# Cryptography for storage systems

Christian Cachin

IBM Research - Zurich

10 May 2013

---

# Overview

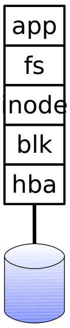
- Encryption in storage systems
- Tweakable encryption
- Integrity protection
- Key management

---

# Encryption in storage systems

---

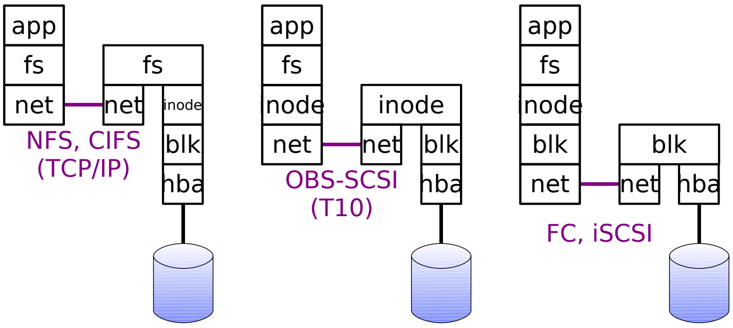
# Traditional storage systems: Inside the box


app
fs
inode
blk
hba

Direct-attached storage

---

# Networked storage systems


app
fs
net
fs
net
inode
blk
hba
NFS, CIFS
(TCP/IP)

NAS
(Network-attached Storage)

app
fs
inode
net
inode
net
blk
hba
OBS-SCSI
(T10)

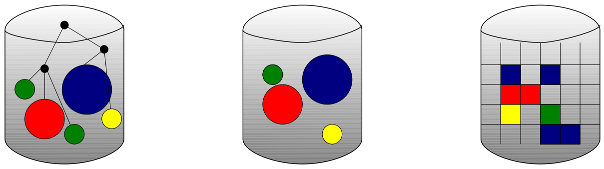
OBS
(Object Storage)

app
fs
inode
blk
net
blk
net
hba
FC, iSCSI

SAN
(Storage-area Network)

---

# Storage-device models



| File server | Object storage dev. | Block device |
|---|---|---|
| - read & write data in file | - read & write bytes in object | - read & write blocks |
| - create & destroy file | - create & destroy object | -- |
| - directory operations | -- | -- |
| - file/dir-based access control | - object-level access control | - device-level access control |
| - space allocation | - space allocation | -- |
| - backup ops | - backup ops | -- |

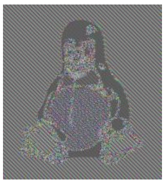
# Tweakable encryption

## Block cipher

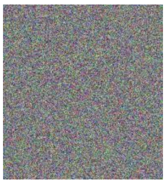
- Deterministic, key-dependent transformation
  - One input block to one output block
  - AES, DES, Blowfish ...
  - Blocks size: typically 128 bits (16 bytes)
  - Key size: typically 128 bits and more
- Formally block cipher implements a pseudo-random permutation (PRP)
  - Appears like a random permutation to any computationally bounded observer (who does not have the key)
- Mode of operation ("chaining" mode) required
  - Electronic-codebook mode (ECB) means no chaining

## Why a block-cipher mode of operation?

Plaintext as bitmap picture



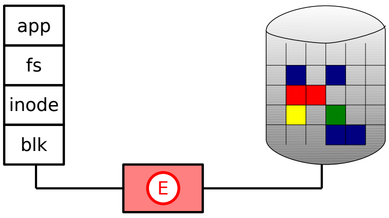
Encrypted in ECB mode



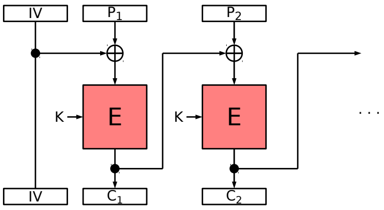
Encrypted in secure mode of operation

## Encryption at the block layer

- “Device-level” encryption of 512-byte sectors
- Transparent to storage system → no extra space available to chaining mode
- IEEE SISW standardization: P1619/ .1 / .2

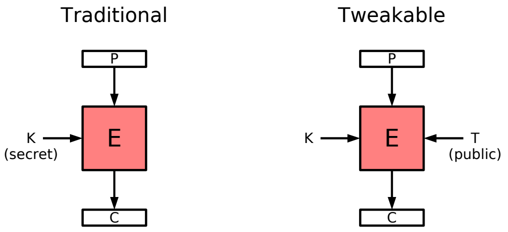


## Using CBC mode



- Random IV required, but there is no space to store
- → Derive IV from sector address
  - IV = $E_K$( disk id || sector address )
  - IV = $E_{Hash(K)}$( disk id || sector address )
- Leaks location of first updated block within sector
- Attack possible if adversary may invoke decryption for some sectors, but not for others

## Tweakable encryption (TwE)

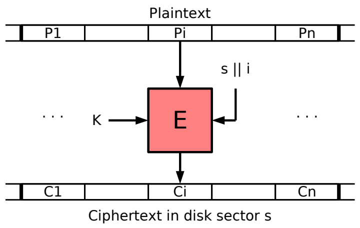


$E_K$() is PRP

$E_{K,T}$() is a PRP for every T

- $E_K$() is a PRP, deterministic after picking K
  - Same permutation in every instance
- Tweakable $E_{K,T}$() is a family of independent permutations, indexed by T [Liskov, Rivest, Wagner, CRYPTO '02]
  - T = address of block

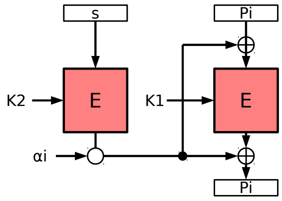
## Narrow-block TwE



Tweaked block
=
cipher block
(16 bytes)

- Every block in sector encrypted independently
  - Tweak is sector address s plus block index i
- Leaks only that block has been updated
- "Better" security against active attacks

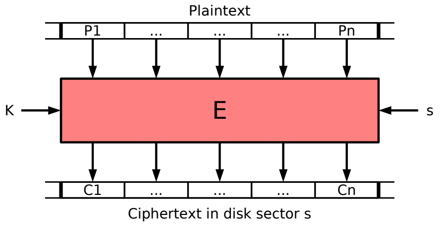
## Narrow-block TwE mode



- XTS-AES mode based on XEX [Rogaway, ASIACRYPT '04]
  - Tweak = sector s || block index i
  - Key K = K1 || K2
  - $\alpha$ in GF($2^{128}$), primitive element, $\alpha^i$ efficient for i=0,1,2...
- Standardized by IEEE P1619 and NIST SP 800-38E
- Used in practice (e.g., Truecrypt, FDE for disk drives)

## Wide-block TwE

- One tweaked blockcipher encryption per sector
- Tweak is sector address s
- Leaks only that sector has been updated



Tweaked block
=
disk sector
(512 bytes)

## Wide-block TwE

- Proposed implementations are slower than AES
  - EME2-AES: 2x AES
  - XCB-AES: 1x AES + 2x GF($2^{128}$)-mult.
- Standardized as IEEE P1619.2 (2010)
- Overhead considered to be (too) costly
  - No practical deployment so far

## Comparison

| | **CBC mode** | **TwE narrow** | **TwE wide** |
|---|---|---|---|
| **Passive adversary**<br>- Localize changes in encrypted file | First changed block in sector | All blocks that changed | Whole sector (best possible) |
| **Active adversary**<br>- Trigger controlled change of plaintext | Change one block & move blocks | None | None |
| **Situation in practice** | Deployed | Deployed | Not used |

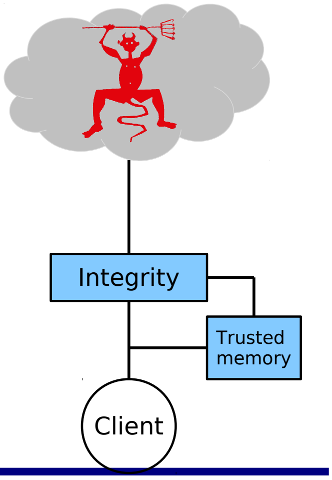
**How realistic are active attacks?**

- Encryption in OS kernel, attack requires access to stored bits
- Unlikely for laptops
- More plausible for virtual disk images on cloud storage

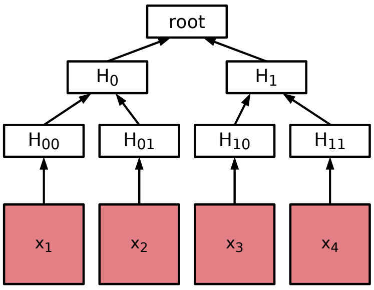
## Integrity protection

# Integrity protection for one client

- Storage consists of n data items $x_1, \ldots, x_n$
- Client accesses storage via integrity-protection layer
  - Uses small trusted memory to store short reference hash value v (together with encryption keys)
- Integrity layer operations
  - Read item and verify w.r.t. v
  - Write item and update v accordingly



# Hash trees for integrity checking (Merkle trees)

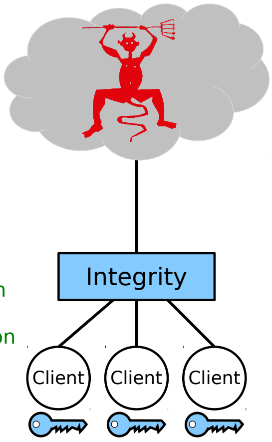


Read & write operations need work O(log n)
- Hash operations
- Extra storage accesses

- Parent node is hash of its children
- Root hash value commits all data blocks
  - Root hash in trusted memory
  - Tree is on extra untrusted storage
- To verify $x_i$, recompute path from $x_i$ to root with sibling nodes and compare to trusted root hash
- To update $x_i$, recompute new root hash and nodes along path from $x_i$ to root

# Multi-client integrity protection

- Single-client solution
  - Relies on hash value v
  - Stored locally in trusted memory
  - Changes after every update operation
- Multiple clients?
  - Need to synchronize trusted memories
- Solution with digital signatures
  - Every client associated with a public/private key pair
  - Write operation produces signature σ on hash v
  - Client stores signature and hash (σ, v) on cloud
- Replay attacks
  - This approach permits replay attacks ...
  - Prevented using trusted coord. service

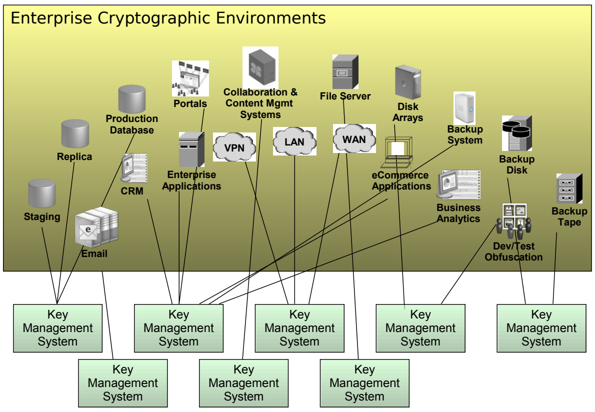


# Multi-client integrity protection and forking attacks

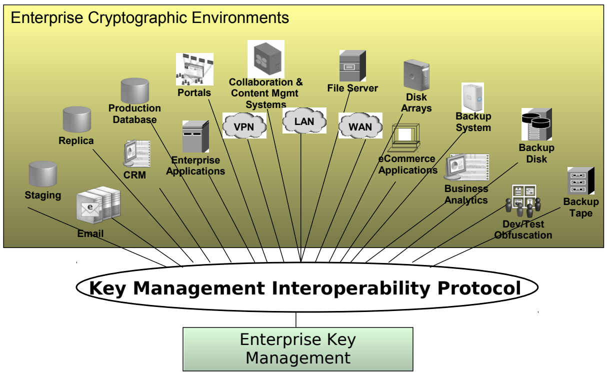
- Server may present different views to separated clients
  - E.g., not show the most recent WRITE operation to a reader
  - Creates a "fork" between their histories
  - Clients cannot prevent this without communication
- Use fork linearizability [Mazieres, Shasha, PODC '02]:
  - If malicious server forks the views of two clients once, then
    → their views are forked ever after
    → they never again see each others updates
- Every inconsistency or integrity violation results in a fork
  - Best achievable guarantee for storage on untrusted server
  - Forks can be detected on a "cheap" low-security external channel
    - Use only a semi-trusted coordinator [Cachin et al., SIAM J. Comput, 2011]
  - Prototype implementation in VENUS [Shraer et al., CCSW 2011]

# Key management

# Today - Proprietary key mgmt.



# Future - Standardized key management



# OASIS Key Management Inter-operability Protocol (KMIP)

- OASIS…? XML
- Client-server protocol
- Defines objects with attributes, plus operations
  - Objects: symmetric keys, public/private keys, certificates, threshold key-shares …
  - Attributes: identifiers, type, length, lifecycle-state, lifecycle dates, links to other objects …
  - Operations: create, register, attribute handling …

# OASIS KMIP

- KMIP draft spec prepared by industry group
  - HP, IBM, RSA-EMC, nCipher/Thales, Brocade, Seagate, LSI, NetApp
  - IBM- and IBM Zurich-led (editor and TC co-chair)
- OASIS KMIP Technical Committee (2009)
  - KMIP v1.0 released in Oct. 2010
  - KMIP v1.1 released in Feb. 2013
- http://www.oasis-open.org/committees/kmip/
- Today deployed by multiple vendors in storage-encryption context

# KMIP objects and attributes

- Objects of four types
  - Symmetric keys, public keys, private keys, certificates
- ~50 attributes
  - Identifier, state, initialization time, activation time, deactivation time …
- Access-control specific attributes
  - ACL, usage …
- KMS accessed by remote users over network

# KMIP operations

- Create(id, parameters) → OK
- Derive(id, parent_id, aux_data) → OK

- Store(id, clear_key) → OK
- Import(unwrapping_key_id, wrapped_key) → OK

- Read(id) → clear_key
- Export(id, wrapping_key_id) → wrapped_key

- Read attributes(id) → {attributes}
- Set attributes(id, {new_attributes}) → OK

- Search(id, condition) → {ids}
- Destroy(id) → OK -- deletes key, but leaves attributes intact
- Delete(id) → OK -- deletes key and attributes (if possible)

Most ops. are straightforward, but some involve cryptography.

# Access control model for KMIP

- Users
  - Determined by user registry (e.g., LDAP)
  - Special users: any, creator
- Permissions
  - Per-object
    - Admin, Derive, Destroy, Export, Read, ReadAttributes, Unwrap, Wrap
  - Per-user
    - Create, Store
- Ever object o has an acl attribute

  $o.acl \subset \{(u, p) \mid u \in Users, p \in Permissions\}$

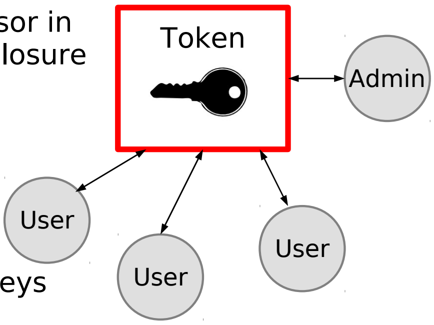
# A key server is a crypto API

- Key server executes cryptographic operations
- So far, cryptographic security APIs have been linked to secure hardware tokens (IBM CCA, PKCS #11 ...)
- We extend the study of cryptographic security APIs to
  - Key-management systems on a network
  - Accessed by multiple users

# Cryptographic tokens?

*Cryptographic processors*
*Hardware security modules (HSM)*

- Crypto co-processor in tamper-proof enclosure
- Keys never leave token in clear
- Executes all cryptographic operations with keys



# Commercial crypto tokens

HP Atalla Ax160

IBM 4765

nCipher/Thales netHSM

Infineon TPM

Tamper-resistant and -responsive according to FIPS 140-2, up to Level 4

# Why cryptographic tokens?

"Cryptographic keys must not leave secure HW."

- Introduce a separation between:
  - Administration of keys → security officer
  - Administration of servers → server operator

  → Fewer opportunities for insider attacks
- Found in many corporate environments
  - Government, finance, telecom ...
- But also in your pocket
  - Smartcards, SIM cards, transport tickets ...

# Interacting with a token

- User u authenticates to token
  u ∈ {security-officer, application}
- u invokes operations through Crypto API
  - Operations on payload
    - Encrypt, decrypt, sign, verify ...
  - Key-management operations
    - Create, store, read*, update* key
    - Derive key from a parent key
    - Wrap key / export
    - Unwrap key / import

    * Restricted to admin!
- Standardized interfaces
  - PKCS #11 [EMC/RSA]
  - Common cryptographic architecture (CCA) [IBM]

# Problems with crypto APIs (1)

- Legacy API policies are often "underspecified"
  - Nevertheless, they aim to protect keys
- Purely logical attacks → API attacks
  - Expose a protected key [Anderson, Bond, Clulow]
- Example attack on PKCS #11
  - *Sensitive* keys must not be exposed in clear
  - PKCS #11 denies read operation by user u ≠ admin if key k is *sensitive*
  - But allows u to wrap k under a non-*sensitive* key d

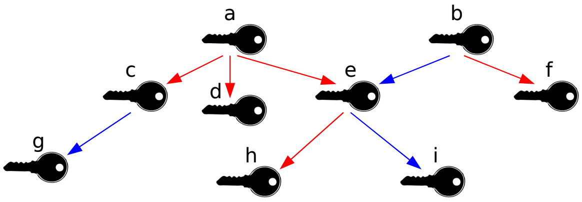
    → user u wraps k under d and reads d

    → this exposes k in clear

## *Problems with crypto APIs (2)*

- Why?
- Why is access control with simple read/write permissions not enough to protect keys?
- Because keys may depend cryptographically on other keys
  - Only cryptographic operations create such dependencies
- Propose to keep track of dependencies with a model for **strict access control** [Cachin, Chandran, CSF '09]

## *Dependencies among keys*



- Key k depends on a key p ⇔
  - Key k was derived from p
    - derive(a,c), derive(a,d), derive(a,e) ...
  - Key k was wrapped under p
    - wrap(c,g), wrap(b,e) ...

## *New attributes for keys*

- strict ∈ {false, true}
  - Determines if object governed by "strict policy"
- dependents ⊆ Objects
  - Other objects whose cryptographic value can be computed from the cryptographic value of the object
- ancestors ⊆ Objects
  - Other objects on which the object depends
- readers ⊆ Users
  - Users who have executed read(k) for some key k such that object ∈ k.dependents

## *Basic and strict policies*

- If o.strict = true, then o benefits from strict security policy
- Otherwise, o underlies basic access-control policy
- Strict security policy respects dependencies between keys in access decisions

## *Basic authorization*

Basic authorization rule of permission p
for user u on object o:

BASICAUTH(u, p, o) =
(any, p) ∈ o.acl **or**
(u = o.creator **and** (creator, p) , p) ∈ o.acl **or**
(u, p) ∈ o.acl.

## *Implementation of read*

Condition for user u to execute read(o):
o.strict = false **and** BASICAUTH(u, Read, o) **or**
o.strict = true **and**
∀ q ∈ o.dependents, BASICAUTH(u, Read, q)

Effect:
**if** o.strict = true **then**
∀ q ∈ o.dependents, q.readers ← q.readers ∪ {u}

# Implementation of export

Condition for user u to execute export(o, w):

o.strict = false **and** BASICAUTH(u, Export, o) **or**  
o.strict = true **and** w.strict = true **and**  
  BASICAUTH(u, Export, o) **and** BASICAUTH(u, Wrap, w) **and**  
  ∀ v ∈ w.readers, ∀ q ∈ o.dependents,  
    BASICAUTH(v, Read, q) **and**  
  w ∉ o.dependents

Effect:

**if** o.strict = true **then**  
  ∀ v ∈ w.readers, o.readers ← o.readers ∪ {v}  
  w.dependents ← w.dependents ∪ o.dependents  
  o.ancestors ← o.ancestors ∪ w.ancestors

Use authenticated encryption for key wrapping

# Implementation of import

Condition for u to execute import(w, wrapped) in strict mode:

BASICAUTH(u, Unwrap, w) **and**  
  w.readers = ∅ **and**  
  w.strict = true **and**  
  !∃ key in DB with same digest as o,  
    **where** o = unwrap(wrapped)

Effect:

w.dependents ← w.dependents ∪ o.dependents  
o.ancestors ← o.ancestors ∪ w.ancestors

Imported key must not yet exist in the system

# Destroy and delete

Condition for u to execute destroy(o):

BASICAUTH(u, Destroy, w)

Destroys only the cryptographic material, leaves the object attributes in DB

Condition for u to execute delete(o):

BASICAUTH(u, Admin, w)

Destroys the object and its attributes, but **only if** o.dependents = ∅.

# Notes

- Model of Cachin-Chandran (CSF '09) has only one key server
  - Server should keep a global history
  - Multiple servers need to synchronize state
- Prototype implementation at IBM Zurich
  - All keys and dependency data stored in DB
  - Compact representation, independent of history
- Requires system to track all operations
- Experience with prototype shows it is efficient
  - No exposure to real world yet

# References

- Christian Cachin, Nishanth Chandran. "A secure cryptographic token interface." In *Proc. Computer Security Foundations (CSF)*, 2009.
- Mathias Björkqvist, Christian Cachin, Robert Haas, Xiao-Yu Hu, Anil Kurmus, René Pawlitzek, and Marko Vukolic. "Design and implementation of a key-lifecycle management system." In *Proc. Financial Cryptography*, 2010.
- OASIS Key Management Interoperability Protocol (KMIP) Technical Committee, "Key Management Interoperability Protocol Version 1.1" OASIS Standard, 2013. https://www.oasis-open.org/committees/documents.php?wg_abbrev=kmip