentation are registered and/or unregistered trademarks of ARM Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

Copyright © 2017 ARM Limited

©ARM 2017