Zurich Research Laboratory

Cryptographic Methods for Protecting Storage Systems

Christian Cachin <cca@zurich.ibm.com>

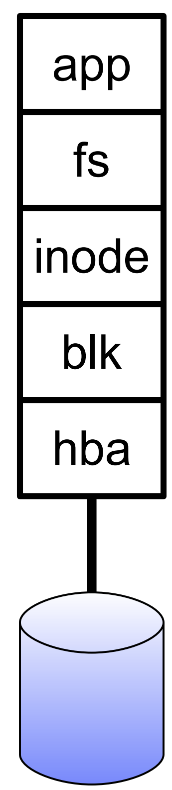
FAST 2008
26 February 2008

www.zurich.ibm.com

Overview

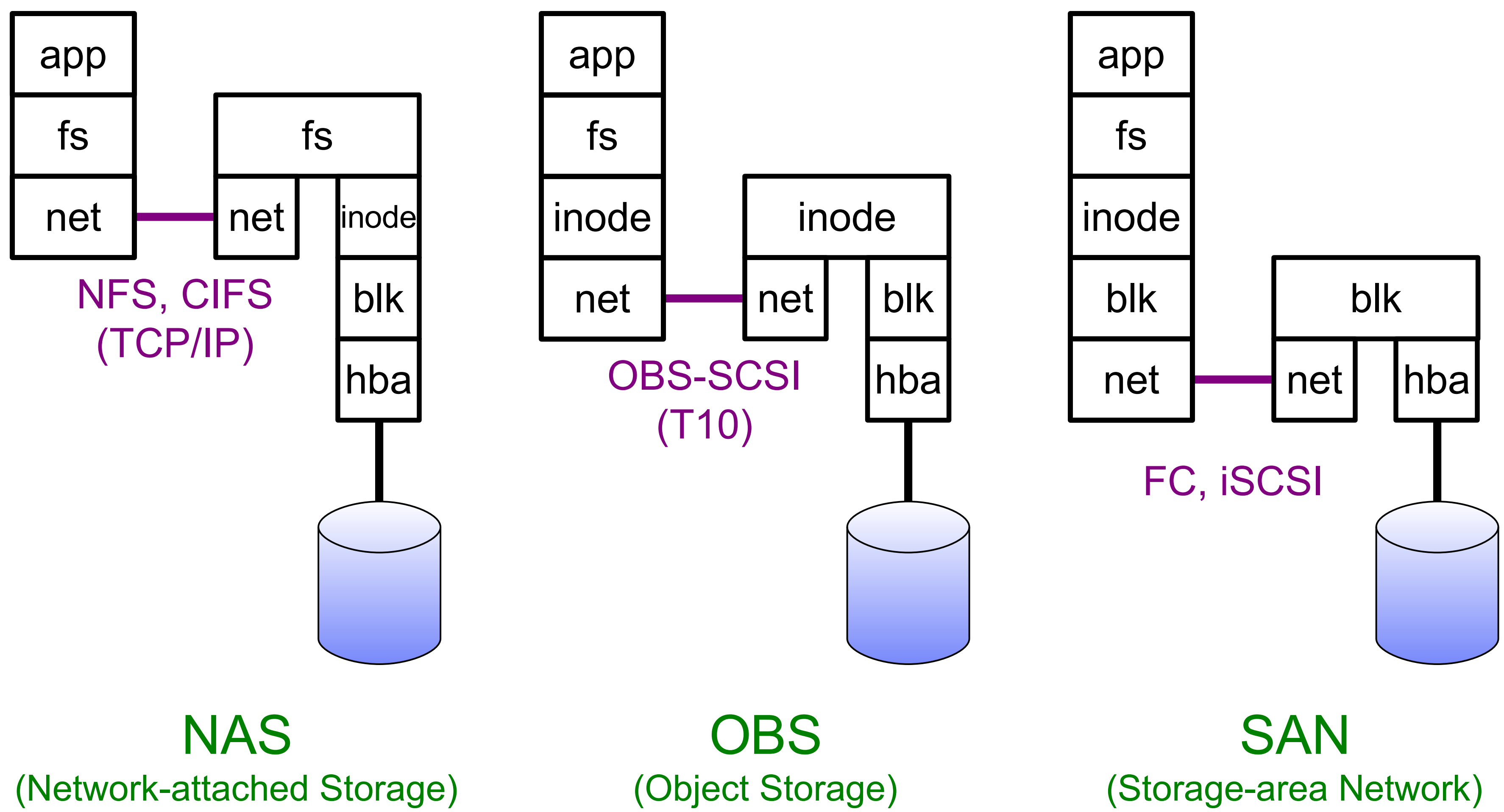
- Design options for security in storage systems
- Block/record-layer security
  - → Tweakable encryption and other block-cipher modes
  - → Hybrid block-integrity protection
  - → Authenticated record-encryption
- Object-layer security
  - → Capabilities in Object Storage
- Filesystem security
  - → Designs for key management
  - → Encryption using lazy revocation and key updating
  - → Integrity protection in filesystems
  - → Consistent access to untrusted storage*
- Cryptography for storage in action
  - → Tape drive with encryption (IBM TS1120)
  - → TCG storage specification and drive-encryption (Seagate)
  - → A cryptographic SAN filesystem

Past Storage Systems: Inside the Box

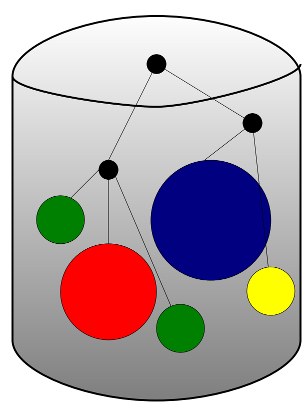


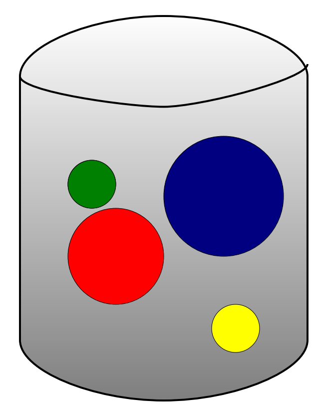
Direct-attached Storage

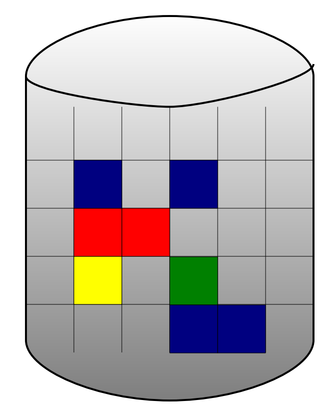
Current Storage Systems: Local



Network-based Storage Devices







File server
- read & write data in file
- create & destroy file
- directory operations
- file/dir-based access control
- space allocation
- backup ops

Object storage dev.
- read & write bytes in object
- create & destroy object
--
- object-level access control
- space allocation
- backup ops

Block device
- read & write blocks
--
--
- device-level access control
--
--

Future Storage Systems: Anywhere

amazon web services™

Amazon S3
(Simple Storage Service)

Google

IBM

BitTorrent

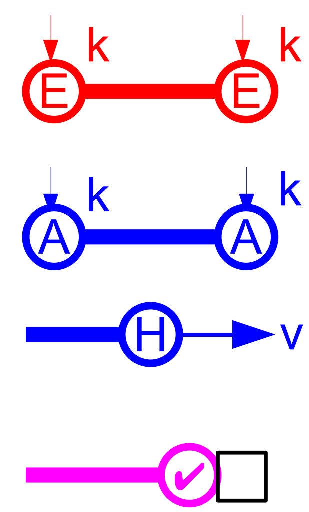
Security in Current Networked Storage Systems

- Existing technology offers little protection
  - → Originally developed for server room
  - → Coarse-grained access control
  - → Storage provider, networks, and clients are trusted
- Security is needed
  - → Storage as a commodity
  - → Networked storage to desktop (iSCSI)
- Threats
  - - physical access to disks
  - - access to network
  - - authorized machines
  - - unauthorized machines
  - ...

Design Options for Security

Security Toolbox

- Goals
  - Confidentiality (no unauthorized access)
  - Integrity (no unauthorized modification)
  - Availability
- Security mechanisms
  - Encryption
  - → Confidentiality based on shared key k
  - Message-authentication code (MAC)
  - → Integrity based on shared key k
  - Hashing and digital signatures
  - → Integrity, w.r.t. reference value v
  - Access control
  - → Confidentiality, integrity, availability



- Any mechanism may be applied on any layer

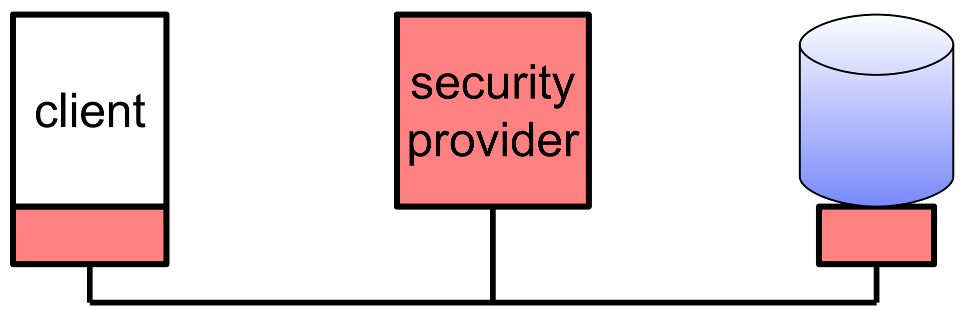
Any Security Mechanism May Be Applied on Any Layer

- Storage systems have these layers for good reasons
  - → Not all security mechanisms are useful and efficient on all layers
  - → Challenge is to select the “right” combination
- Some representative examples are presented

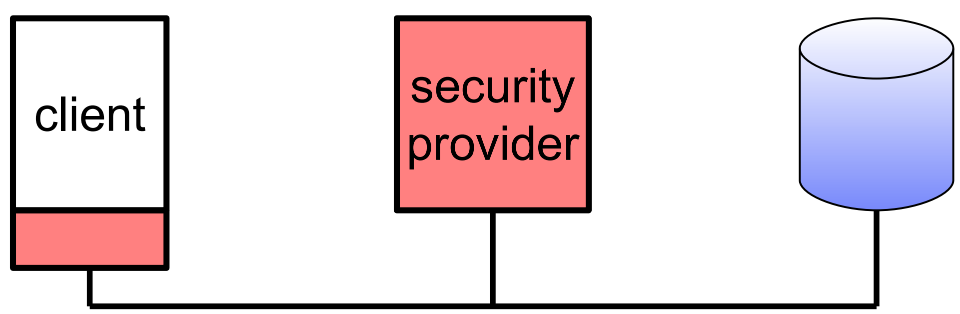
	Ⓔ	Ⓐ	✓
file	key mgmt. & lazy revocation	hash trees & fork-linearizability	
object			OBS security protocol
block	block-cipher modes & IEEE P1619	hybrid block-integrity protection	

Generic Model for a Secure Storage System

- Option 1: Protect data in flight
  - → Trusted client, trusted storage (untrusted network)



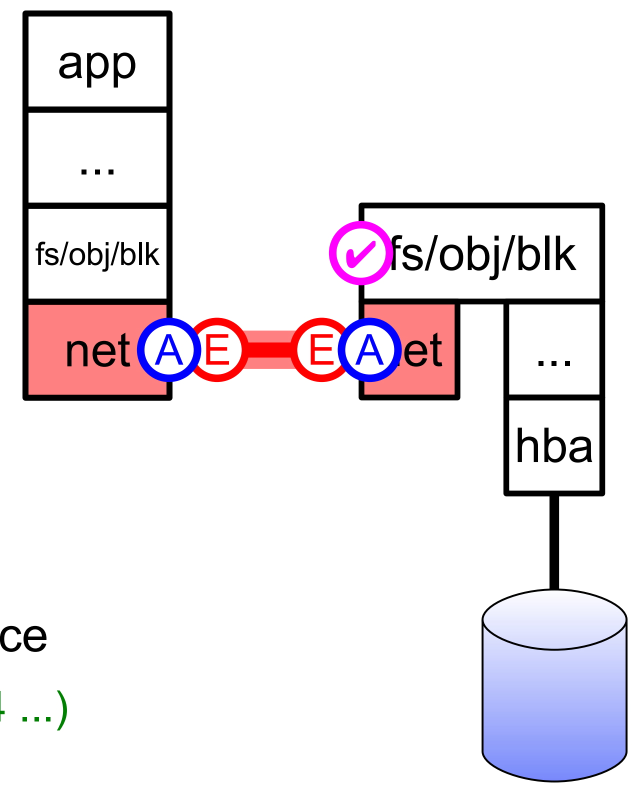
- Option 2: Protect data at rest
  - → Trusted client (untrusted storage and untrusted network)
  - → Allows DoS attack, data may be lost



Security for Networked Storage Systems (1)

Option 1: Protect the data in flight

- ✔ Access control
- Ⓐ Authentication/integrity protection
- Ⓔ Encryption

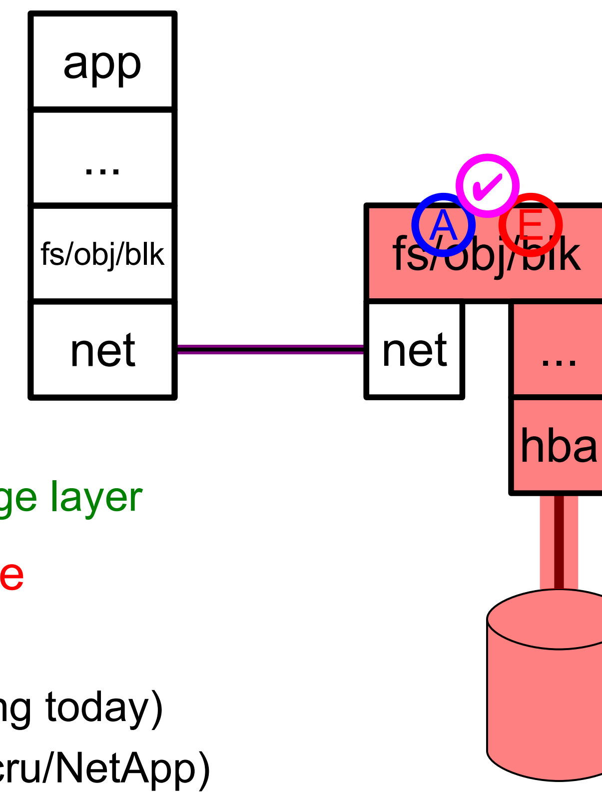


- Encrypt the communication
  - → Session, transport or packet layer
  - → Secure RPC, SSL, IPsec, FC-SP ...
- Layer-specific access control on storage device
  - → NAS at filesystem layer (exists in AFS, NFSv4 ...)
  - → ObjectStore at object layer (in standard)
  - → SAN at block layer (proposed)

Security for Networked Storage Systems (2)

Option 2: Protect the data at rest

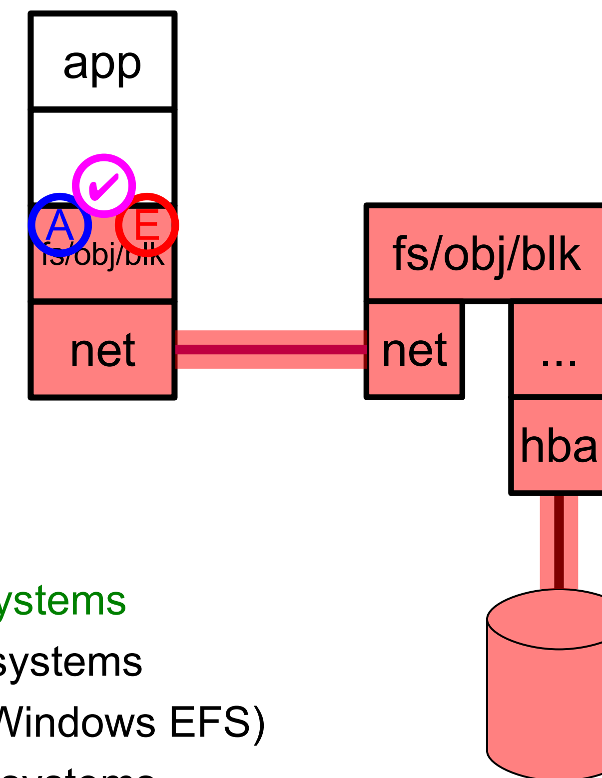
- ✔ Access control
- (A) Authentication/integrity protection
- (E) Encryption



- Encrypt the storage space
  - → Encryption and integrity protection for a storage layer
- Layer-specific cryptography on storage device
  - → Typically on low layers: block encryption
    - - In tape and disk storage devices (emerging today)
    - - As separate appliance (existing, e.g., Decru/NetApp)

Security for Networked Storage Systems (3)

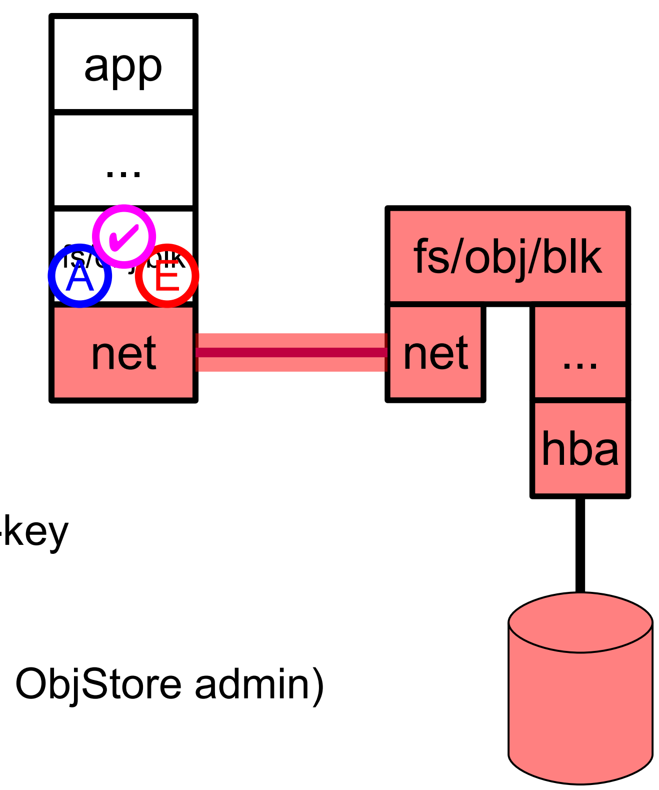
Combining Options 1 & 2:
Protecting data in flight & at rest



- Encrypt the storage space
  → But don't trust the network
  and don't trust the storage device
- Layer-specific cryptography on client
  → Typically on higher layers: cryptographic filesystems
    - Available today in local cryptographic filesystems
      (CFS, SFS, Linux loopback encryption, Windows EFS)
    - Not yet widely available for distributed filesystems

Design Dimensions

- Encryption: keys?
  - Separate security admin server
  - Encrypted with user/group public key
  - Held by hardware module
- Integrity verification: reference values?
  - Integrated in directory
  - Inode tree is hash tree
  - Digital signatures under user/group public-key
- Access control: credentials?
  - Separate security admin server (Kerberos, ObjStore admin)



Outline of Presentation

- Storage systems have these layers for good reason
  - → Not all security mechanisms are useful and efficient on all layers
- Challenge is to select the “right” combination

	(E)	(A)	✓
file	key mgmt. & lazy revocation	hash trees & fork-linearizability	
object			OBS security protocol
block	block-cipher modes & IEEE P1619	hybrid block-integrity protection	

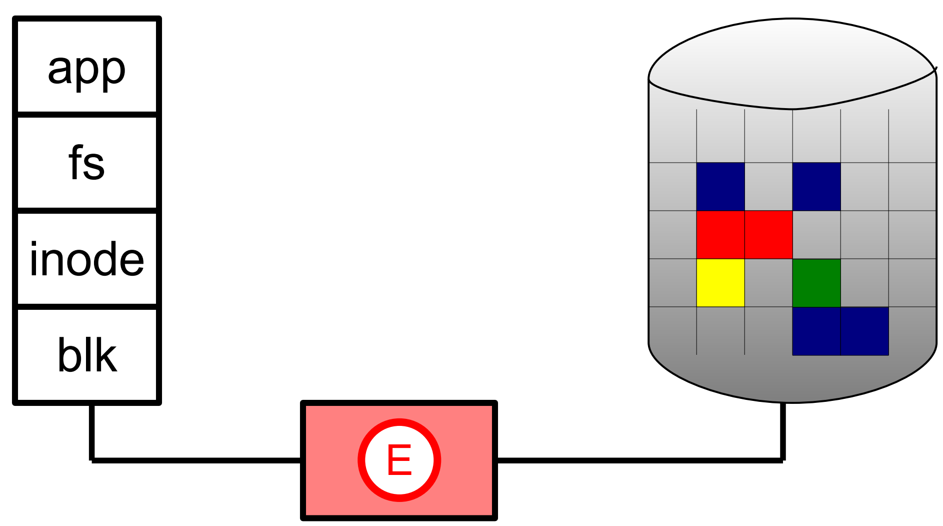
presentation order ↑

Block Layer

- Tweakable encryption and other block-cipher modes
- Hybrid block-integrity protection
- Authenticated record-encryption

Encryption at the Block Layer

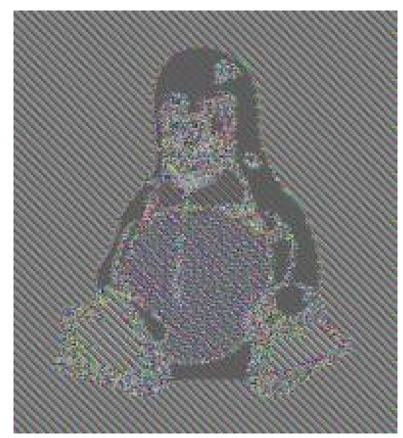
- “Sector” encryption, 512-byte blocks
- Transparent to storage system → no extra space available for chaining mode



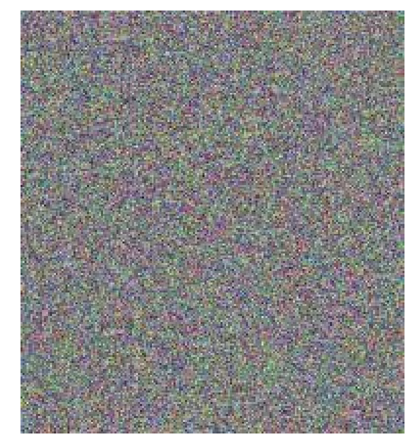
- IEEE SISW standardization effort: P1619, P1619.1, P1619.2, ...

Why a Block-Cipher Mode of Operation?

Plaintext bitmap picture

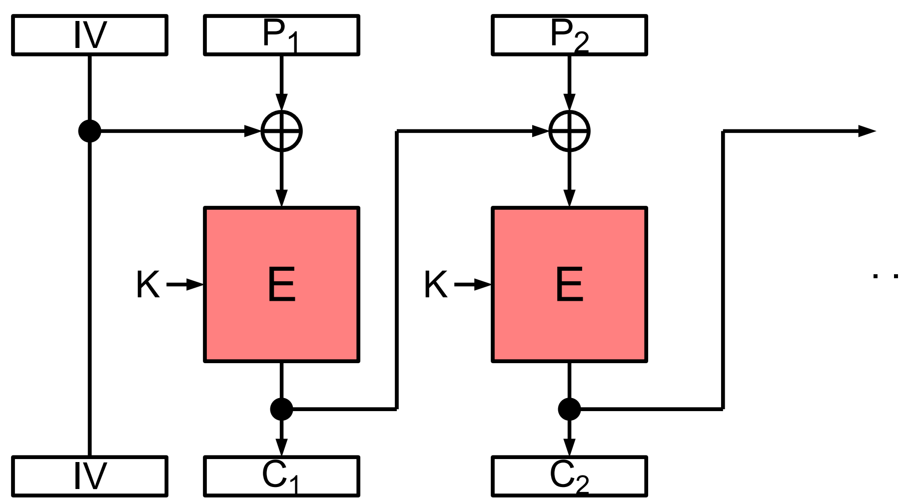


Encrypted in ECB mode



Encrypted in secure chaining mode

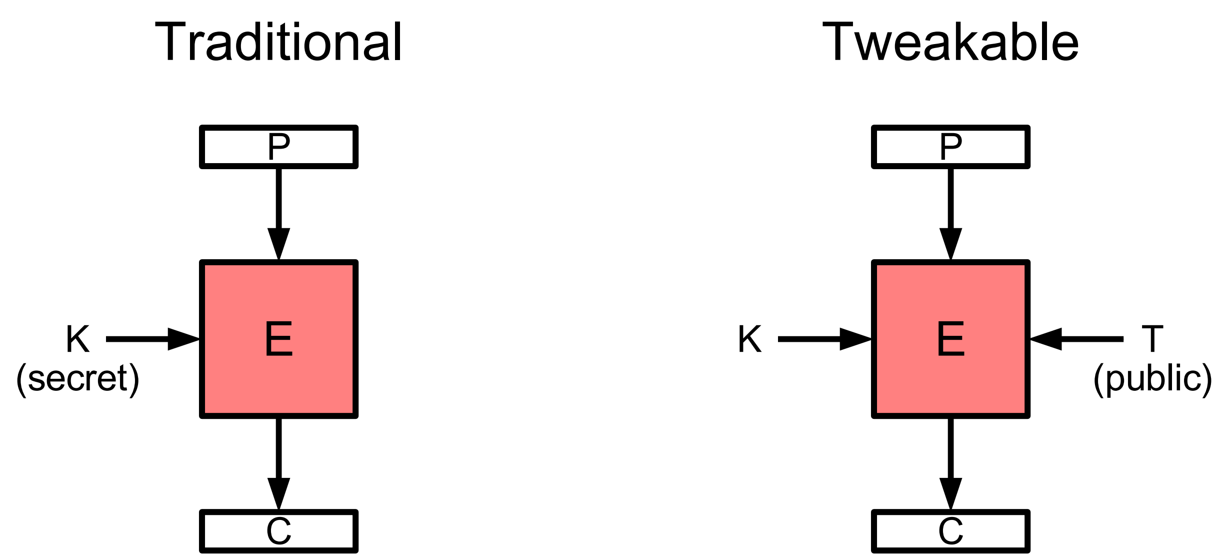
Using CBC Mode



- IV chosen at random → must be stored (but there is no room)
- Derive IV from offset of sector on disk

  IV = E_K(disk id || sector offset)
- Leaks location of first updated block within sector (a passive attack)
- Active attack possible if adv. can decrypt some sectors but not others

Tweakable Block Encryption [LRW02]

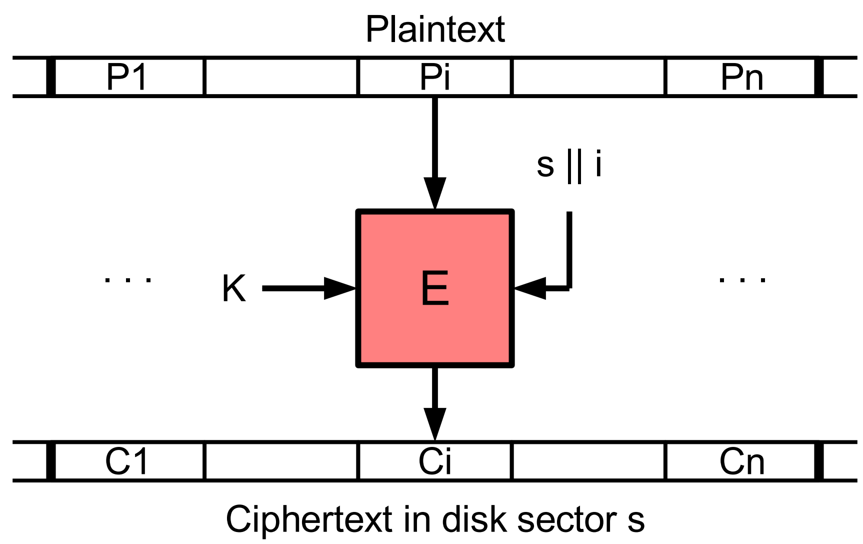


$E_K()$ is PRP

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

- $E_K()$ is a pseudo-random permutation (deterministic after picking K)
  - → Change even one bit of C to C' → decrypted P' totally independent of P
  - → But the same permutation in every instance
- Tweakable $E_{K,T}()$ is a family of independent permutations (indexed by T)
  - → T = address of block

Narrow-block Tweakable Encryption

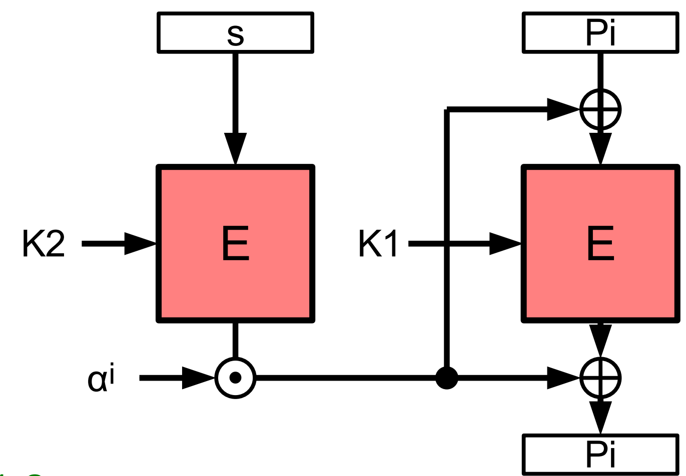


Tweaked block = cipher block

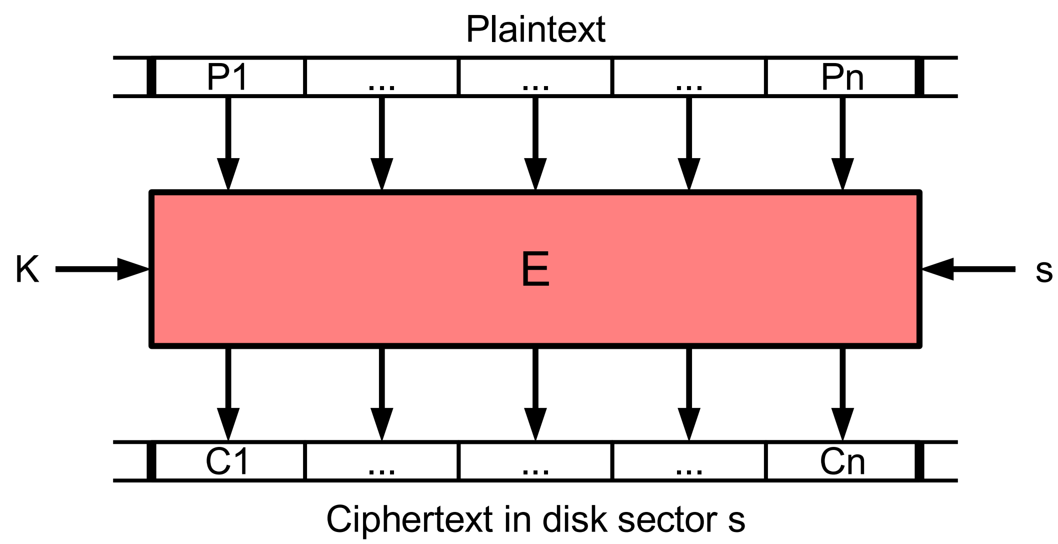
- All blocks of sector encrypted independently (unlike CBC)
- Tweak is sector s plus block index i
- Leaks only location of updated blocks within sector

Narrow-block Tweakable Encryption Scheme

- XTS-AES mode based on XOR-Encrypt-XOR (XEX) [R04]
- Tweak = sector s || block index i
- Key K = K1 || K2
- Arithmetic in GF(2^{128})
  - → α is primitive element in GF(2^{128})
  - → α^i computation is efficient for i=0,1,2...
- XTS-AES is standardized by IEEE P1619 (almost final)



Wide-block Tweakable Encryption

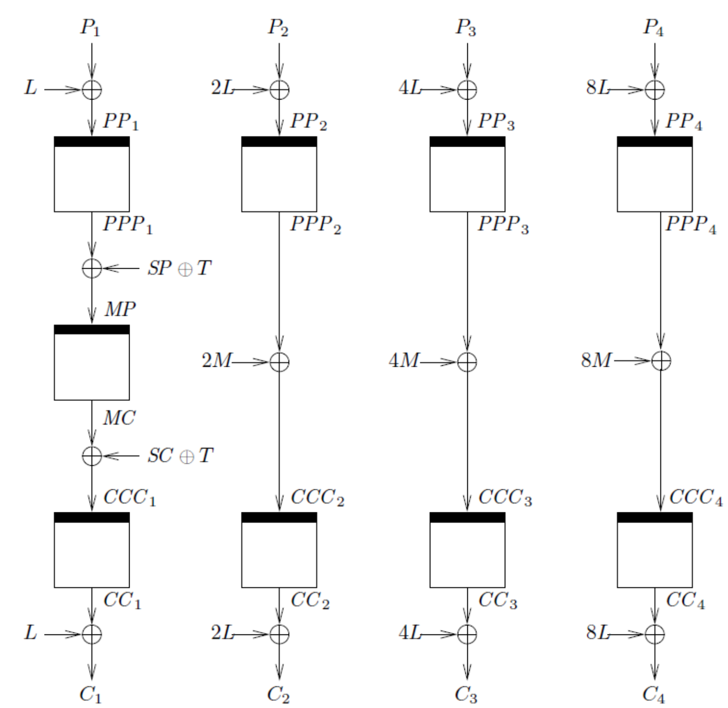


Tweaked block
=
disk sector

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

Wide-block Tweakable Encryption Scheme

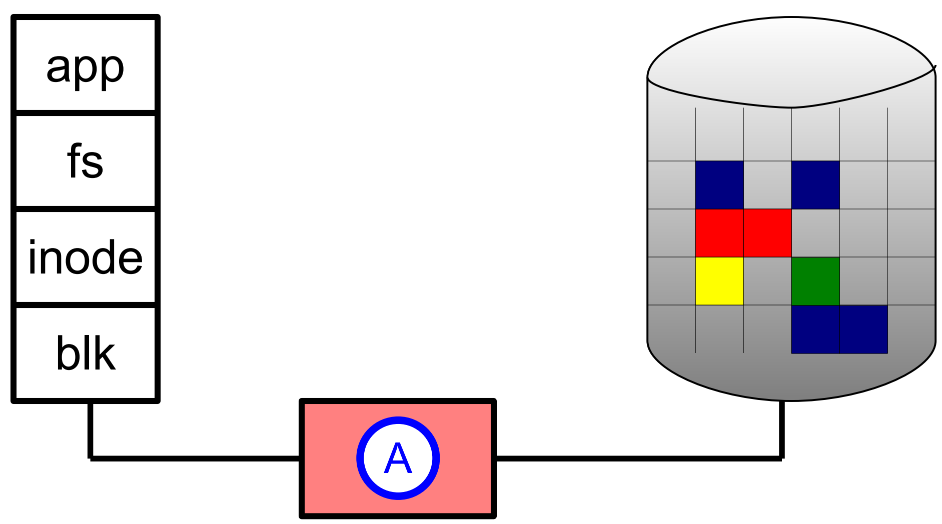
- EME mode [HR04],
  calls to E are parallelizable:



- EME requires ≈2 block cipher calls per plaintext block
  - → Considered too costly by many
- IEEE P1619.2 standardization (far from final)

Integrity Protection at the Block Layer

- No extra space available → really problematic for integrity
- All integrity protection and data authentication methods require extra space for a tag or a hash value



- If there was space, use a MAC or a hash tree (see later) ...

Hybrid Integrity Protection at the Block Layer [ORY05]

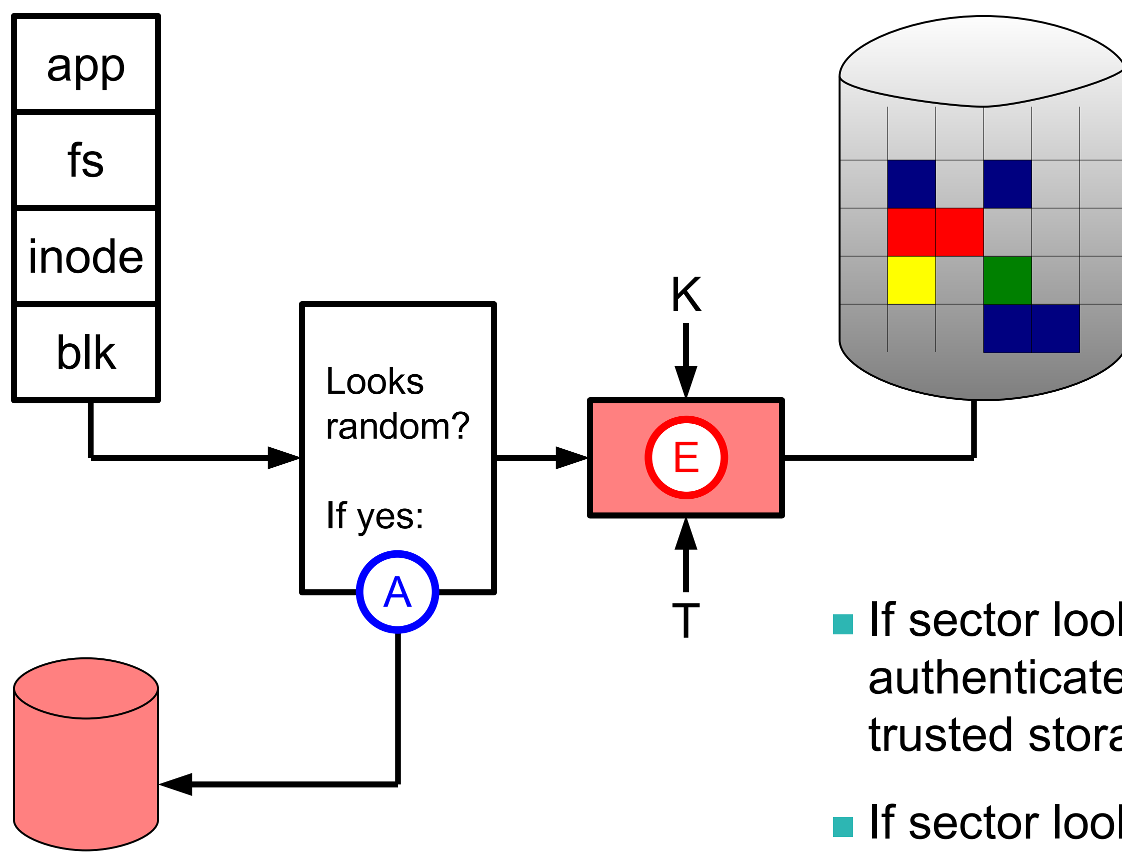
- Data is encrypted
- Use tweakable encryption mode on wide block (sector of 512B)
- Idea:

  If data contains redundancy, then any modification of ciphertext is detectable because decrypted plaintext will look random.

  → “Redundant” sectors are not extra protected for modification detection
  → “Random” sectors are protected in traditional way

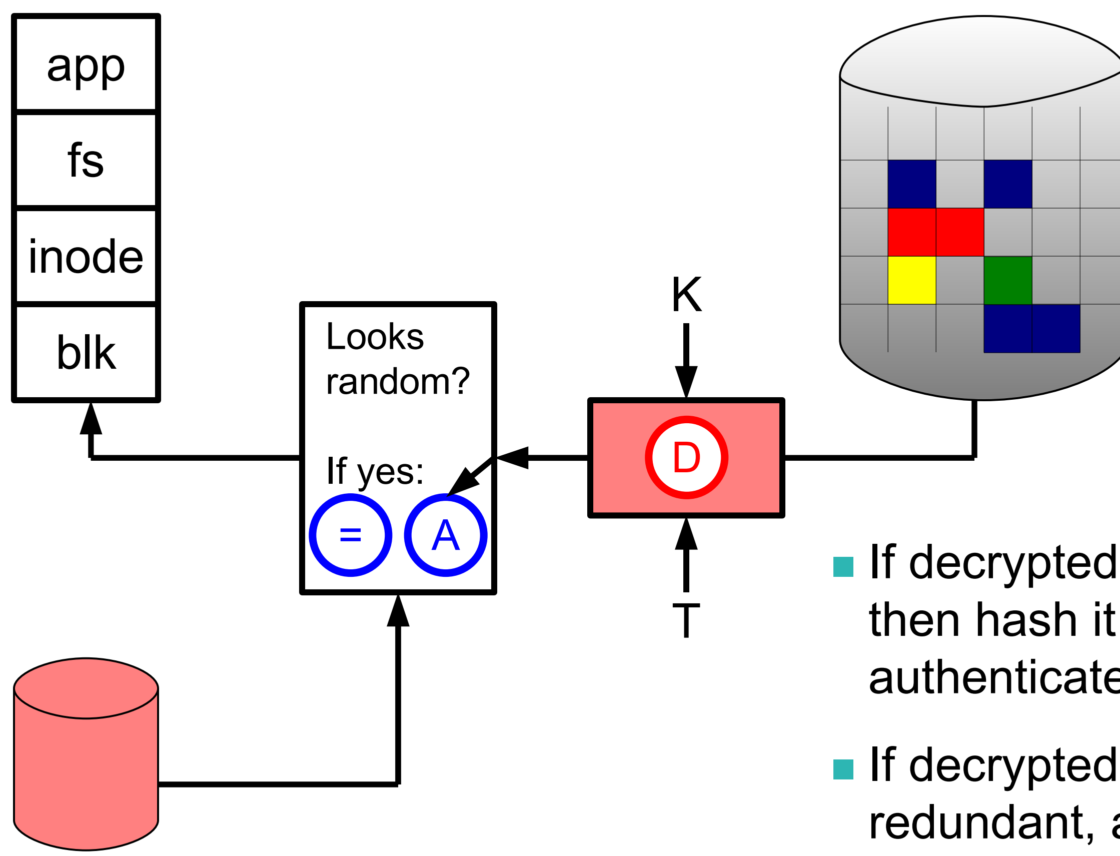
- Needs a heuristic test for “redundancy” or “randomness” in a sector

Writing Data



- If sector looks random, authenticate it using extra trusted storage.
- If sector looks redundant, just store it.

Reading Data

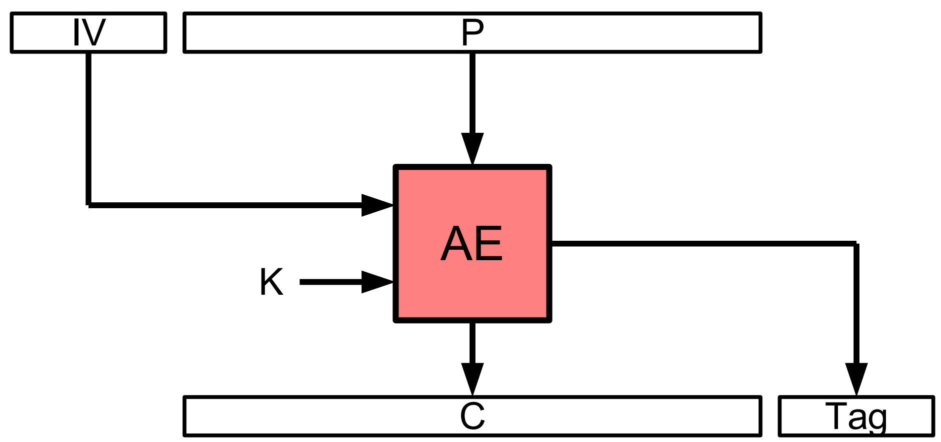


- If decrypted sector looks random, then hash it and compare it with authenticated value.
- If decrypted sector looks redundant, accept it as authentic.
  - → Allows replay attack with previous content of sector.

Discussion of Hybrid Scheme

- Performance depends on payload data
- Suffers from replay attacks
- Depends on estimator for redundancy
  - → Simple 1-st order entropy test on 8-bit blocks in 1024-byte sector
    - Threshold set to 7.7 bits
    - 98% of blocks from filesystem trace have observed entropy < 7.7
  - → Saves 98% storage space compared to hashing every block
    (Or: protects integrity of 98% of observed data.)
- Cannot achieve ideal security for arbitrary payload

Authenticated Record-Encryption



Authenticated Encryption (AE)

- AE combines encryption and authentication (MAC) in one pass

  AE(K, IV, P) → (C, Tag)

  AE^{-1}(K, C, Tag) → P / "FAIL"

- Length-expanding → suitable for tape, but not for disk

Authenticated Record-Encryption Standards

- IEEE P1619.1 has standardized four authenticated encryption schemes:

  CCM-128-AES-256

  → Counter mode encryption with CBC-MAC
  using AES-256 with 128-bit wide CBC-MAC (used by Sun)

  GCM-128-AES-256

  → Galois/counter mode encryption
  using AES-256 with 128-bit wide tag (used by IBM, LTO)

  CBC-AES-256-HMAC-SHA-*

  → CBC mode encryption with HMAC
  using AES 256 and SHA-*

  XTS-AES-256-HMAC-SHA-512

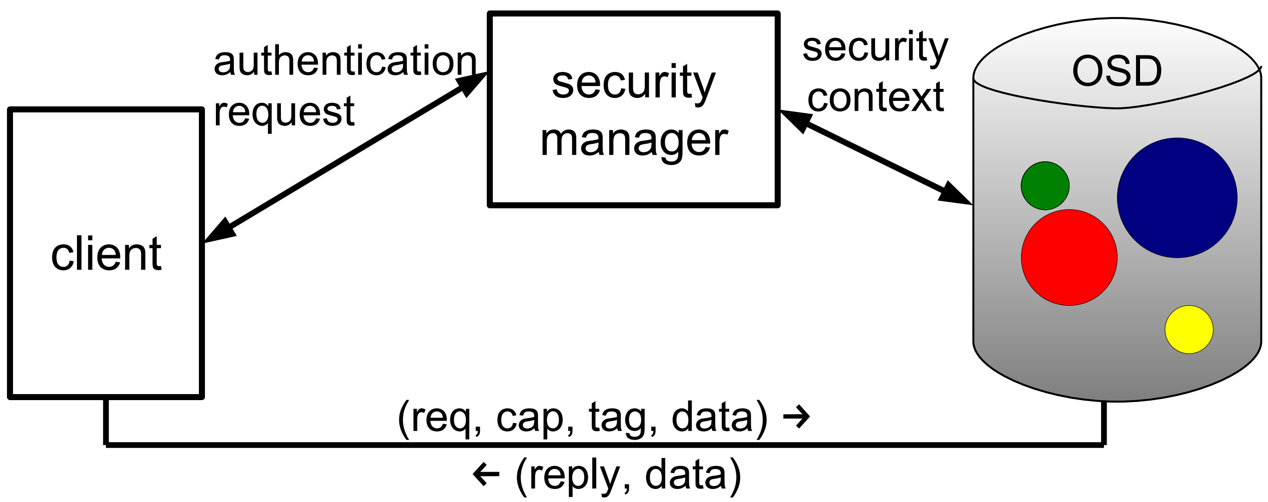
  → XTS narrow-block tweakable encryption (P1619.1) with HMAC
  using AES 256 and SHA-512

- Standard status is final, adoption by industry is guaranteed

Object Layer

- Capabilities in Object Storage

Object Store Security Protocol [ACF+02, FNN+05]



- Capability-based protocol to authenticate requests and traffic between client and object-storage device (OSD)
- Key establishment protocol between OSD and security manager
- Protocol between client and security manager specific to filesystem

Protocol Features

- Security methods

  NONE: --

  CAPKEY: authenticate requests at OSD level, no transport security

  → tag computed only over capability

  CMDRSP: above plus transport integrity for request and reply

  → tag computed over capability and request

  ALLDATA: above plus transport integrity for payload data

  → tag computed over capability, request, and data

- May replace IPsec for iSCSI or FCsec for Fibre Channel (also duplicates some of their functionality)

OSD Data Types

- Object hierarchy

  OBS → Partition → Object

- Key hierarchy

  Master key: to initialize OSD and create root key

  Root key: to manage partitions and their keys

  Partition key: only to create per-partition working key

  Working key: per partition, changed frequently, useful for revocation (among other uses), protects all objects in partition

OBS Security Protocol Details (CAPKEY)

- PRF F
- Capabilities

  (obj, exptime, permissions, nonce)

- Client requests credential from security manager and receives

  cred = (cap, Kcap)

  where Kcap = F_K(cap) under appropriate partition's working key K

- Client sends

  (req, cap, tag)

  to OSD, with a unique channel id (or nonce) chosen by the OSD, and

  tag = F_{Kcap}(cap || channel id)

- OSD verifies that

  1. req is an allowed operation by cap for this partition
  2. validates tag from channel id, using key K' = F_K(cap) with its working key K of current partition

File Layer

- Designs for key management
- Encryption using lazy revocation and key updating
- Integrity protection in filesystems
- Consistent access to untrusted storage

Key Management in Cryptographic Filesystems

- Two approaches

  On-line and centralized

  - Only symmetric-key crypto
  - Simple and efficient
  - Limited scope and scalability
  - Ex. eCryptfs (as in Linux Kernel 2.6.19), Cryptographic SAN.FS [PC07] ...

  Off-line and de-centralized

  - Requires public-key crypto
  - Complex, computationally expensive
  - Scalable
  - Ex. SFS [FKM02], Windows EFS, Plutus [KRS+03], Sirius [GSMB03] ...

De-centralized Key Management

- Users have SK/PK pair
- Groups have SK/PK pair; every member of group knows SK
- Files encrypted using FEK with block cipher
- Confidentiality: Store FEK encrypted in meta-data
  - → Encrypted under every PK of every user/group that has access

Example: File X, encrypted with FEK_X

```
owner: A, rwx, E_{PK_A}(FEK_X),
group: G, rw-, E_{PK_G}(FEK_X),
world:    ---
```

- Integrity: Add FSK_X / FVK_X, key pair for digital signatures, to X
  - → Store FSK like this in every encrypted file
- Drawback: key revocation is tedious

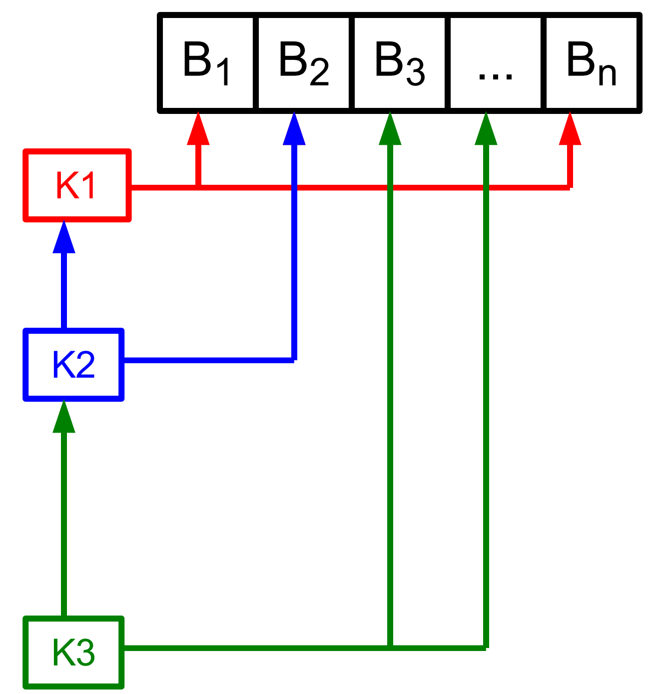
Key Revocation

- User revoked ➔ change all keys that were known to user
  - → Re-encrypt all data with fresh keys
- Expensive and disruptive operation
- Idea: Lazy Revocation [F99]
  - → Re-encrypt data only when it changes after revocation, keep old keys around.
- All versions of a key must remain accessible

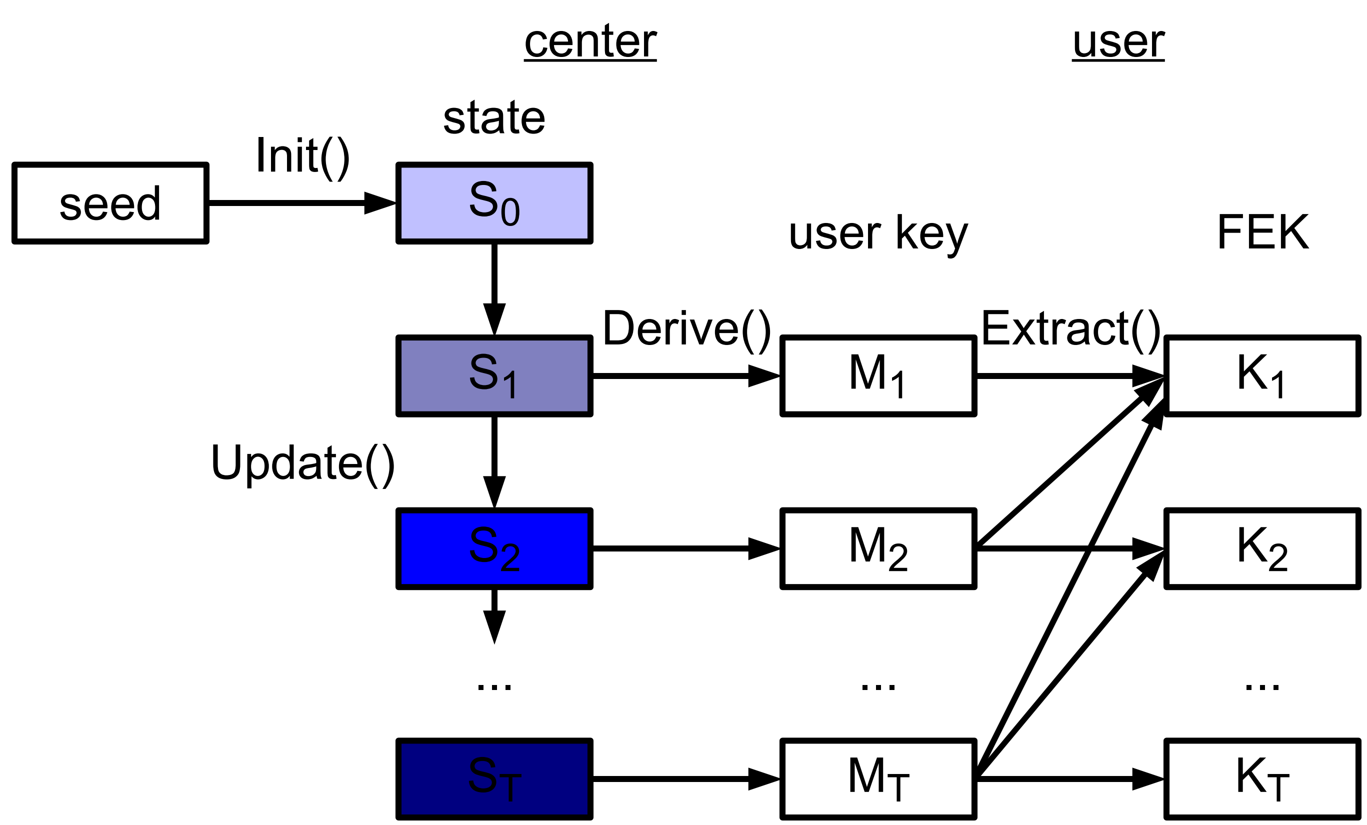
Lazy Revocation [KRS+03]

time	center	users: u	u'	u''
	S1(K1)	K1	K1	K1
u' is revoked				
	S1(K1, K2)	K2	--	K2
u'' writes B_2				
u'' is revoked				
	S1(K1, K2, K3)	K3	--	--
u writes B_3 ...				

storage



Key Updating Schemes for Lazy Revocation



- Requirements
  - → User can obtain K_1 ... K_t from M_t
  - → Adversary with M_t cannot distinguish K_{t+1} from uniformly random string

Formalization [BCO05, BCO06, FKK06]

- Key updating scheme for T periods

  KU_T = (Init, Update, Derive, Extract)

- Metrics of interest
  - → Time of Update(), Derive(), and Extract()
  - → Size of center state S_t
  - → Size of user key M_t

Composition of Key Updating Schemes [BCO06]

- Addition

  $KU^1_{T1} \oplus KU^2_{T2} \;=\; KU^{\oplus}_{T1+T2}$

  Construction

  → First T1 intervals use KU^1

  → Subsequent T2 intervals use KU^2 and include M_{T1} in user key

- Multiplication

  $KU^1_{T1} \otimes KU^2_{T2} \;=\; KU^{\otimes}_{T1 \cdot T2}$

  Construction

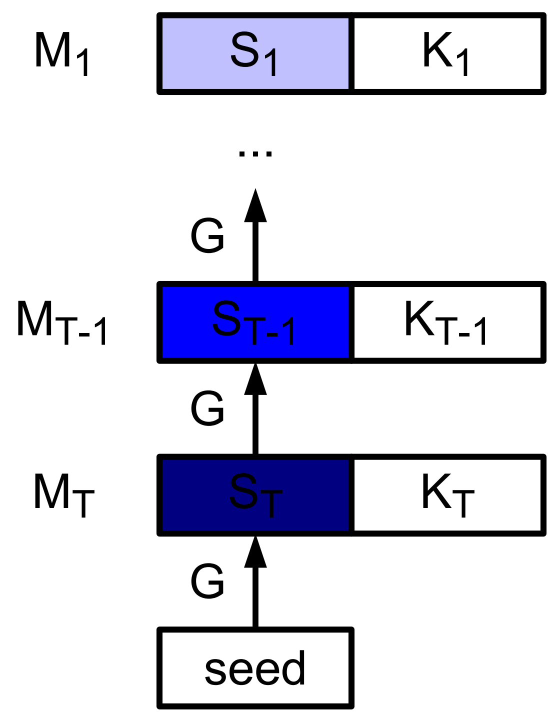
  → Every key generated with KU^1 is used to seed an instance of KU^2

Constructions

- Chaining construction
- Trapdoor permutation-based
- Tree construction

Chaining Construction ("Hash Chain")

- Using pseudo-random generator G

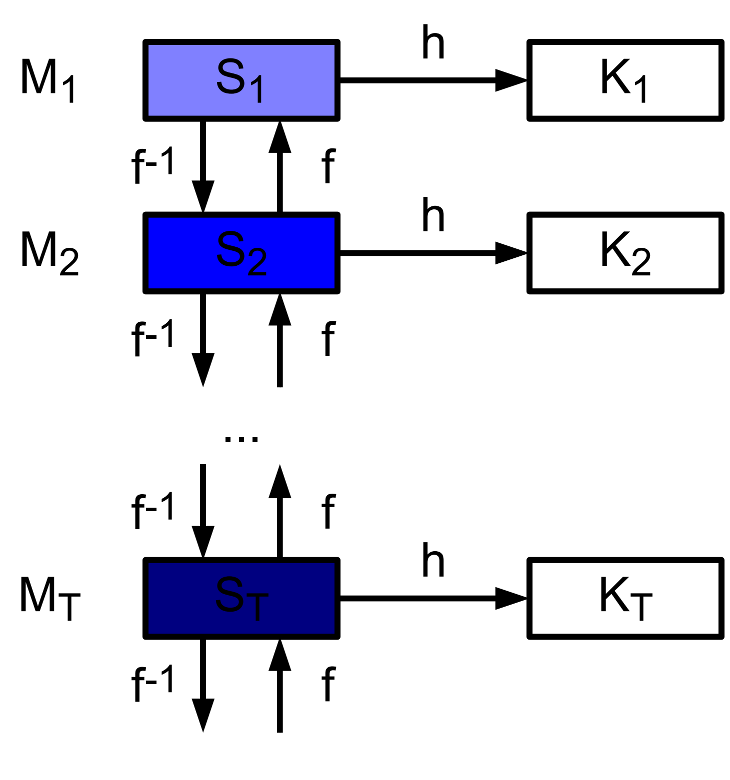


State	Update	Derive	Extract
seed	0	O(T) PRG	O(T) PRG

- Drawback: Fixed T

Trapdoor Permutation Construction [KRS+03]

- Using a trap-door permutation f, f^{-1} (TDP), where f is easy and f^{-1} is hard without private key, hash function h() in Random-Oracle Model

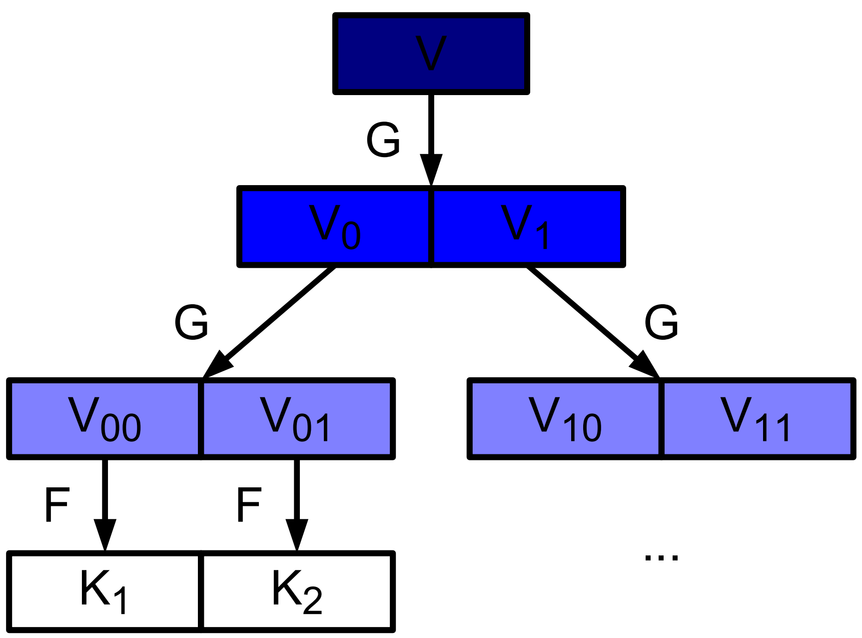


State	Update	Derive	Extract
seed	TDP	const.	O(T) TDP

Advantage: Flexible T
Drawback: Public-key operations

Tree Construction [BCO06]

- Using pseudo-random generator G and pseudo-random function F



State	Update	Derive	Extract
O(log T)	O(log T) PRG	0	O(log T) PRG

Advantages:
- Symmetric-key operations
- Practically unbounded T

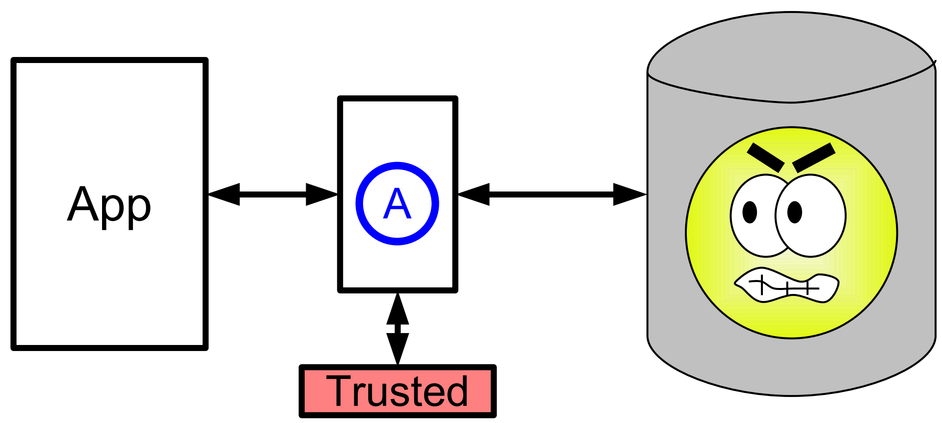
- User key M_t is smallest set of nodes needed to derive K_1 ... K_t
- T fixed, but practically unbounded, as cost is logarithmic in T

Comparison of Key Updating Schemes

- Trapdoor scheme using RSA-1024
- PRF/PRG using AES-128
- Average times [ms] measured on Intel 2.4 GHz Xeon

Scheme	T	Derive + Update	Extract
Chaining	1024	1.28	1.24
Trapdoor	1024	15.4	15.2
Tree	1024	0.015	0.006
Tree	2^{16}	0.015	0.008
Tree	2^{25}	0.015	0.01

Integrity Protection in Filesystems



- Storage consists of n data items x_1, ..., x_n (entries in list, blocks of file ...)
- Applications access storage via integrity checker
  - → Checker uses small trusted memory to store short reference value v (i.e., together with encryption key in meta-data)
- Integrity checker operations
  - → Read item and verify w.r.t. v
  - → Write item and update v accordingly

Implementing an Integrity Checker

- Use hash function H to compute v? $v = H(x_1 \| \ldots \| x_n)$
  - → Infeasible for long files
  - → No random access to item
- Use a secret key with a MAC?
  - → Suffers from replay attacks
- Well-known solution: Hash tree [Merkle 79]
  - → Overhead of read/verify and write/update is logarithmic (in n)
- Recent alternatives

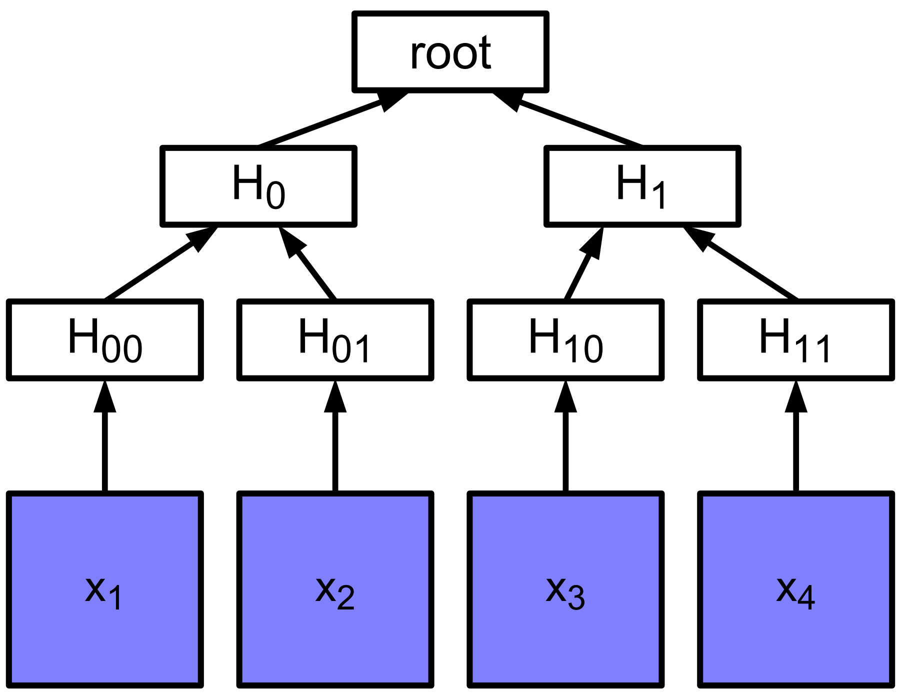
  Dynamic accumulators [CL02]

  → Overhead of read/verify is constant

  Incremental hashing [BM97,CDDGS03]

  → Overhead of write/update is constant

Hash trees for Integrity Checking [Merkle 79]



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

Dynamic Accumulator for Integrity Checking

- An accumulator is a cryptographic abstraction for collecting data values and checking their presence:

  Init() → (a, k) -- generates authenticator/accumulator value a and key k

  Add(a, i, x_i, k) → a' -- adds x_i to accumulator at position i

  Update(a, i, x_i, k) → a' -- updates accumulator at position i to x_i

  Witness(a, i, x_i, k) → w -- produces a witness w for presence of x_i

  Verify(a, i, x_i, w) → "yes" / "no" -- checks if witness w is valid and proves that entry x_i was added to accumulator at position i

- Without k, it must be infeasible to forge i', x', w' that verify for given a

- Impl. with public-key crypto under strong RSA assumption [CL02]:

  → Given an RSA modulus N = P · Q (with P, Q safe primes), and r ∈ $\mathbf{Z}_N$, it is infeasible to find a, b s.t. $a^b = r \mod N$

  Accumulator a containing $x_1, \ldots, x_n$ is $a = r^{H(1||x1) \cdots H(n||xn)} \mod N$

  Witness for x_i in a is $w = a^{1/H(i||xi)} \mod N$

  Verify that x_i is contained in a by checking $w^{H(i||xi)} = a \mod N$?

Incremental Hashing for Integrity Checking

- Hash function $IH(x_1, ..., x_n)$ on n entries $x_1, ..., x_n$ that allows updates:

  Given $h = IH(x_1, ..., x_i, ..., x_n)$ and values x_i and x'_i,
  one can compute $h' = IH(x_1, ..., x'_i, ..., x_n)$ in time independent of n.

- Implementation based on number theory [BM97]:

  $IH(x_1, ..., x_n) = H(1||x_n) \cdots H(n||x_n) \bmod p$

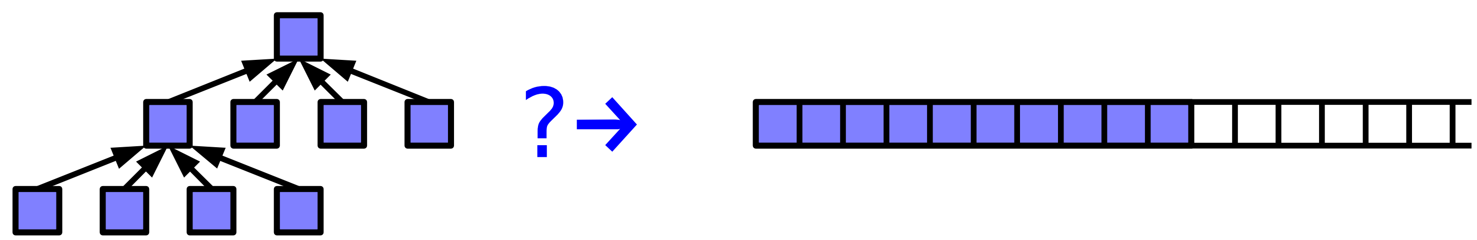
  for large prime p and ordinary hash function $H(\cdot)$

Integrity Checking Schemes Summary

Scheme	Update time	Verify time	
Hash tree	O(log n)	O(log n)	Fast, only hash operations
Accumulator	n	constant	Slow, public-key operations
Incr. Hash	constant	n	Fast, but verify is slow

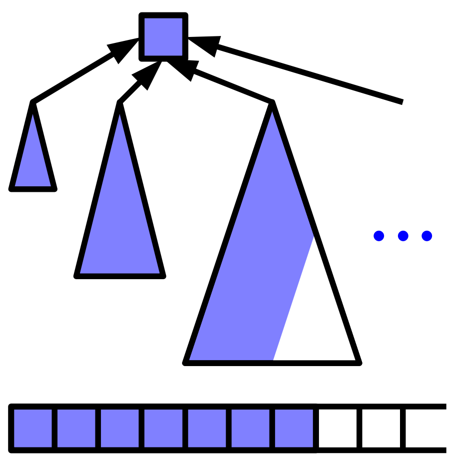
In practice, integrity checking is usually done with hash trees.

Implementing Hash Trees [L06]



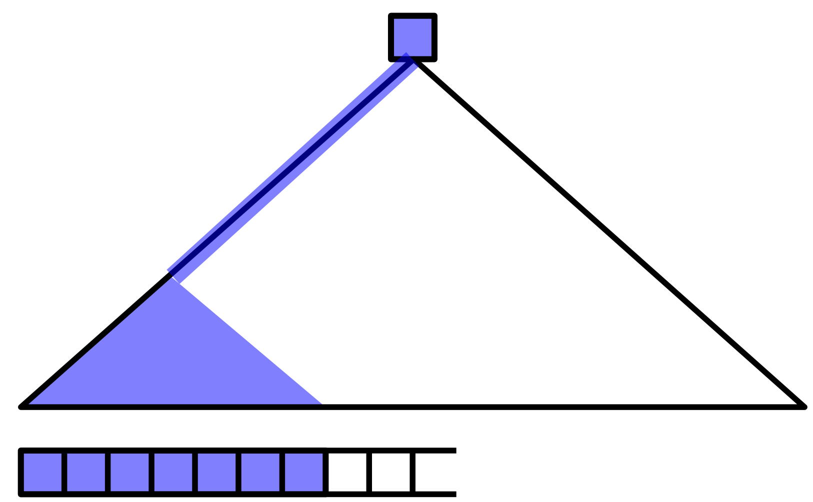
- How to serialize tree with minimal overhead?
  - Storage access should cover contiguous region
  - File may grow & shrink
- Which tree? → Topologies
- Naïve scheme? Hash only once (depth 1)

Hash Tree Topologies for Filesystems



Imbalanced tree

- Adapts to file size by “growing”



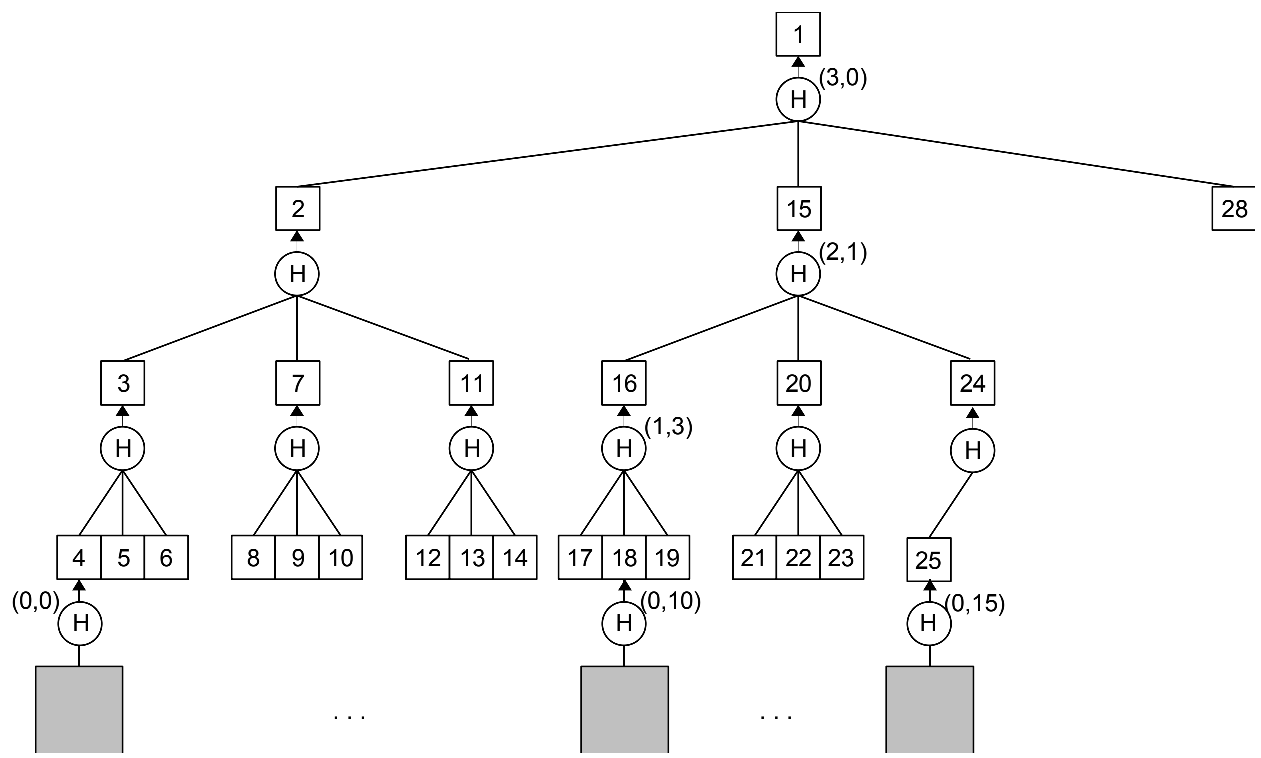
Implicit complete tree

- Sparse allocation

How to enumerate nodes?

- Breadth-first order (BFO)
- Pre-order

Pre-order Enumeration of Hash Tree Nodes [PC07]



- Implicit sparse allocation of maximum-size tree
- Typical file starting at offset 0 maps to a contiguous range
- Takes care of file holes

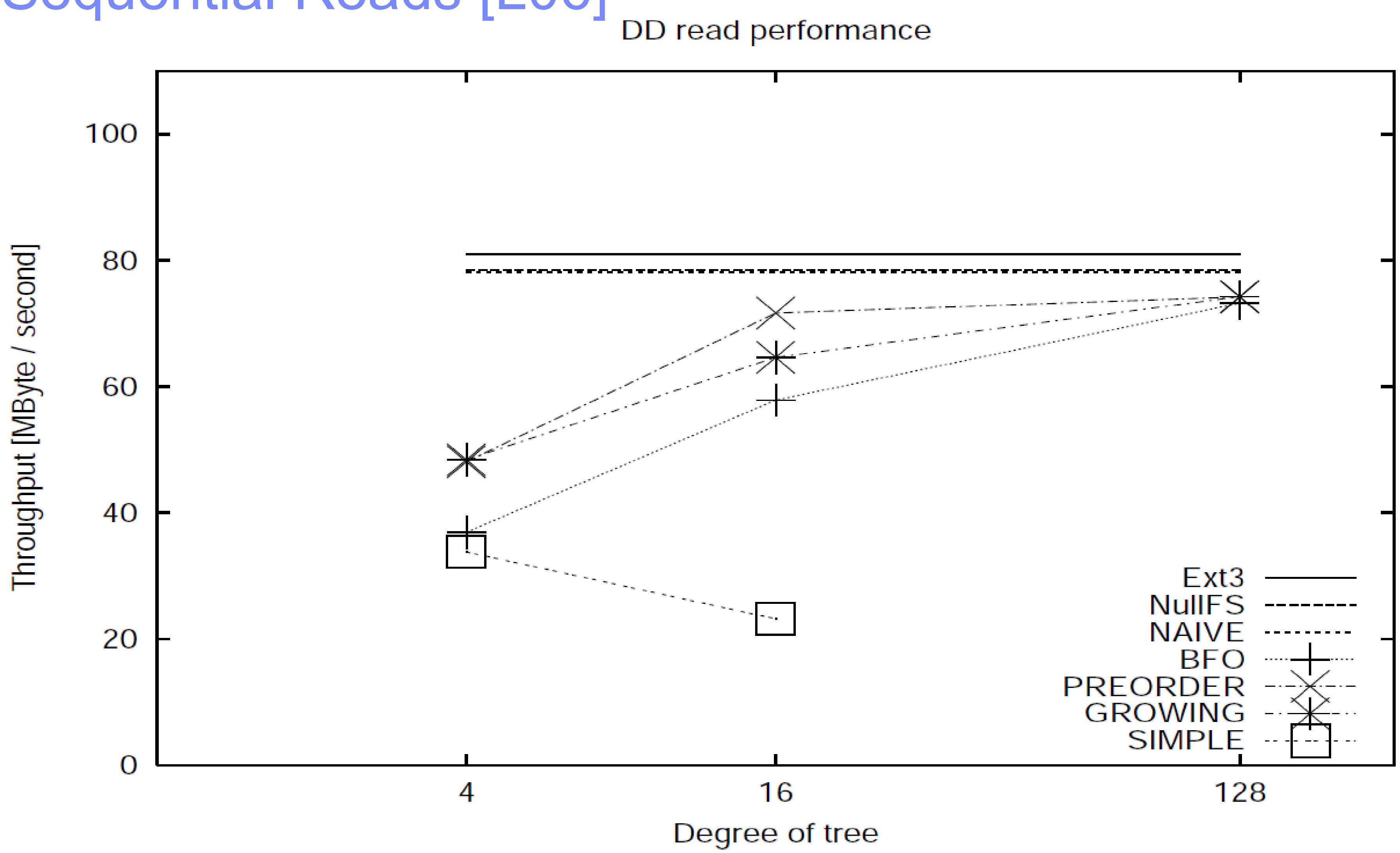
Hash Tree Implementations in Filesystems

- Ensure consistency between two mutually dependent data paths
  - → Much more complex than encryption in filesystem
- Buffer current tree-path with all siblings
  - → Sequential read & write of whole file in O(n) work (constant overhead per access)
- Cache whole tree
  - → Potentially large memory footprint
  - → Typical tree size 1‰ ... 1% of file size
- Journaling needed for crash-resilience
  - → Otherwise crash results in integrity violation
  - → Solution demonstrated only once to date [MVS00]

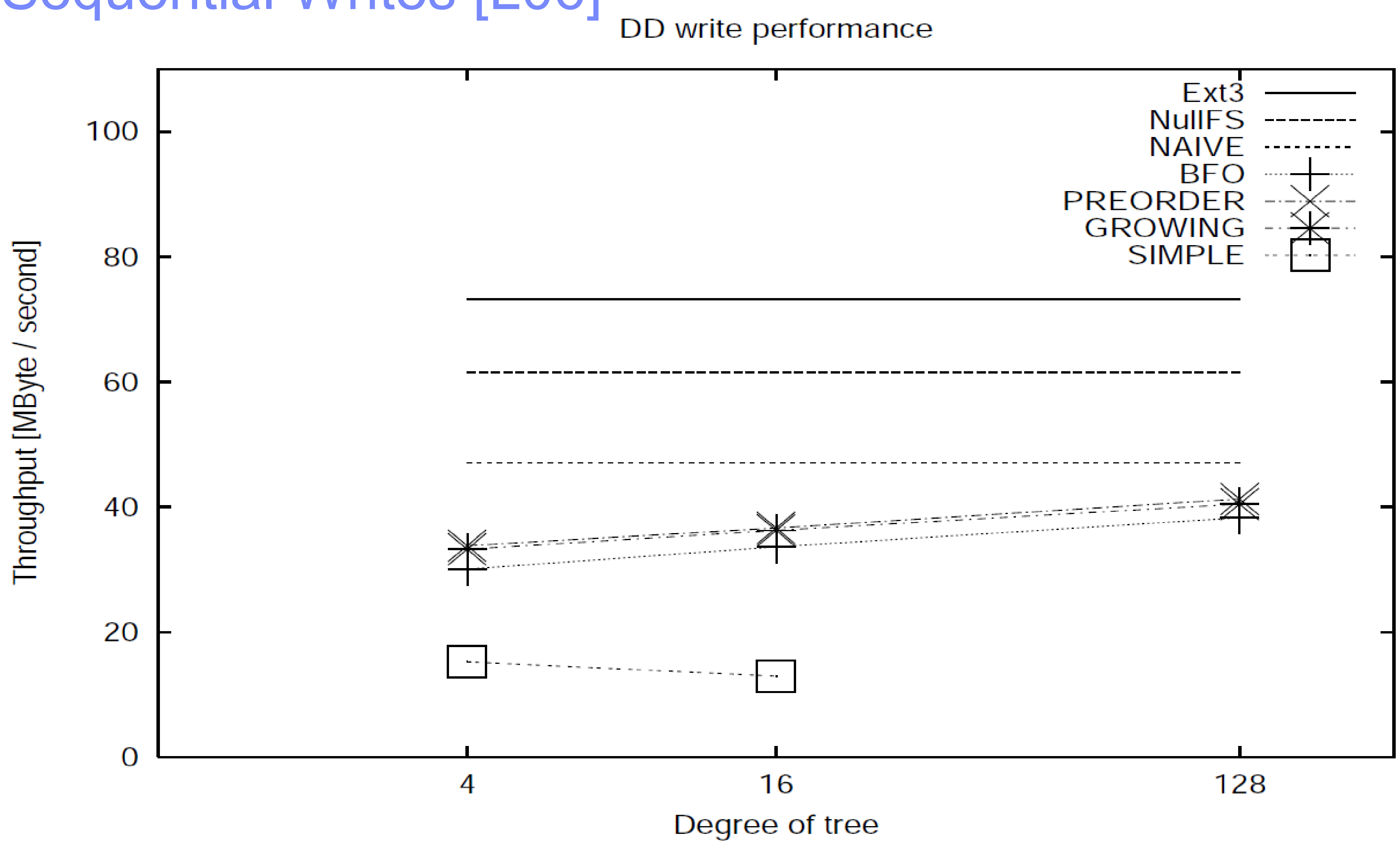
An Experimental Comparison [L06]

- Integrity-protecting virtual filesystem in Linux
  - → Kernel 2.6, user-space, with FUSE (Filesystem in USErspace)
  - → Physical filesystem was local ext3
  - → IBM x346 server, dual 3.2 GHz Xeon CUPs
    - - 3GB RAM, several 73GB IBM SCSI disks with 10k RPM
- Benchmarks
  - → Sequential reads & writes of large files (8GB, `dd`)
  - → PostMark synthetic benchmark
    - - Creates, reads, writes, deletes many 1-2 MB files
- Topologies and layouts of tree
  - → NAIVE (tree of depth 1)
  - → SIMPLE (no buffered nodes)
  - → BFO / PREORDER enumeration (incomplete trees with buffered path)
  - → GROWING (imbalanced tree with buffered paths and pre-order enum.)
  - → Degree: 4 / 16 / 128

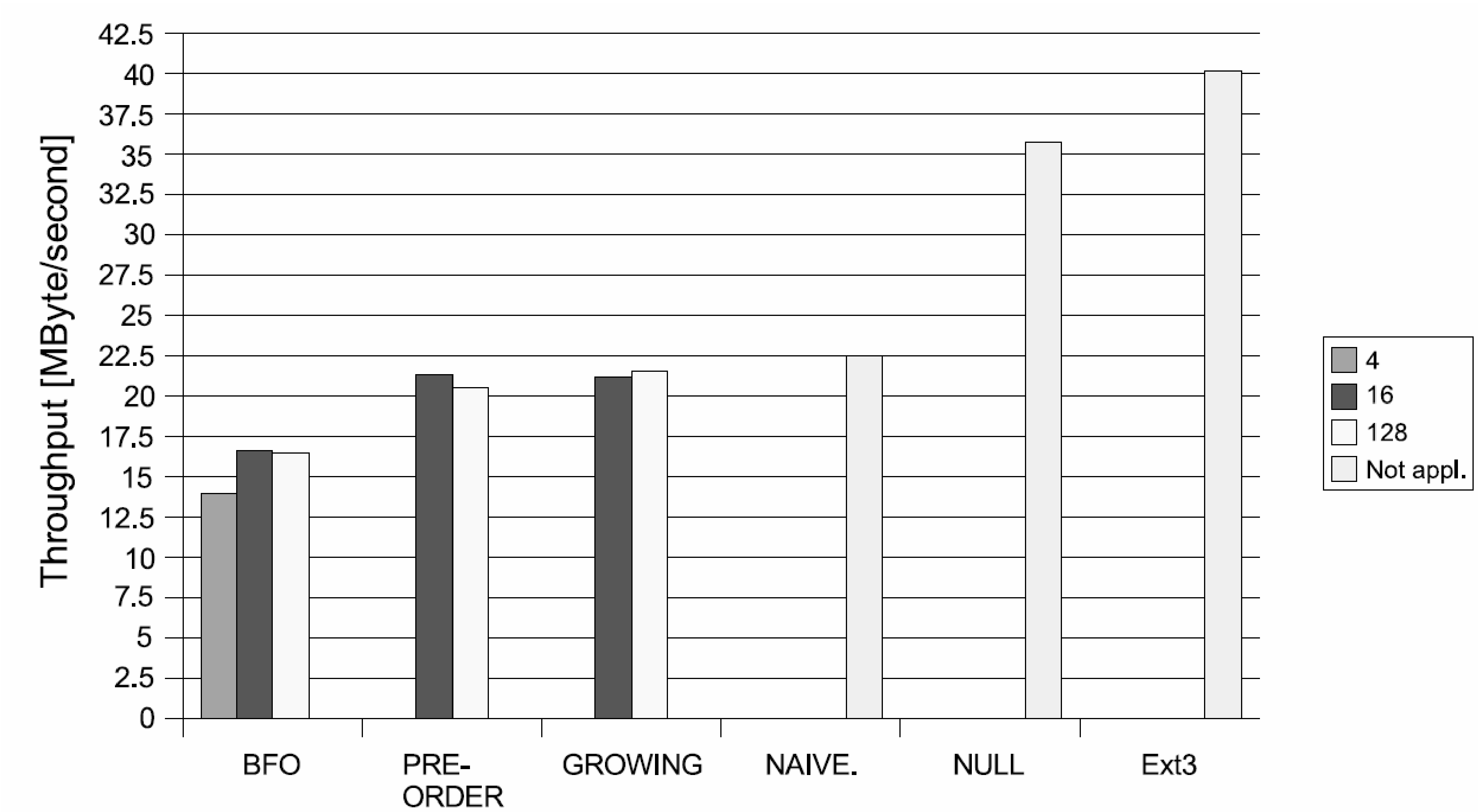
Sequential Reads [L06]



Sequential Writes [L06]



PostMark Benchmark [L06]

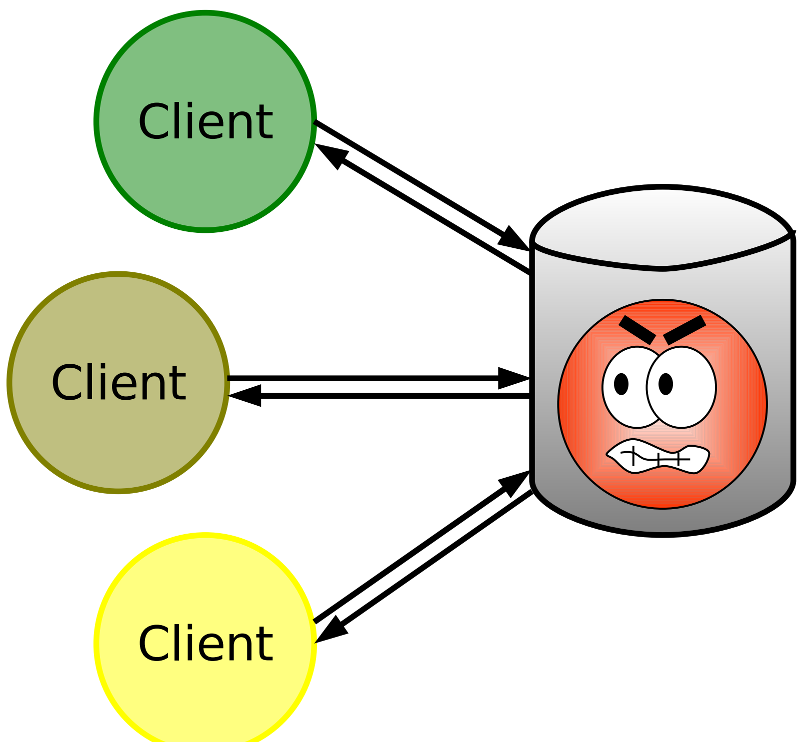


Hash Trees in Filesystems - Summary

- Naïve approach works surprisingly well here
  - → But not for first access!

- Topology and degree may vary
  - → Best determine experimentally (≈ 128)
  - → Pre-order enumeration simplifies design

Consistent Access to Untrusted Storage*



- Many independent clients
  - Correct
  - Store data on server
  - Communicate only with server
  - Small trusted memory
- Storage server
  - Untrusted
  - Potentially corrupted
- Clients read and write concurrently

How to ensure consistent view of data to all clients?

(* Advanced topic, applies to future storage systems.)

Consistent Access to Untrusted Storage

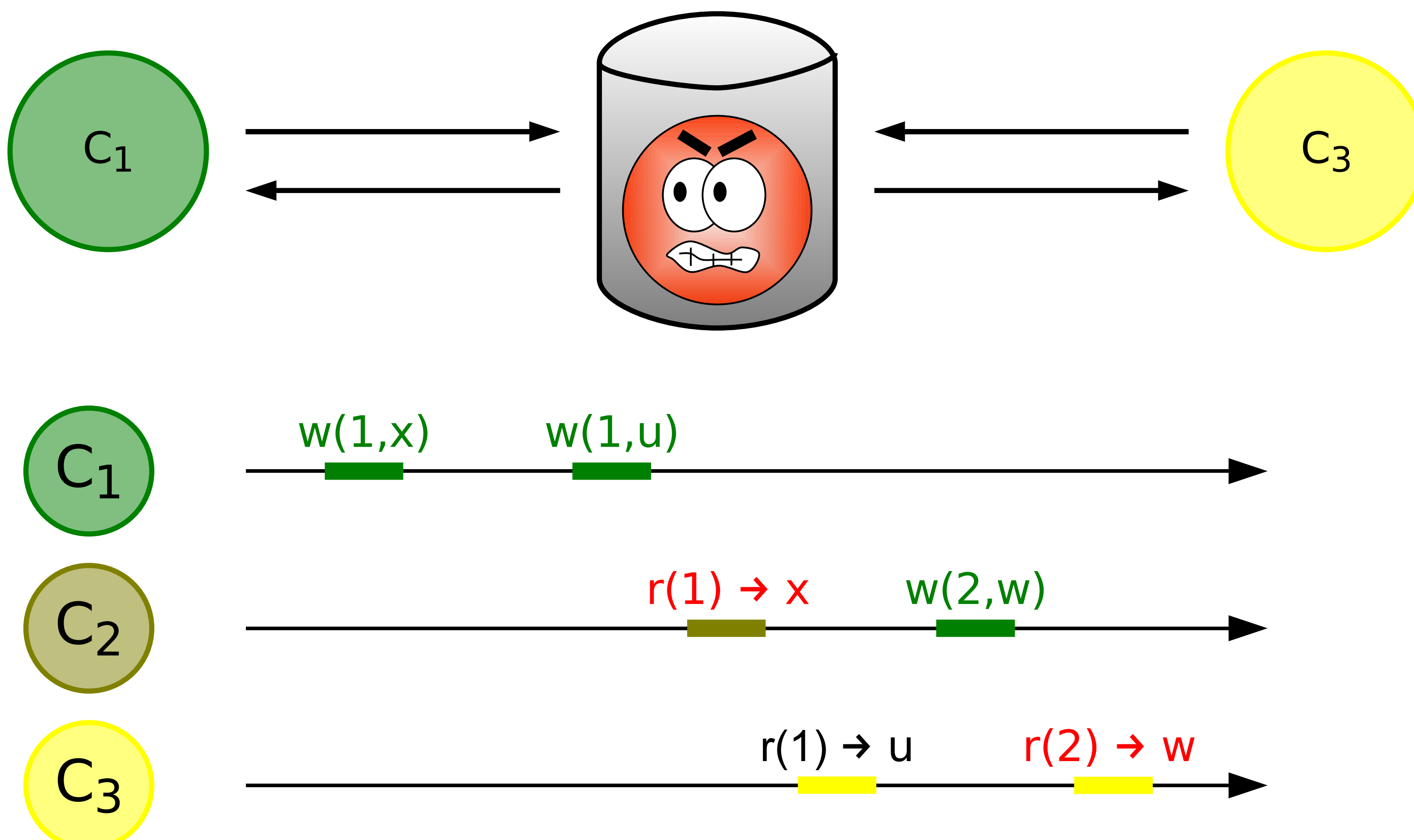
- Loose synchronization and concurrency pose a new problem
- Suppose clients sign data with digital signatures:

  Server cannot forge any values ...

  → But answer with outdated value (“replay attack”)

  → Or send different values to different clients

Illustration of the Problem



Solution: Fork linearizability [MS02, CSS07]

- Server may present different views to clients
  - → “Fork” their views of history
  - → Clients cannot prevent this

- Fork linearizability

  If server forks the views of two clients *once*, **then**

  → their views are forked *ever after*

  → they *never again* see any updates of each other

- Forks are easier to detect than subtle data modifications
  - → Needs a separate channel for detection

- Cryptographic protocols can ensure fork linearizability [MS02, CSS07]
  - → Implemented in SUNDR file system [LKMS04]

Cryptography for Storage in Action

- Tape drive with encryption (IBM TS1120)
- TCG storage specification and drive-encryption (Seagate)
- A cryptographic SAN filesystem [PC07]

Tape Drives with built-in Encryption (IBM TS1120)

- Hardware-based encryption in drive

  Authenticated encryption in Galois/counter mode with AES-256

- Hybrid encryption scheme

  → Cartridge analogous to a PGP message

  Data Key (DK) encrypts raw data on tape (AES key)

  → DK chosen randomly, like a session key

  Key-Encryption Key (KEK) encrypts DK (public key of receiver)

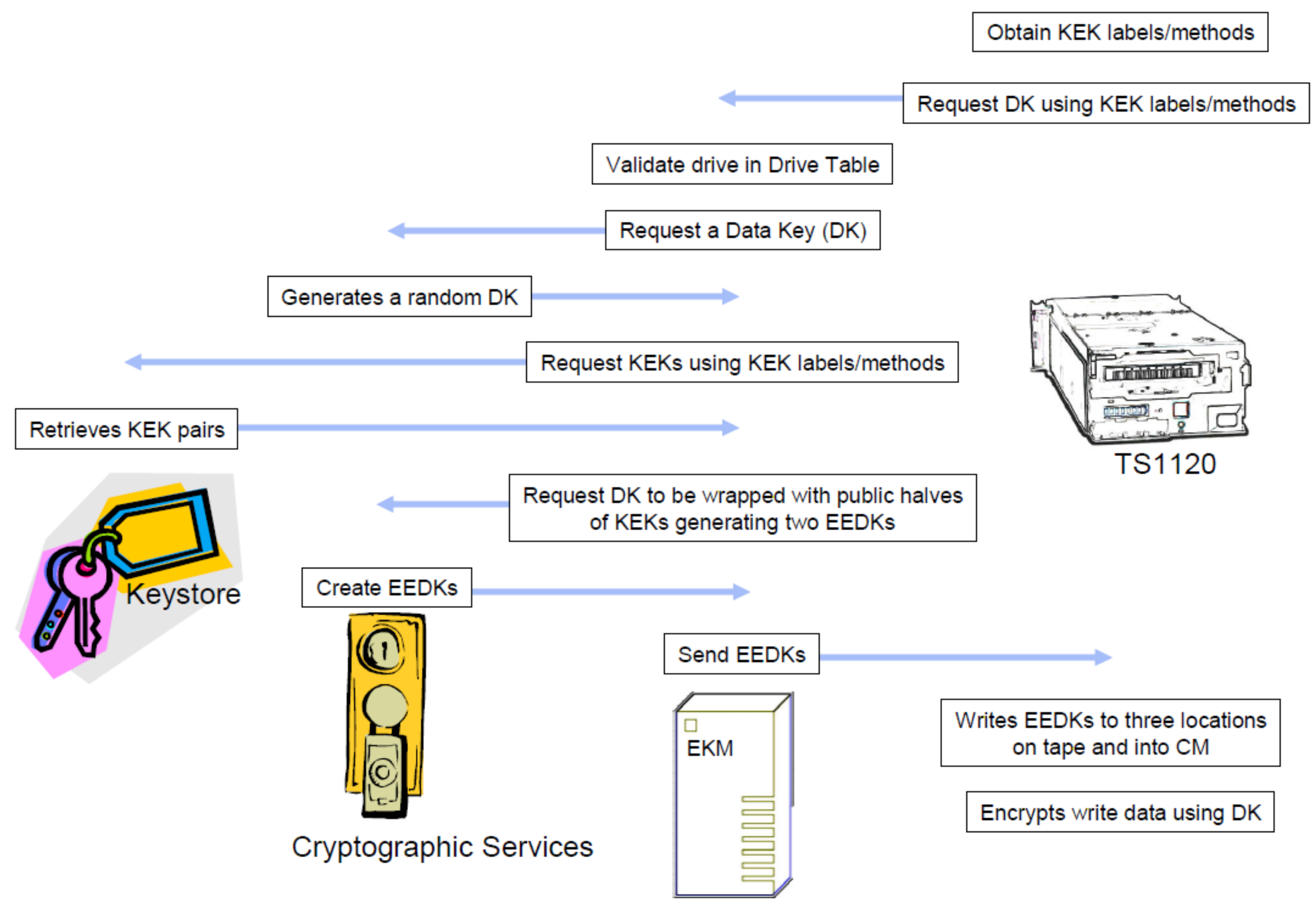
  → Result is Encrypted DK (EEDK)

  → EEDK is stored on tape and in cartridge memory

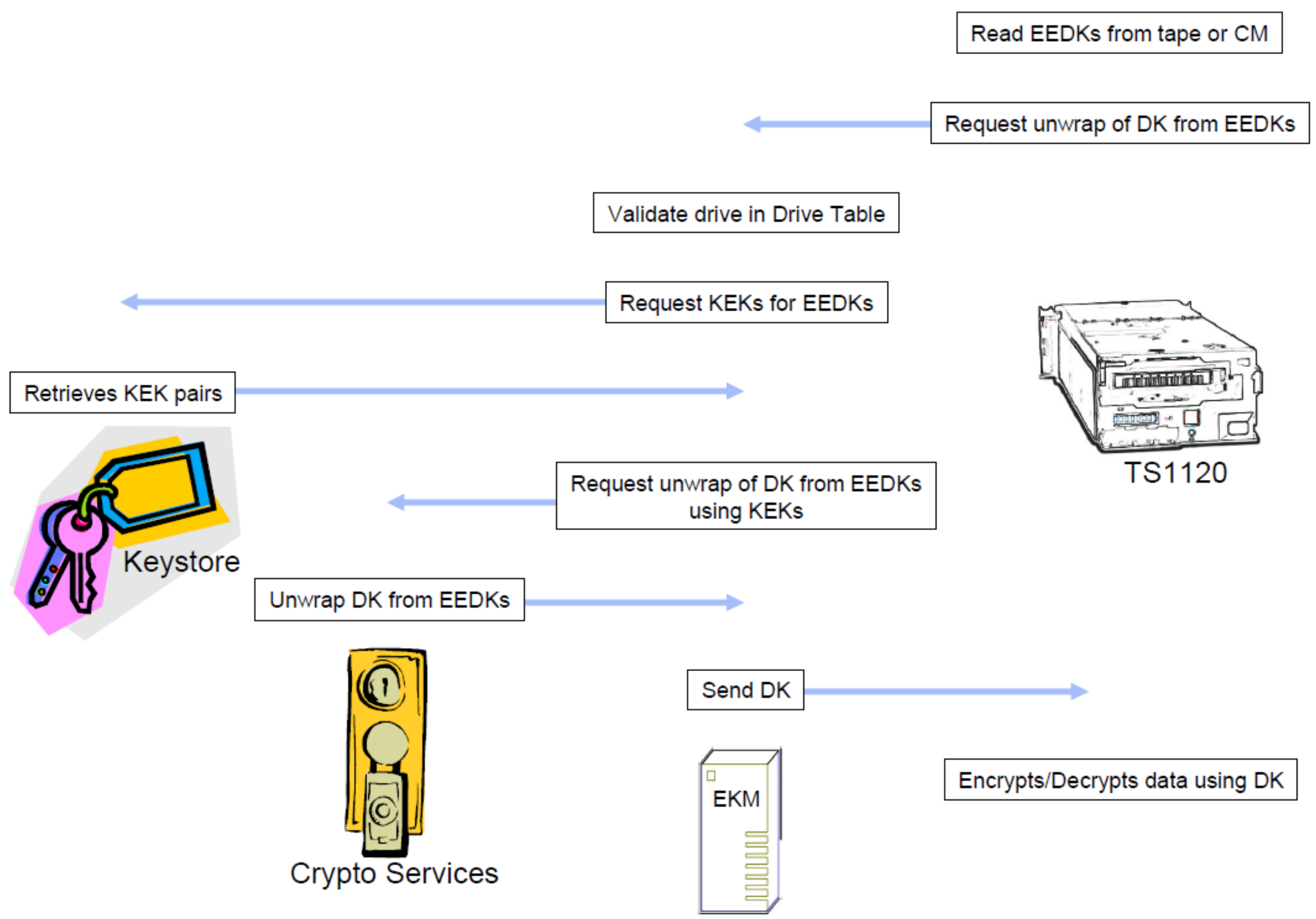
  Up to 2 EEDKs per cartridge

- Public-key operations for key serving done by Encryption-Key Manager (EKM) on host

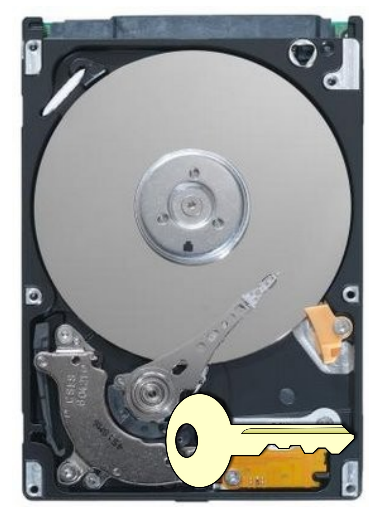
Data Encryption Process for Writing Tape



Data Decryption Process for Reading Tape

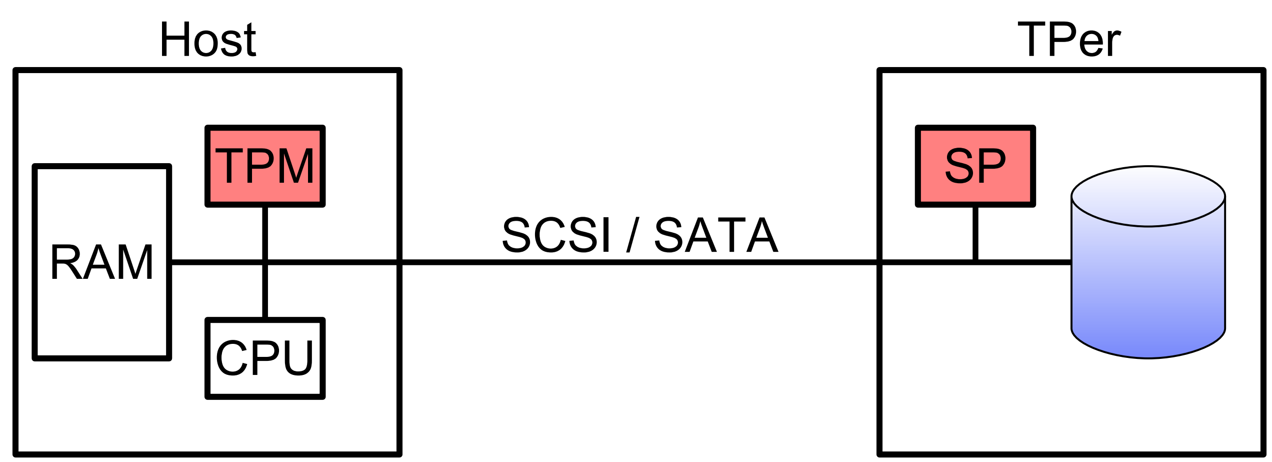


Disk Drives with built-in Encryption (Seagate)



- Encryption in hardware on the drive
  - → Transparent to application
  - → No performance issues (scales with storage space)
- Key stored in drive logic inside disk enclosure
  - → Never leaves drive
  - → May exploit smartcard-like secure hardware
- User or host authenticates to drive before OS boot
  - → Security is shifted to authentication
  - → Authentication methods
    - Password/PIN entered via BIOS
    - Cryptographic methods (Public-key signature or MAC)
- Seagate's FDE drive
  - → AES for bulk encryption (details not public, but NIST has validated its ECB mode ...)

TCG Storage Architecture



- Trusted Peripheral (TPer) contains a Security Provider (SP)
- TPer communicates with host, its TPM, or other devices via:
  - → SCSI (T10) Security Protocol IN/OUT commands
  - → SATA (T13) Trusted Send/Receive commands
- SP acts as a root of trust, in storage device
  - ≠ most other methods presented here, where storage is not trusted

TCG Storage Architecture Details

- Security Provider (SP)
  - → SP: logical group of security features
  - → Tables: register-like primitive storage and control functions
  - → Methods: simple get/set commands
  - → Access control over methods and tables
- Cryptographic functions
  - → Encryption, decryption, hashing, MAC, signing, verifying ...
  - → AES, RSA, ECC, SHA-2, HMAC ...
- SP has a life-cycle that needs support
  - → Manufacturing ↔ issued / active ↔ disabled / active ↔ frozen
  - → Life-cycle of TPer: Produce, own, enroll, connect, use ...
- Currently a draft standard ...

A Cryptographic SAN Filesystem [PC07]

SANs and SAN Filesystems

- SAN today:

  Clients access block storage devices directly

  → Fibre Channel (SCSI)

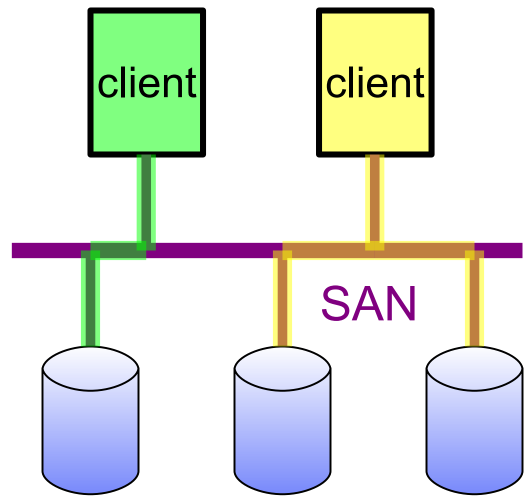
  Static configuration

  → OS sees a local block storage device

  Static access control

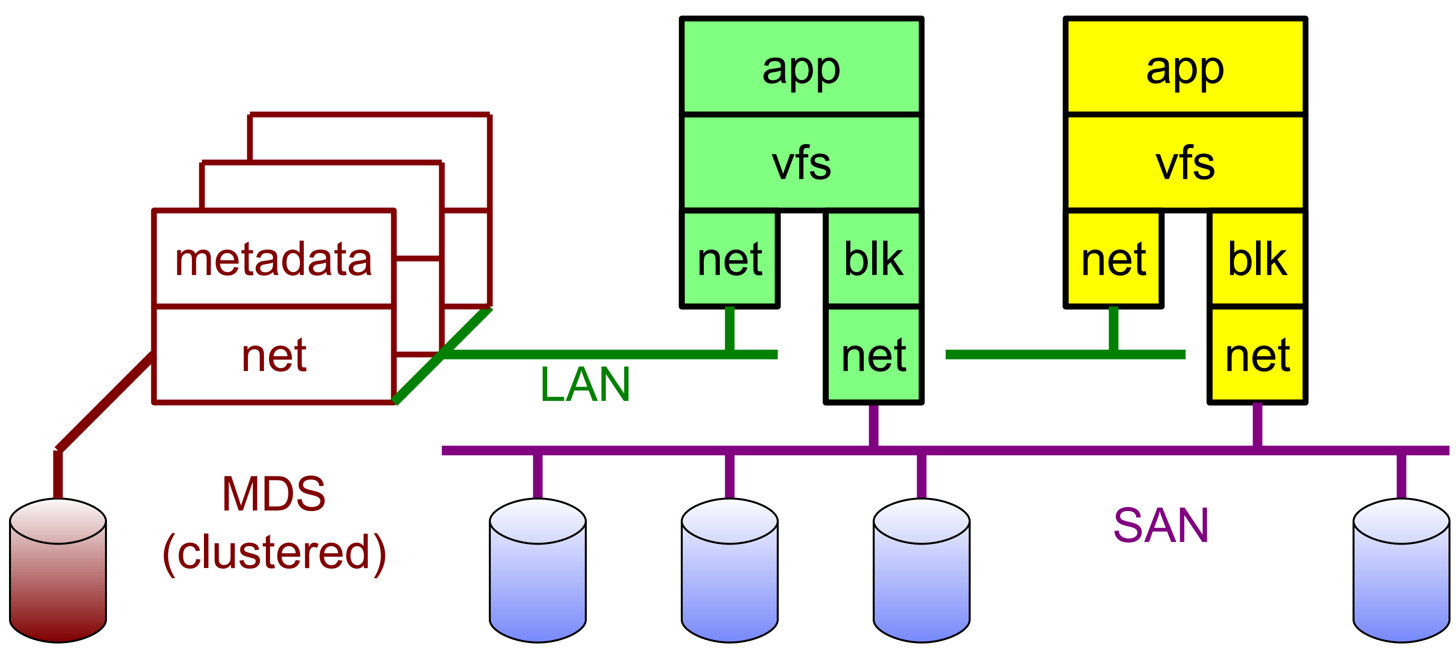
  → Zoning & fencing in FC switch

  Inside server room only



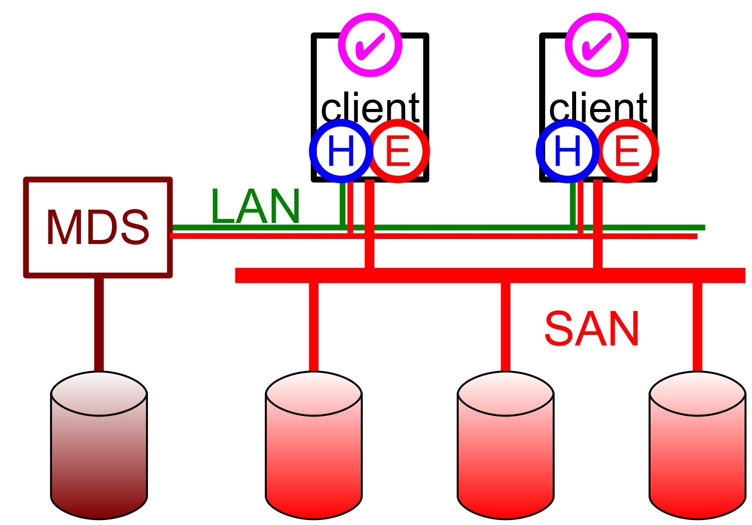
SAN Filesystems (e.g. IBM's StorageTank)

- Virtualized block storage space
- Block access managed by metadata server (MDS)
- Single filesystem name space
- Heterogeneous clients



Design of a Cryptographic SAN Filesystem

- Integrity verification & encryption in client
  - → Scalable
  - → End-to-end security
- MDS is trusted, provides encryption keys & reference data
  - → Integrate key management with metadata
  - → No modification of storage interface
- Needs
  - secure LAN connection (IPsec)
  - trusted client kernels



✓ Access control
H Integrity protection
E Encryption

Summary

- Any security mechanism can be applied on any layer
- Challenge is to select the “right” combination

	Ⓔ	Ⓐ	✓
file	key mgmt. & lazy revocation	hash trees & fork-linearizability	
object			OBS security protocol
block	block-cipher modes & IEEE P1619	hybrid block-integrity protection	

Thank you!

- More information?

  http://www.zurich.ibm.com/~cca

  <cca@zurich.ibm.com>

References (1)

- [BCO05] M. Backes, C. Cachin, and A. Oprea. Lazy revocation in cryptographic file systems. In Proc. 3rd Intl. IEEE Security in Storage Workshop, December 2005.
- [BCO06] M. Backes, C. Cachin, and A. Oprea. Secure key-updating for lazy revocation. In Proc. 11th European Symposium On Research In Computer Security (ESORICS), vol. 4189 of Lecture Notes in Computer Science, Springer, 2006.
- [BM97] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In Advances in Cryptology: EUROCRYPT '97, vol. 1233 of Lecture Notes in Computer Science, Springer, 1997.
- [CDDGS03] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental multiset hash functions and their application to memory integrity checking. In Advances in Cryptology: ASIACRYPT 2003, vol. 2894 of Lecture Notes in Computer Science, Springer, 2003.
- [CL02] J. Camenisch and A. Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Advances in Cryptology: CRYPTO 2002, vol. 2442 of Lecture Notes in Computer Science, Springer, 2002.
- [CSS07] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In Proc. 26th ACM Symp. Principles of Distributed Computing (PODC), 2007.

References (2)

- [FKK06] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In Proc. Network and Distributed Systems Security Symposium (NDSS), 2006.
- [FKM02] K. Fu, M. Kaminsky, and D. Mazières. Using SFS for a secure network file system. ;login: --- The Magazine of the USENIX Association, 27(6), December 2002.
- [FNN+05] M. Factor, D. Nagle, D. Naor, E. Riedel, and J. Satran. The OSD security protocol. In Proc. 3rd Intl. IEEE Security in Storage Workshop (SISW 2005), pages 29-39, 2005.
- [Fu99] K. Fu. Group sharing and random access in cryptographic storage file systems. Master Thesis, MIT LCS, 1999.
- [GSMB03] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In Proc. Network and Distributed Systems Security Symposium (NDSS), 2003.
- [HR04] S. Halevi and P. Rogaway. A parallelizable enciphering mode. In Topics in Cryptology: CT-RSA 2004, vol. 2964 of Lecture Notes in Computer Science, Springer, 2004.
- [KRS+03] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In Proc. 2nd USENIX Conference on File and Storage Technologies (FAST 2003), 2003.

References (3)

- [L06] B. Lalin. Transparent Data Integrity for a File System. Master thesis, KTH Stockholm and IBM Zurich Research Laboratory, 2006.
- [LKMS04] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In Proc. Symp. Operating Systems Design and Implementation (OSDI), 2004.
- [LRW02] M. Liskov, R. R. Rivest, and D. Wagner. Tweakable block ciphers. In Advances in Cryptology: CRYPTO 2002, vol. 2442 of Lecture Notes in Computer Science, Springer, 2002.
- [MS02] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In Proc. 21st ACM Symp. Principles of Distributed Computing (PODC), 2002.
- [MVS00] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In Proc. Symp. Operating Systems Design and Implementation (OSDI), 2000.
- [ORY05] A. Oprea, M. Reiter, and K. Yang. Space-efficient block storage integrity. In Proc. Network and Distributed Systems Security Symposium (NDSS 2005), 2005.
- [PC07] R. Pletka and C. Cachin. Cryptographic security for a high-performance distributed file system. In Proc. 24th Mass Storage Systems and Technologies (MSST), Sept. 2007.
- [R04] P. Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In Advances in Cryptology: ASIACRYPT 2004, vol. 3329 of Lecture Notes in Computer Science, Springer, 2004.

Further Reading

- [KK05] V. Kher and Y. Kim. Securing distributed storage: Challenges, techniques, and systems. In Proc. Workshop on Storage Security and Survivability (StorageSS), 2005.
- [RKS02] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In Proc. USENIX Conference on File and Storage Technologies (FAST), 2002.
- [WDZ03] C. Wright, J. Dave, and E. Zadok. Cryptographic file systems performance: What you don't know can hurt you. In Proc. 2nd International IEEE Security in Storage Workshop (SISW), 2003.