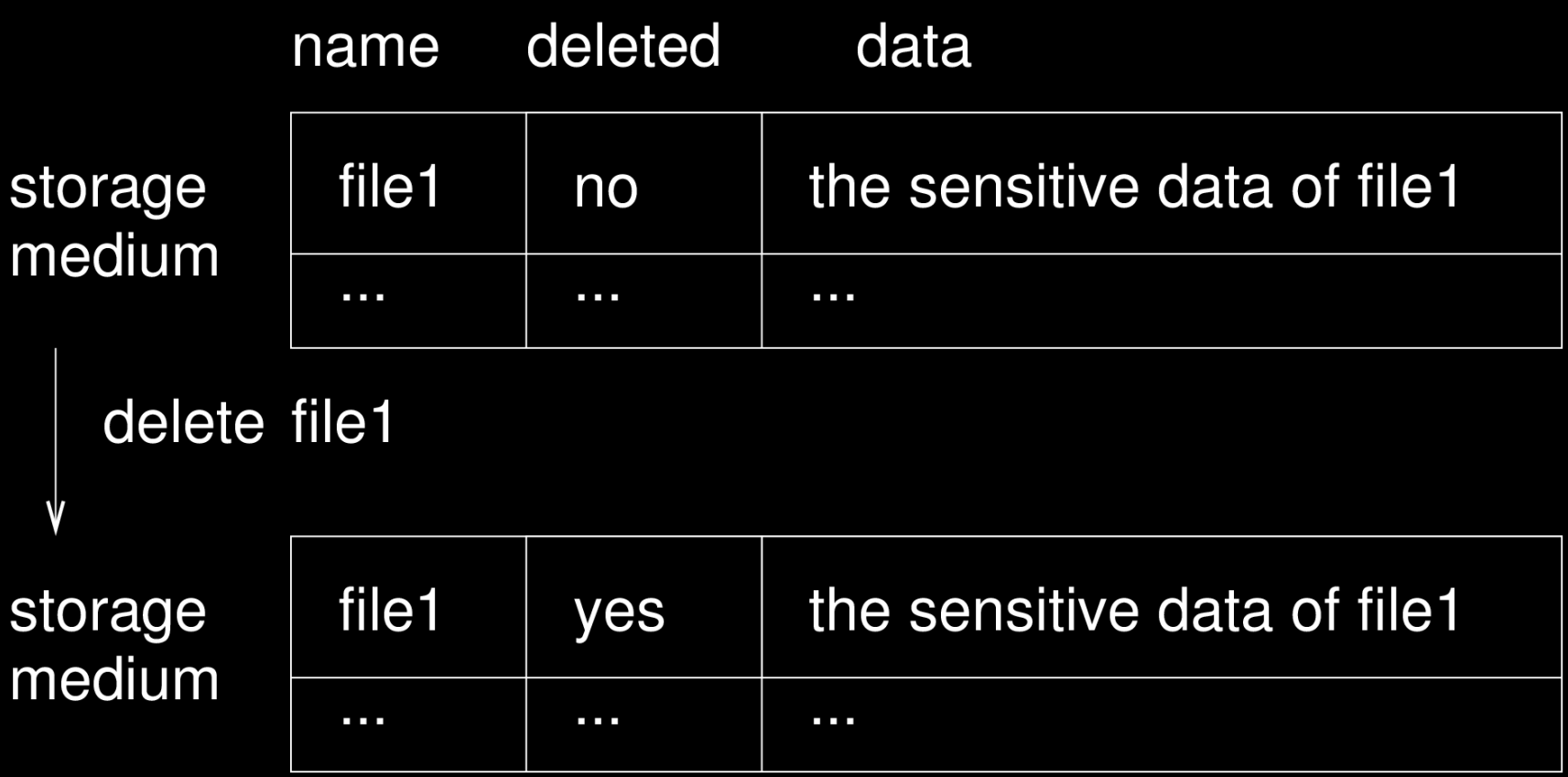
Data Node Encrypted File System: Secure Deletion for Flash Memory

Joel Reardon Srdjan Čapkun David Basin
ETH Zurich, Switzerland

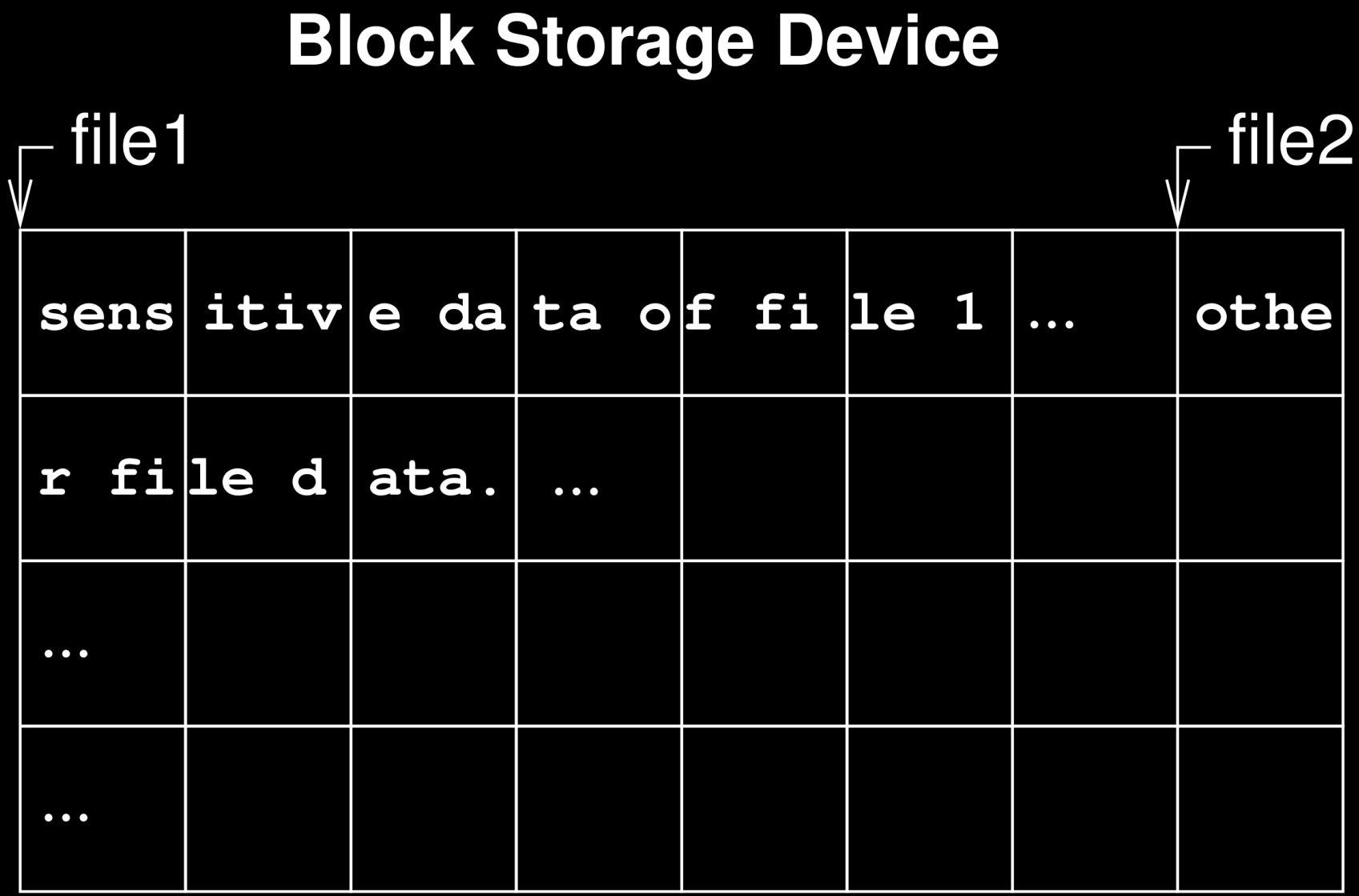
9.8.12

Secure deletion: security task of deleting information such that it becomes irrecoverable (to a coercive attacker)

Secure Deletion Problem

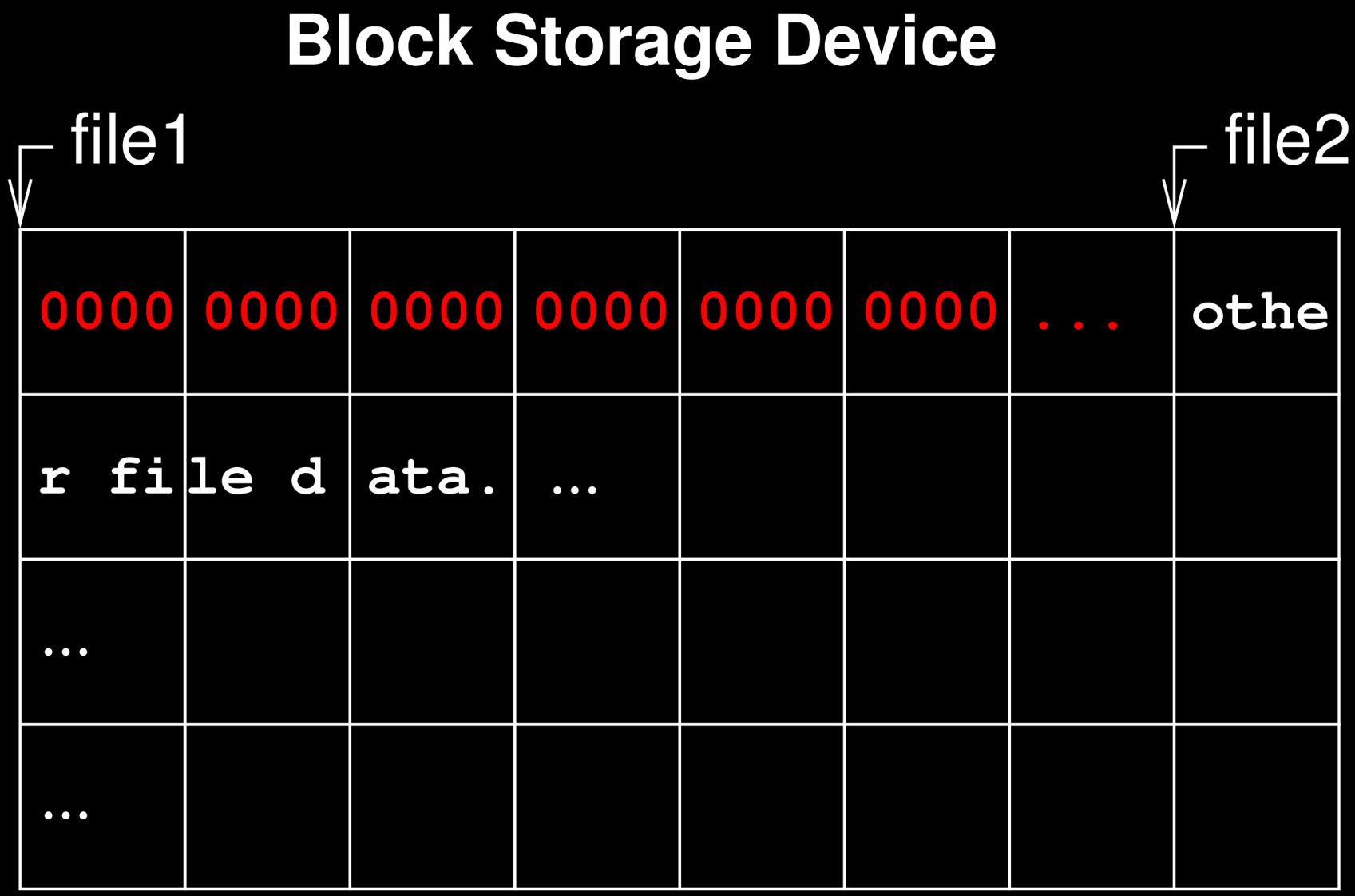


Secure Deletion Easily Solved for Block Devices



- block device layout

Secure Deletion Easily Solved for Block Devices



- block device layout after overwriting file data

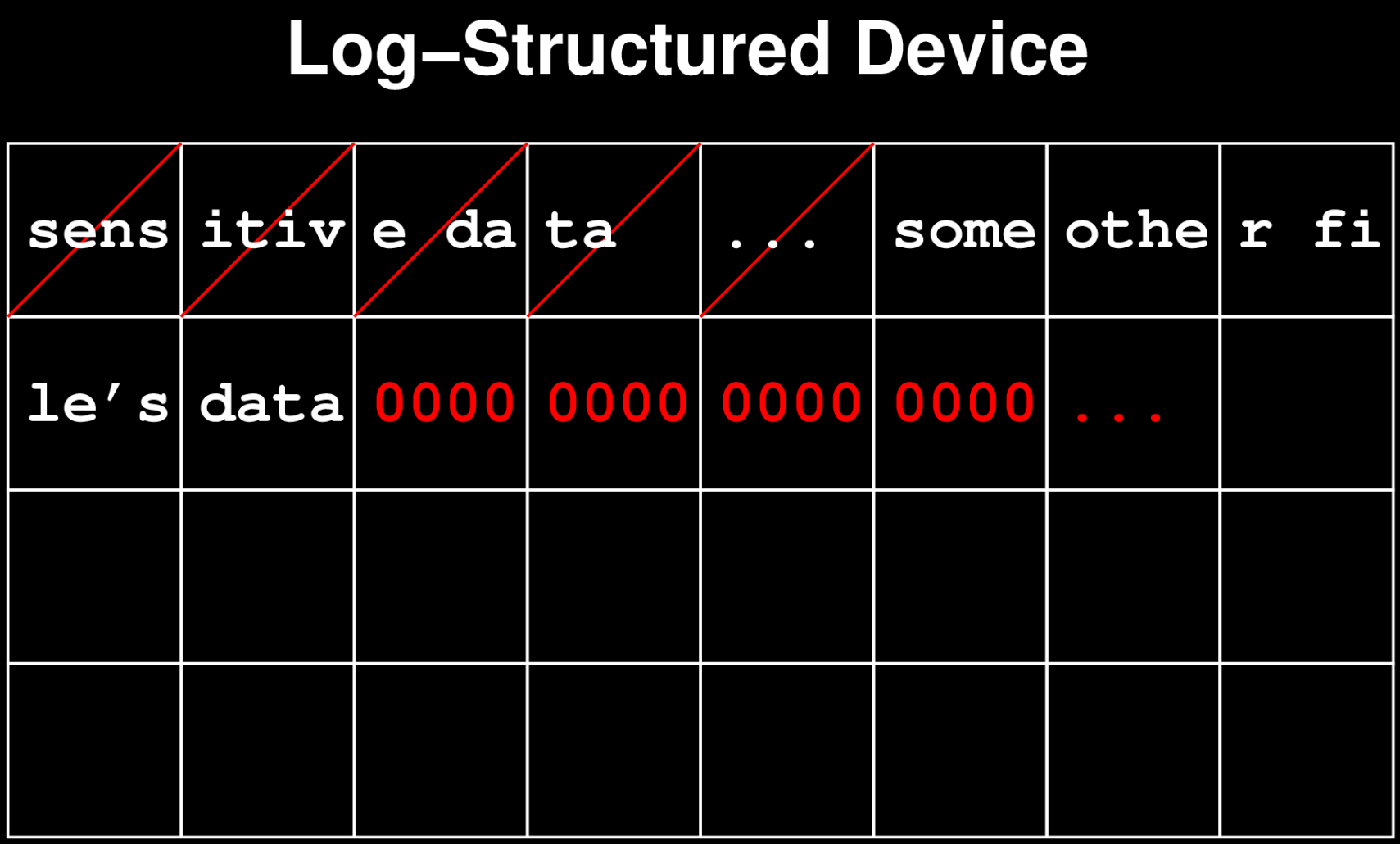
Secure Deletion not Trivial for Log-Structured FSes

Log–Structured Device

sens	itiv	e da	ta	...	some	othe	r fi
le's	data						

- log-structured device layout

Overwriting in Log-Structured File Systems



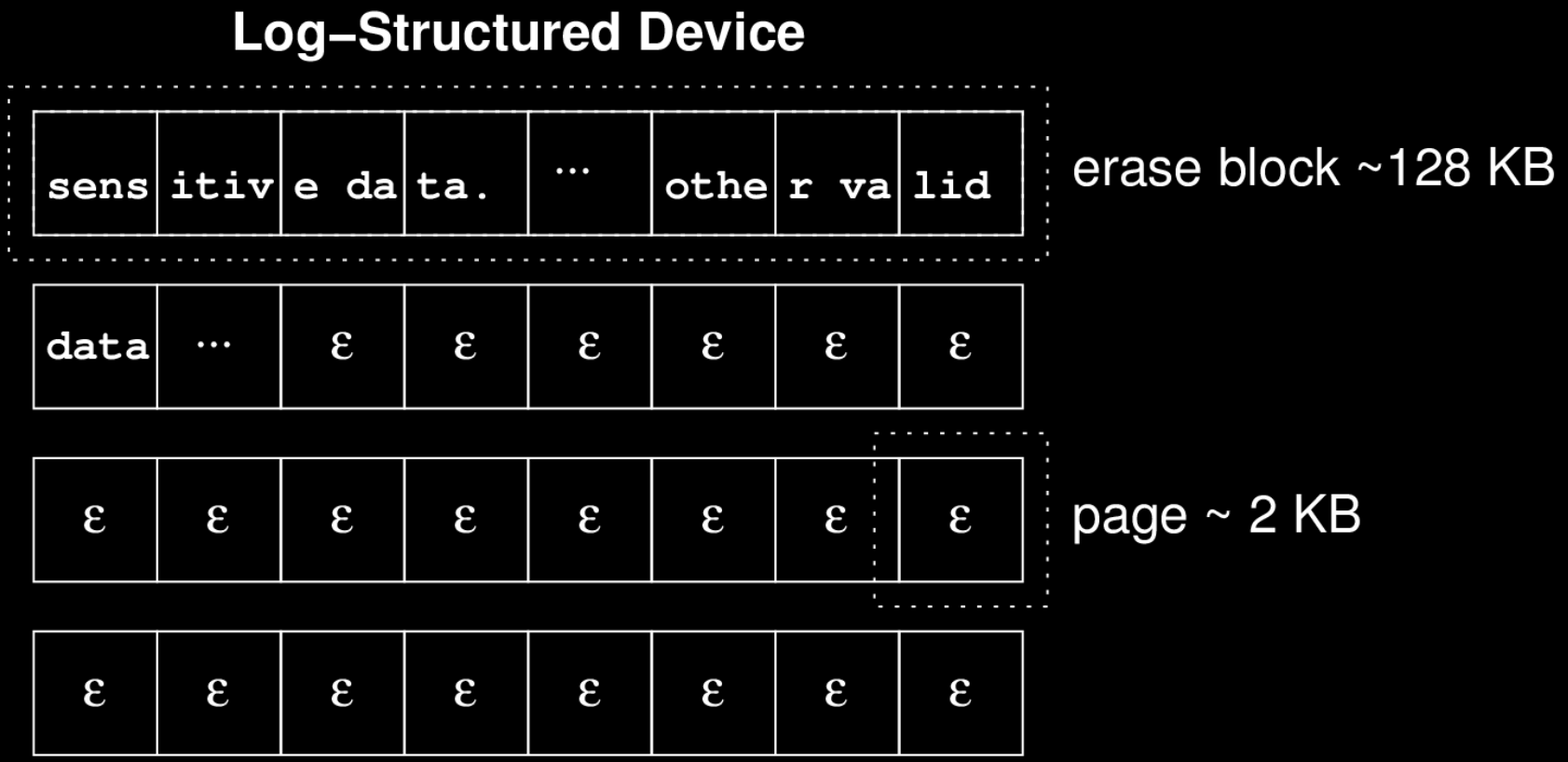
- log-structured device layout after overwriting file data

Why are log-structured file systems relevant?
Paradigm is ubiquitously used for flash memory
(ubiquitously used for portable devices)

YAFFS (used on Android phones)

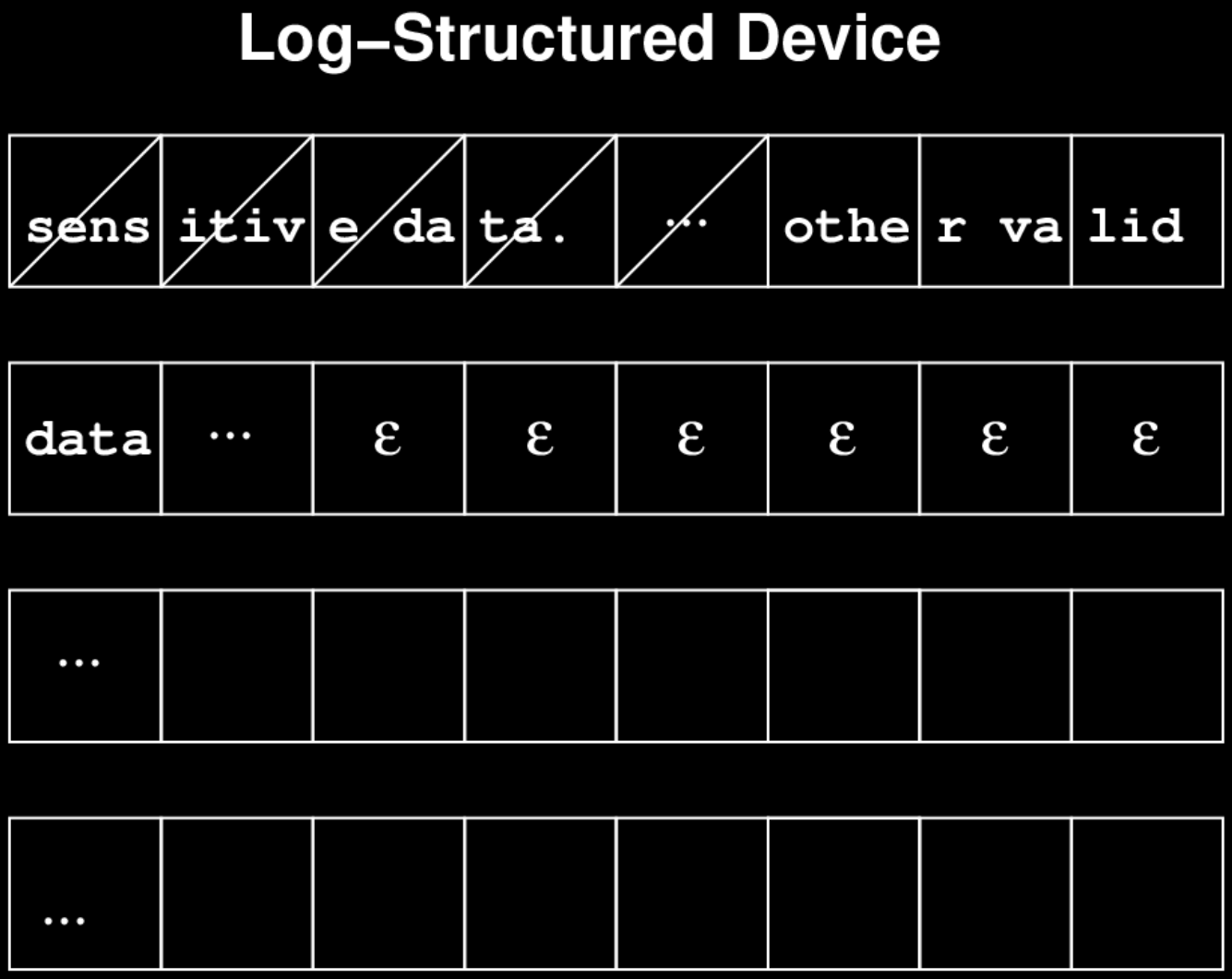
- YAFFS is a log-structured file system
- deleted data remains with average use upwards of 48 hours and months with infrequent use

Log-Structured File Systems and Flash Memory



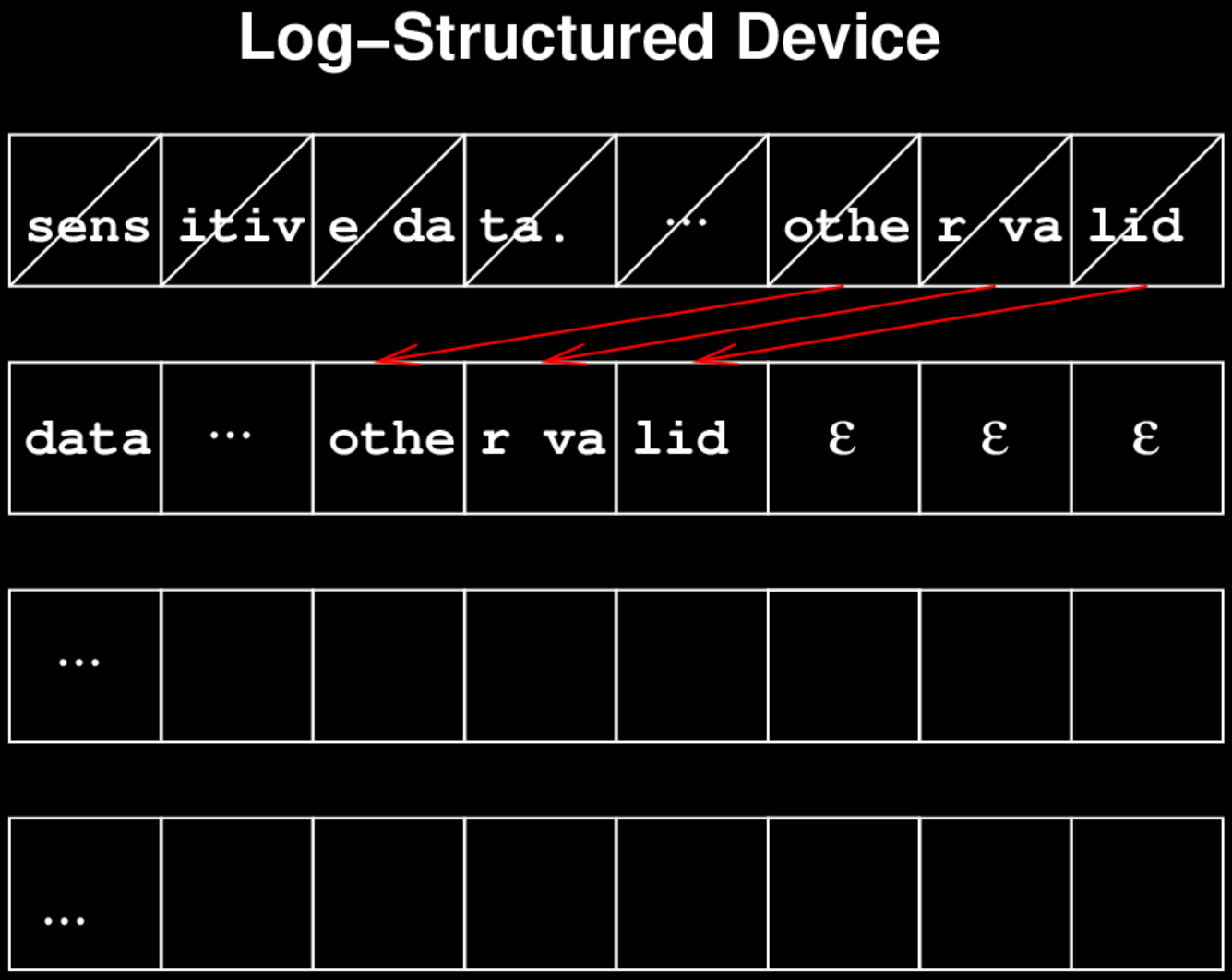
- flash memory holds electrical charge without power
  - erasing is a brute operation that fills the charge of many cells
  - writing is a surgical operation that drains particular cells
- erasures are costly: power, wear, time
  - erasure is natural efficiency metric
  - erasures should also be evenly levelled

Naive Approach



- erase blocks contain an eclectic mixture of colocated data

Naive Approach



- deleted all data in this way is very costly

Design Goals

- possible solutions
  - only securely delete sensitive files
  - encrypt each file with a key
  - drain charge from remaining cells (scrubbing)
- what we want to achieve
  - work within specification of flash memory
  - be transparent to the application and users
  - small cost in space, memory, and computation
  - efficient fine-grained secure deletion

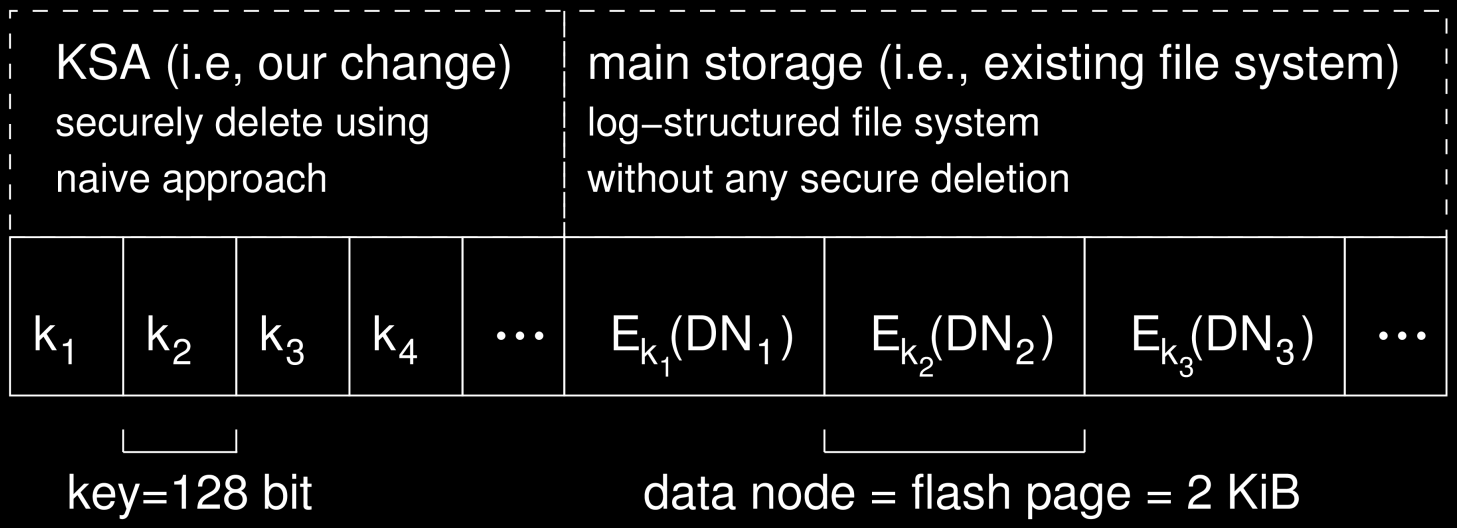
Our contributions

- Data Node Encrypted File System (DNEFS)
  - general file system change that affords efficient secure deletion
- UBIFSec
  - full implementation of DNEFS for the Linux Flash File system UBIFS

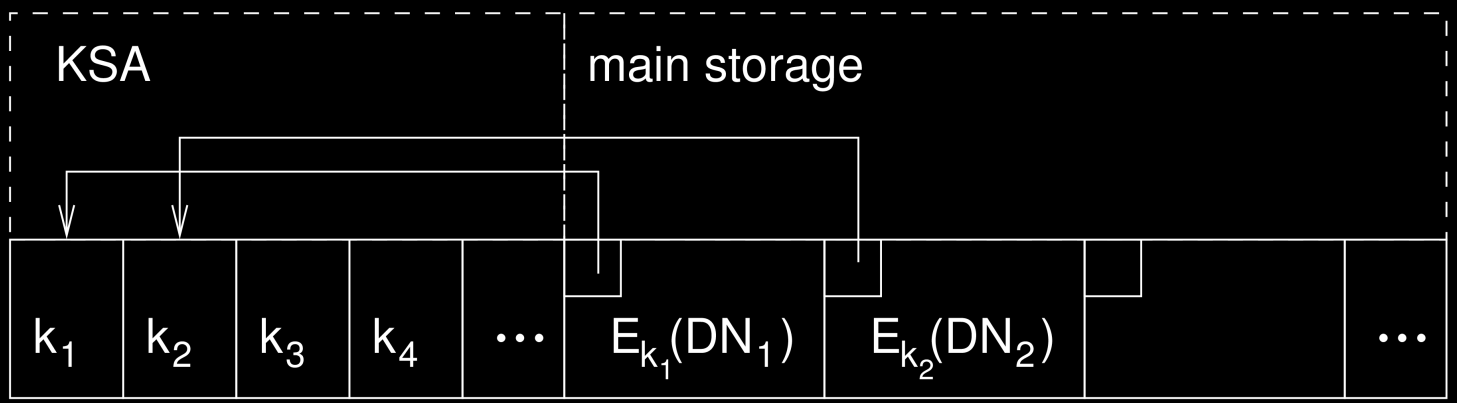
Data Node Encrypted File System (DNEFS)

- intuition:
  - we need (at least) to erase an erase block to delete some data
    - without batching, this reduces to the inefficient naive solution
  - goal now is to maximize ratio of bytes deleted to erase blocks erased
- solution:
  - encrypt each data node with a unique key
  - colocate the keys in a (dense) key storage area (KSA)
  - periodically purge KSA to remove deleted keys

DNEFS Outline

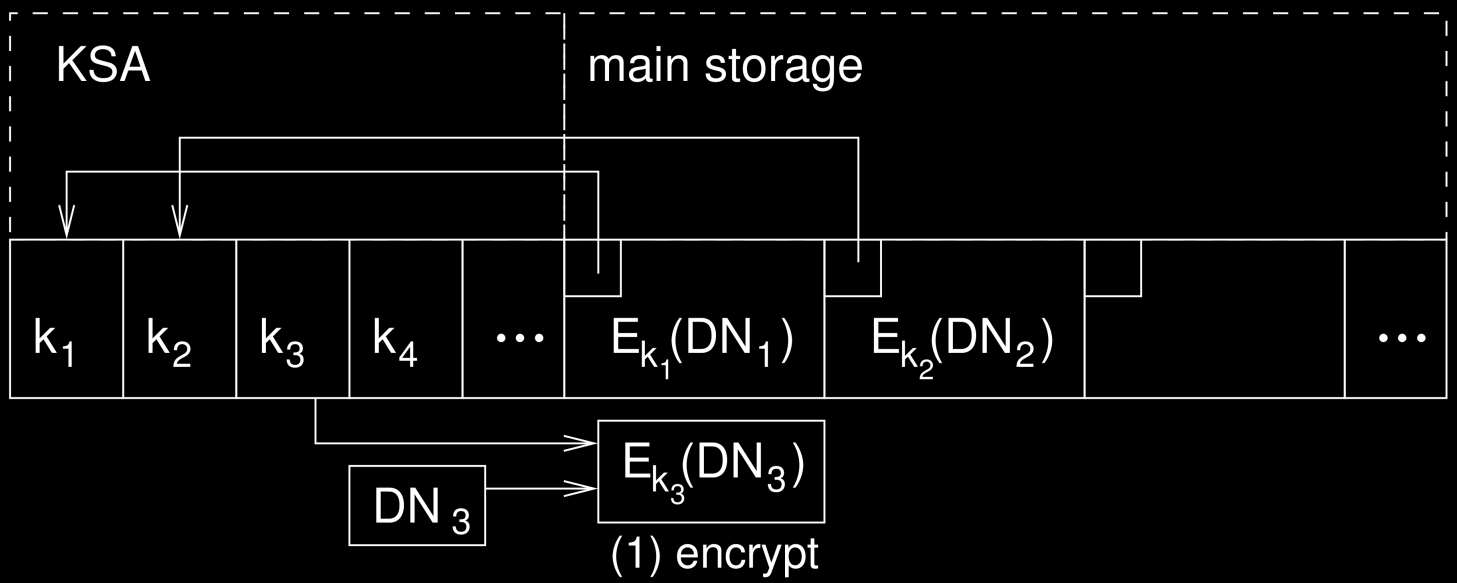


DNEFS Write

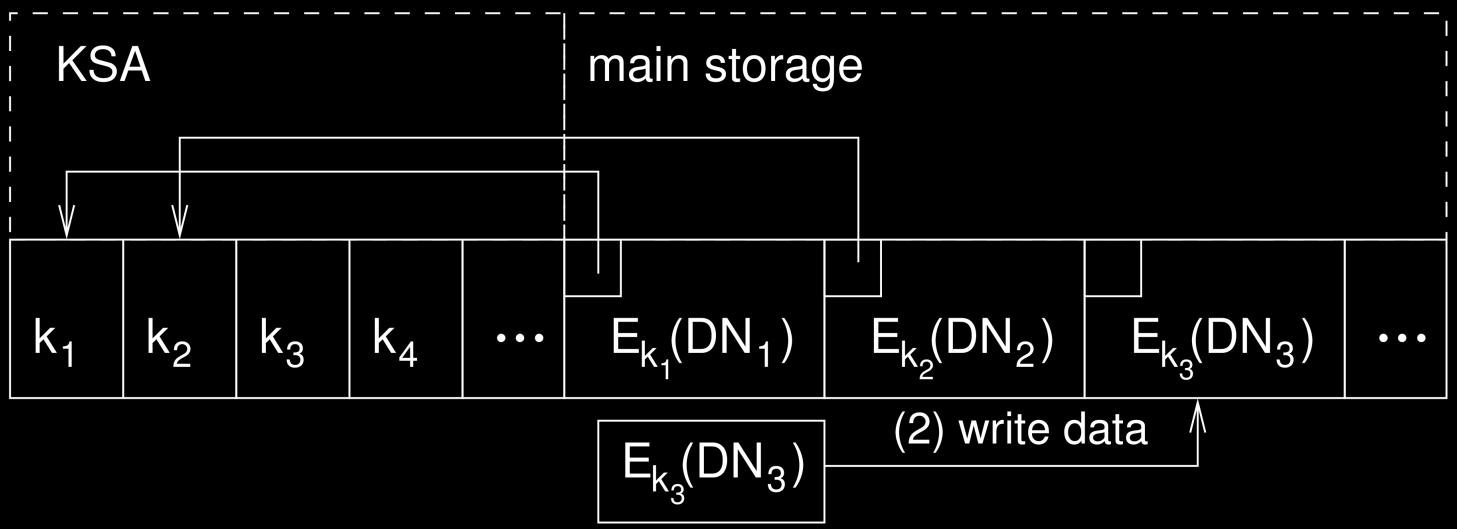


WRITE DN_3

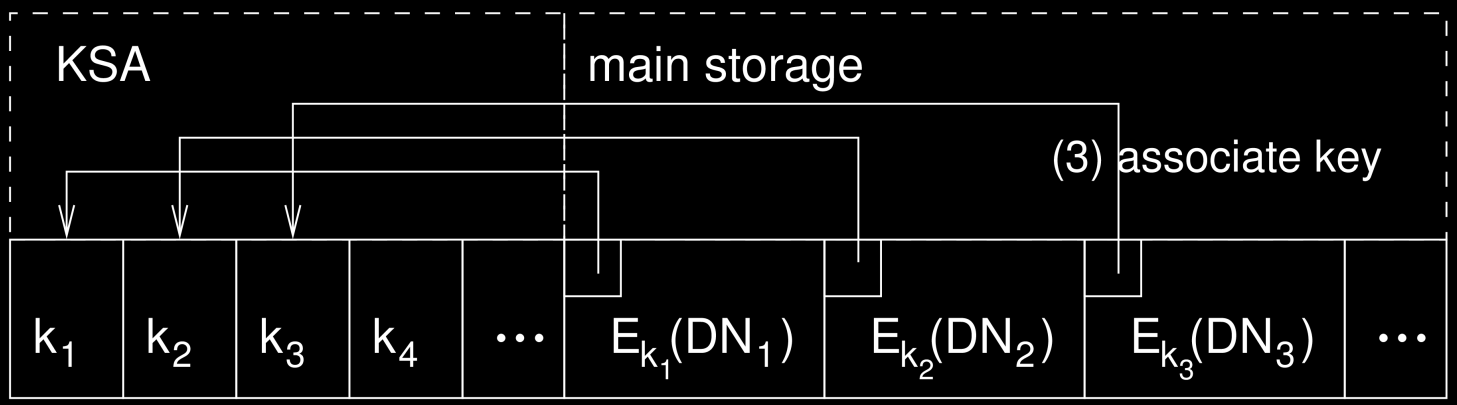
DNEFS Write


KSA
main storage
k_1
k_2
k_3
k_4
$\cdots$
$E_{k_1}(DN_1)$
$E_{k_2}(DN_2)$
$\cdots$
$E_{k_3}(DN_3)$
DN_3
(1) encrypt

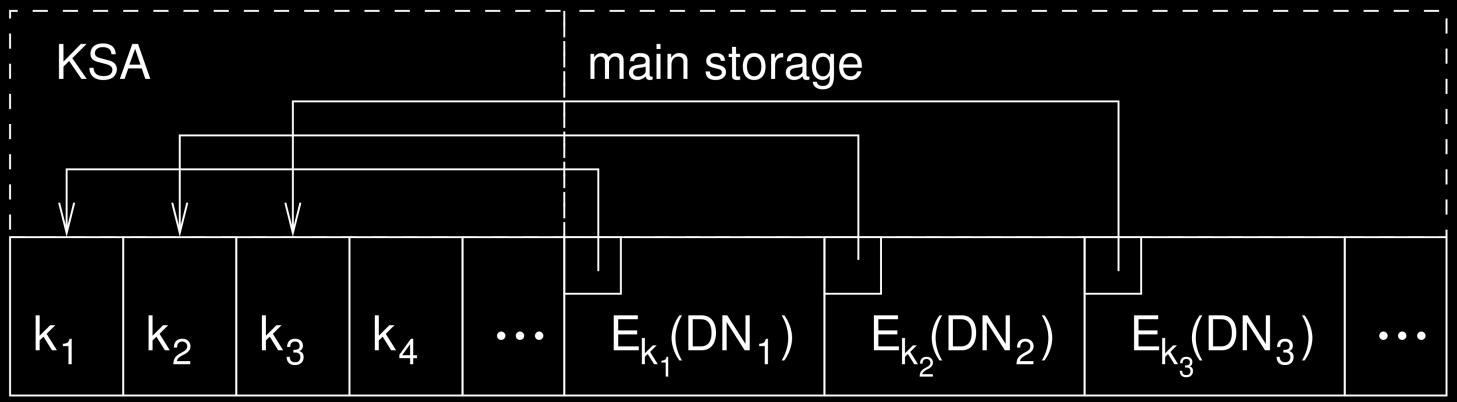
DNEFS Write


KSA
main storage
k_1
k_2
k_3
k_4
$\cdots$
$E_{k_1}(DN_1)$
$E_{k_2}(DN_2)$
$E_{k_3}(DN_3)$
$\cdots$
(2) write data
$E_{k_3}(DN_3)$

DNEFS Write

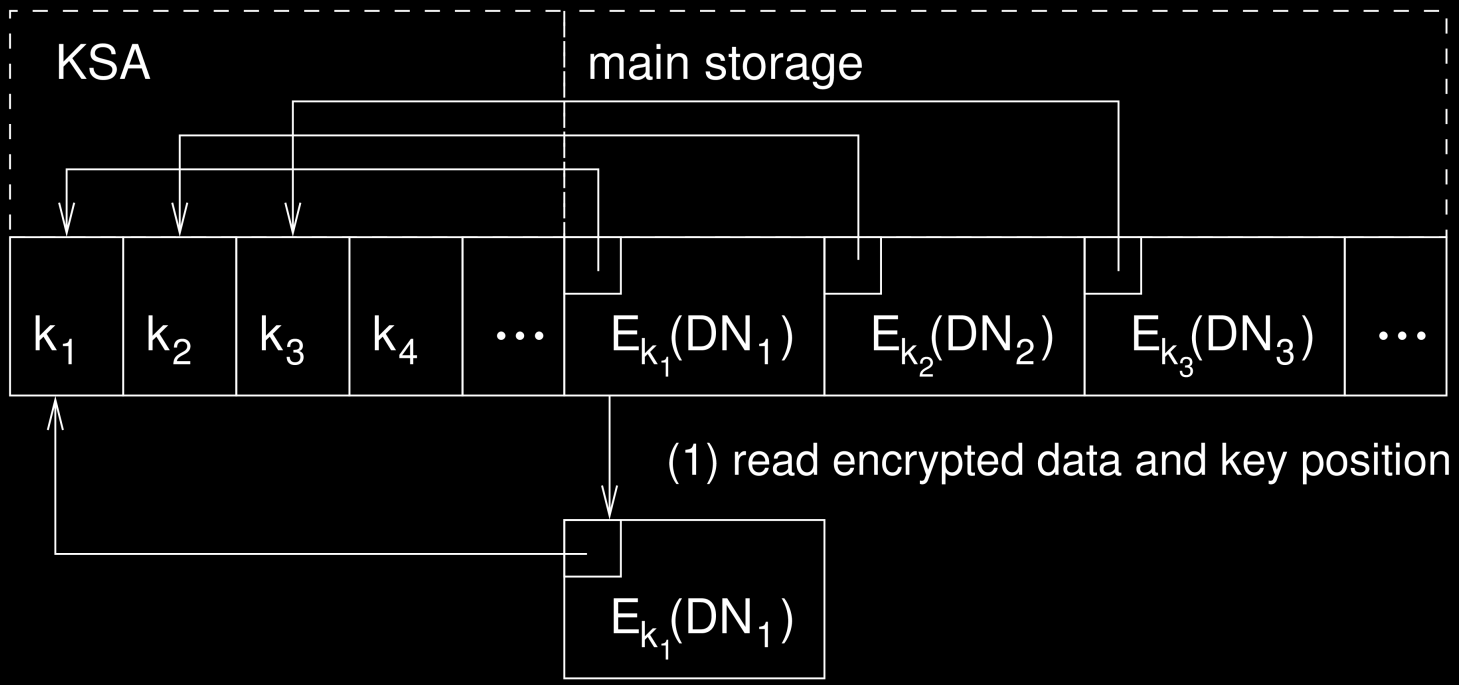

KSA
main storage
(3) associate key
k_1
k_2
k_3
k_4
$\cdots$
$E_{k_1}(DN_1)$
$E_{k_2}(DN_2)$
$E_{k_3}(DN_3)$
$\cdots$

DNEFS Read

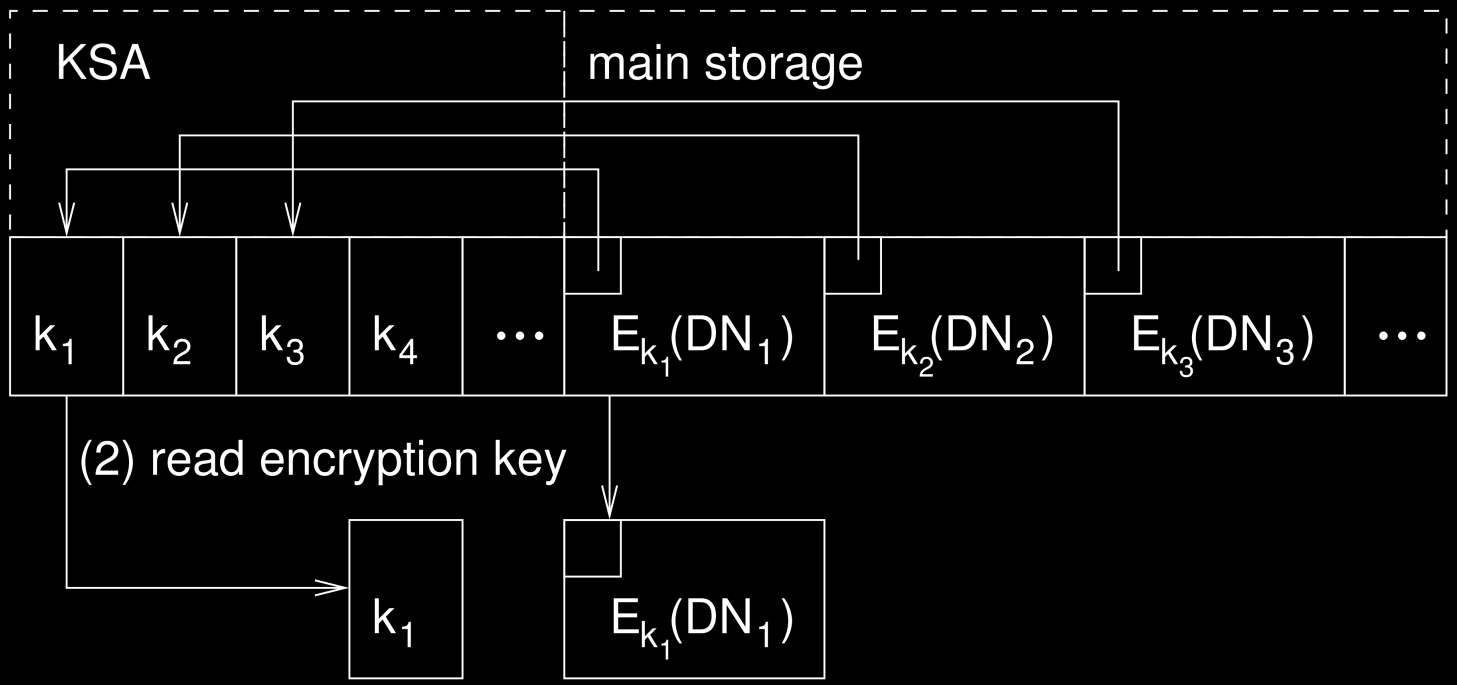


READ DN_1

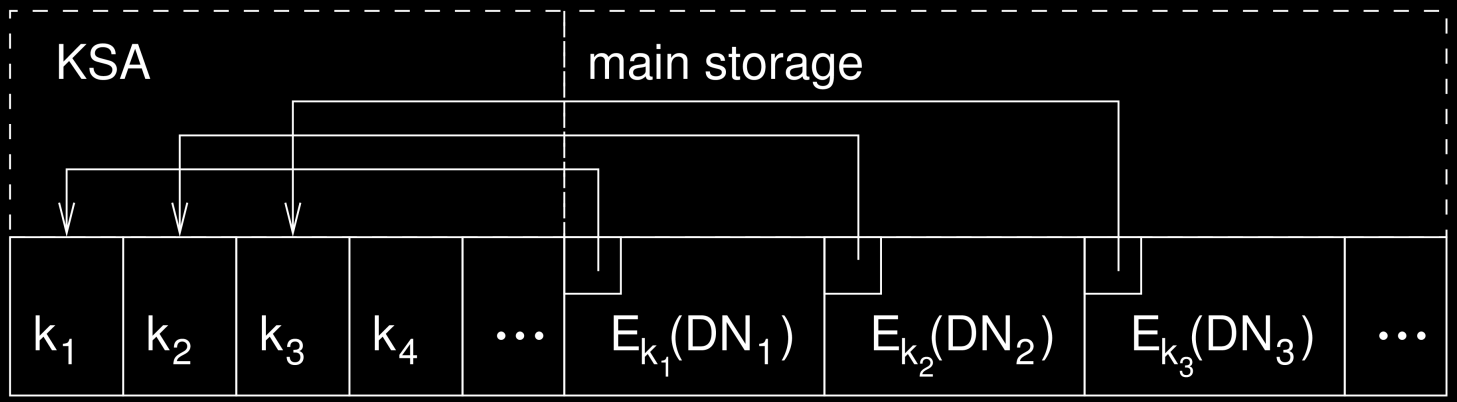
DNEFS Read

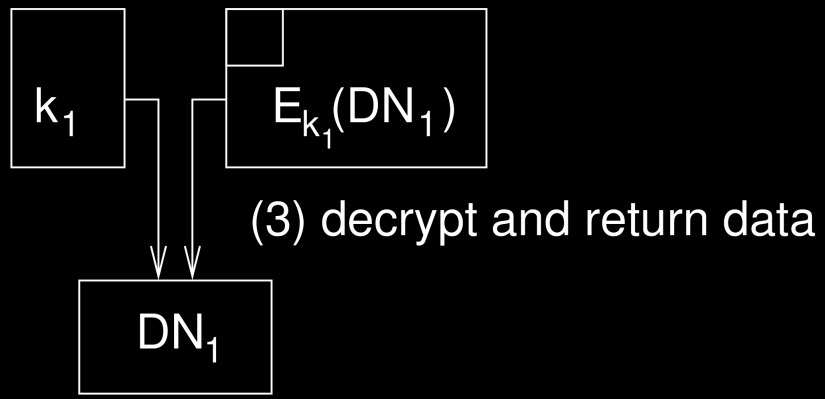

KSA
main storage
k_1
k_2
k_3
k_4
$\cdots$
$E_{k_1}(DN_1)$
$E_{k_2}(DN_2)$
$E_{k_3}(DN_3)$
$\cdots$
(1) read encrypted data and key position
$E_{k_1}(DN_1)$

DNEFS Read

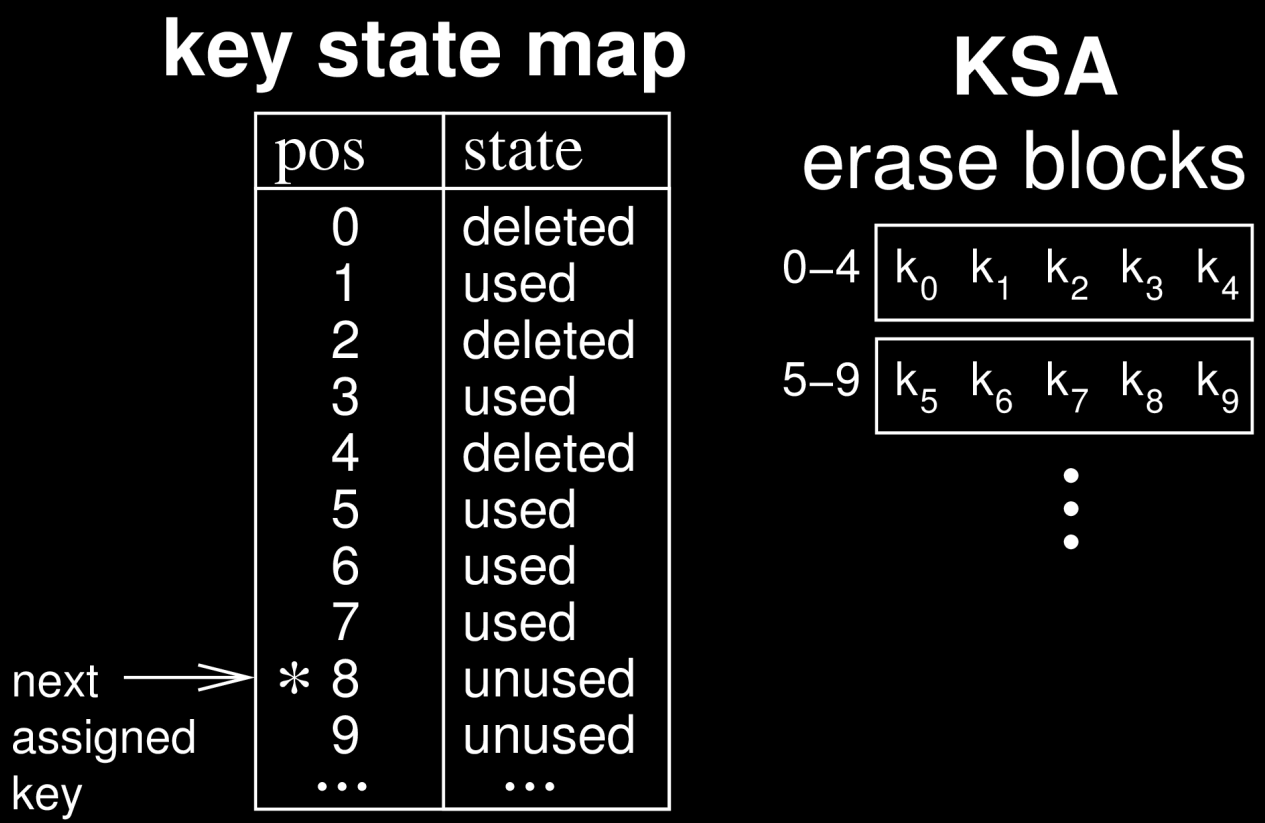

KSA
main storage
k_1
k_2
k_3
k_4
$\cdots$
$E_{k_1}(DN_1)$
$E_{k_2}(DN_2)$
$E_{k_3}(DN_3)$
$\cdots$
(2) read encryption key
k_1
$E_{k_1}(DN_1)$

DNEFS Read

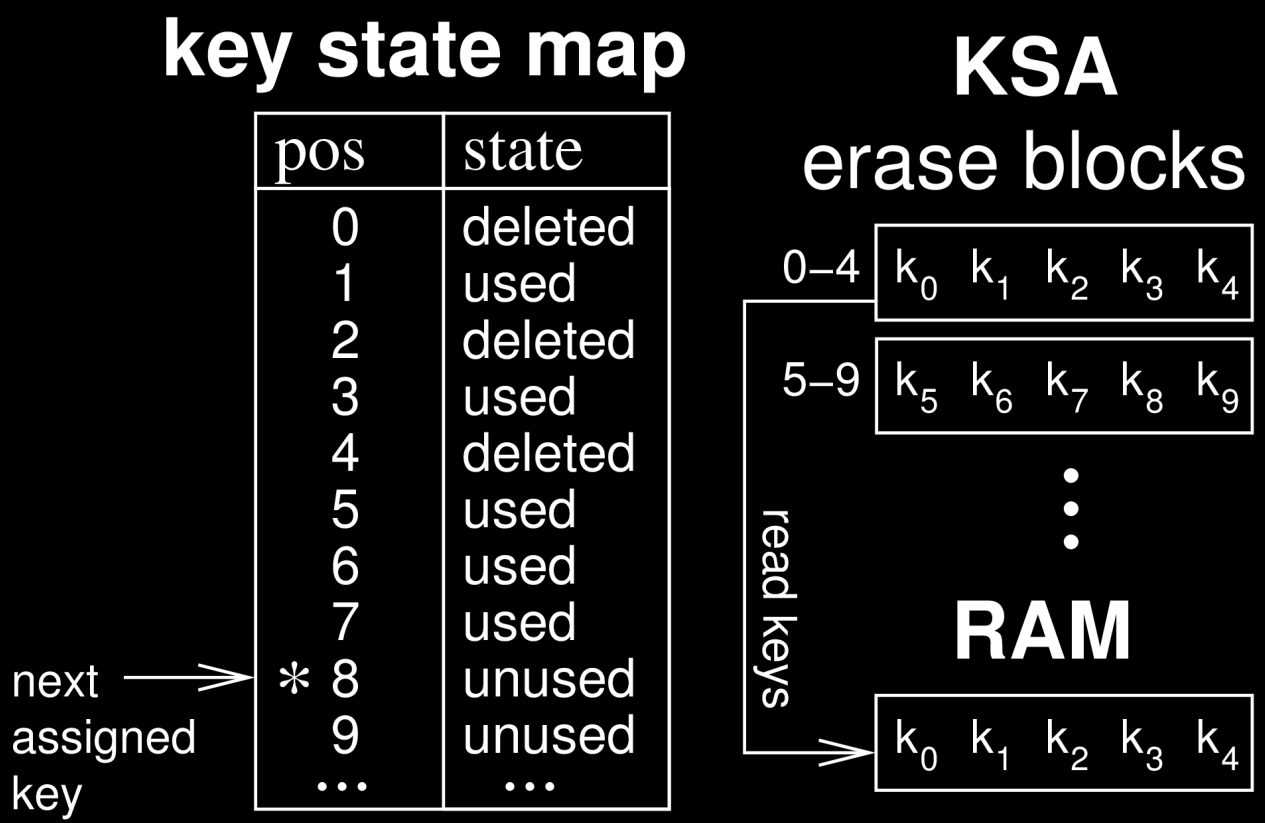

KSA
main storage
k_1
k_2
k_3
k_4
...
$E_{k_1}(DN_1)$
$E_{k_2}(DN_2)$
$E_{k_3}(DN_3)$
...


k_1
$E_{k_1}(DN_1)$
(3) decrypt and return data
DN_1

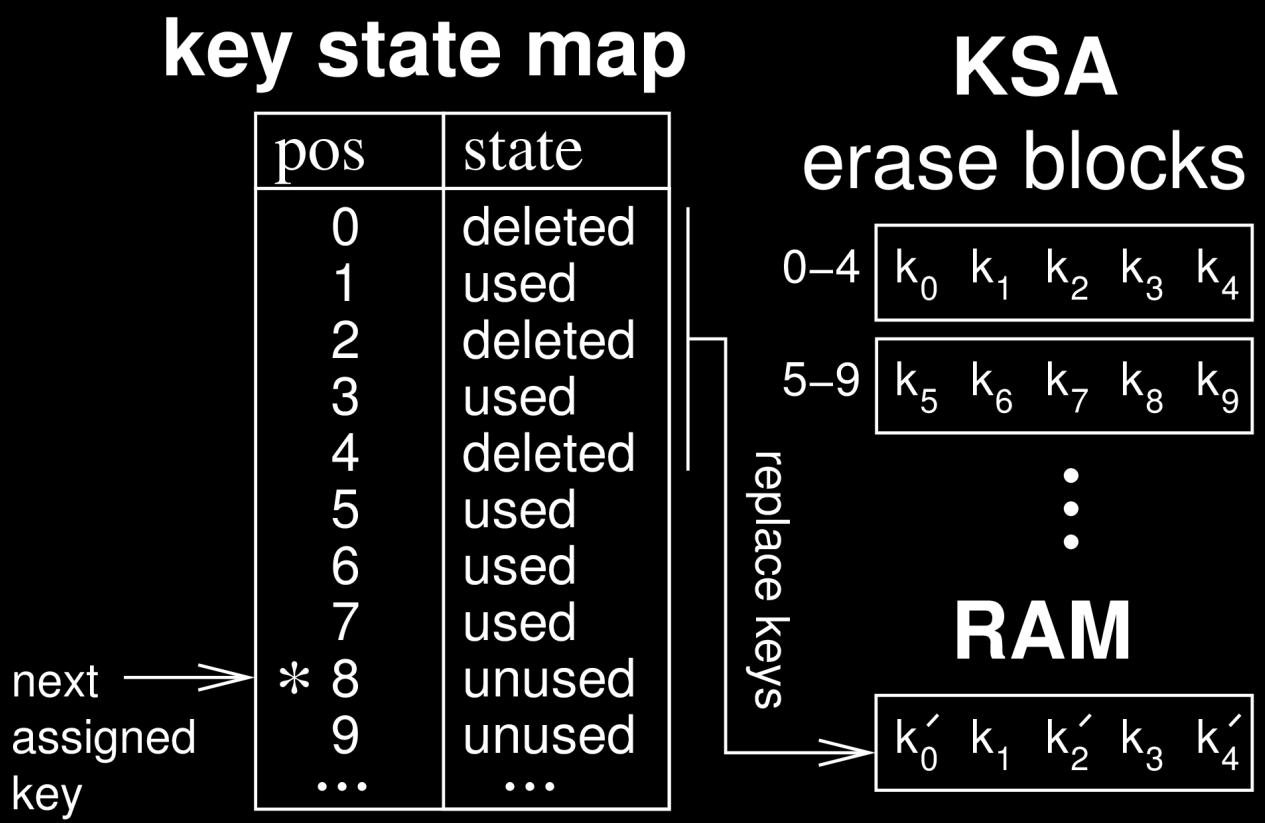
Key State Map and KSA



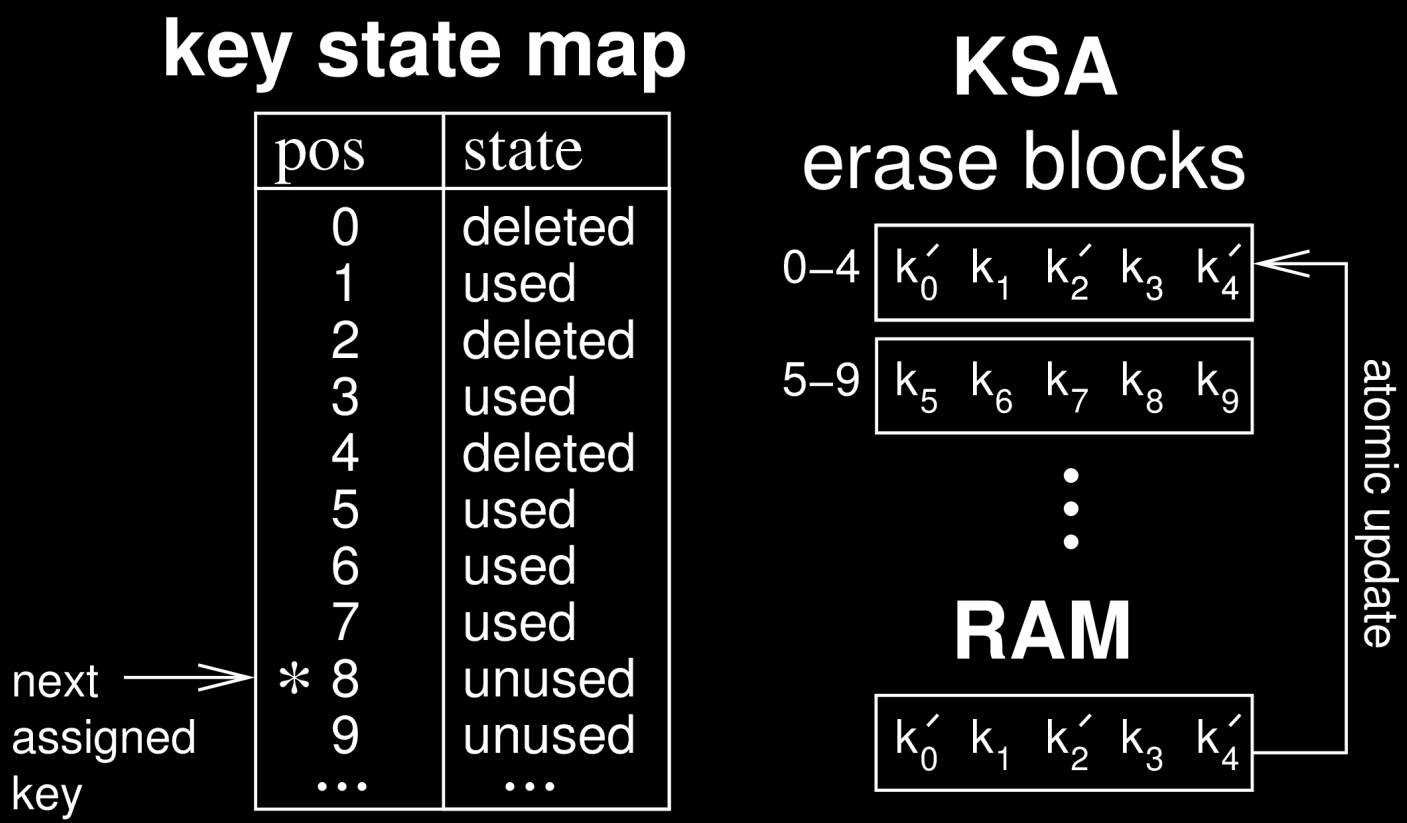
Key State Map and KSA



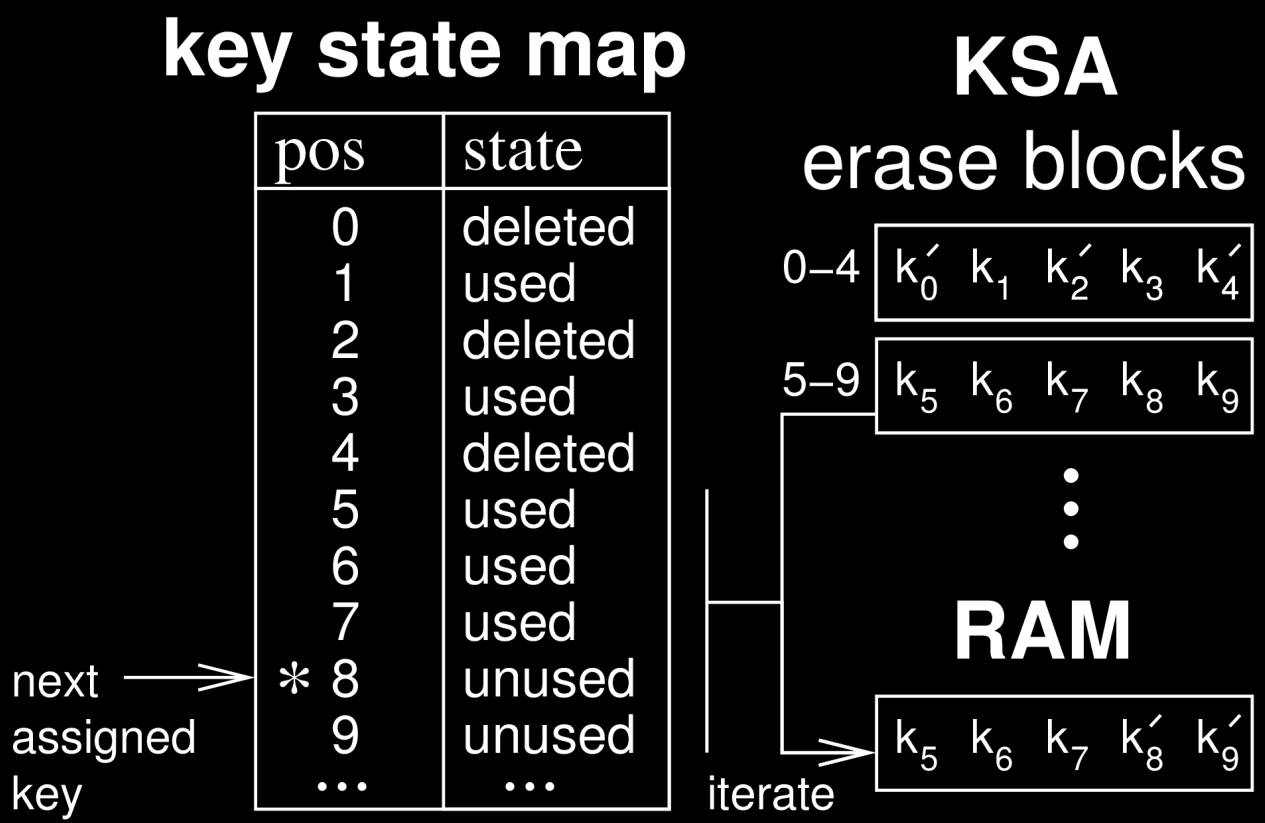
Key State Map and KSA



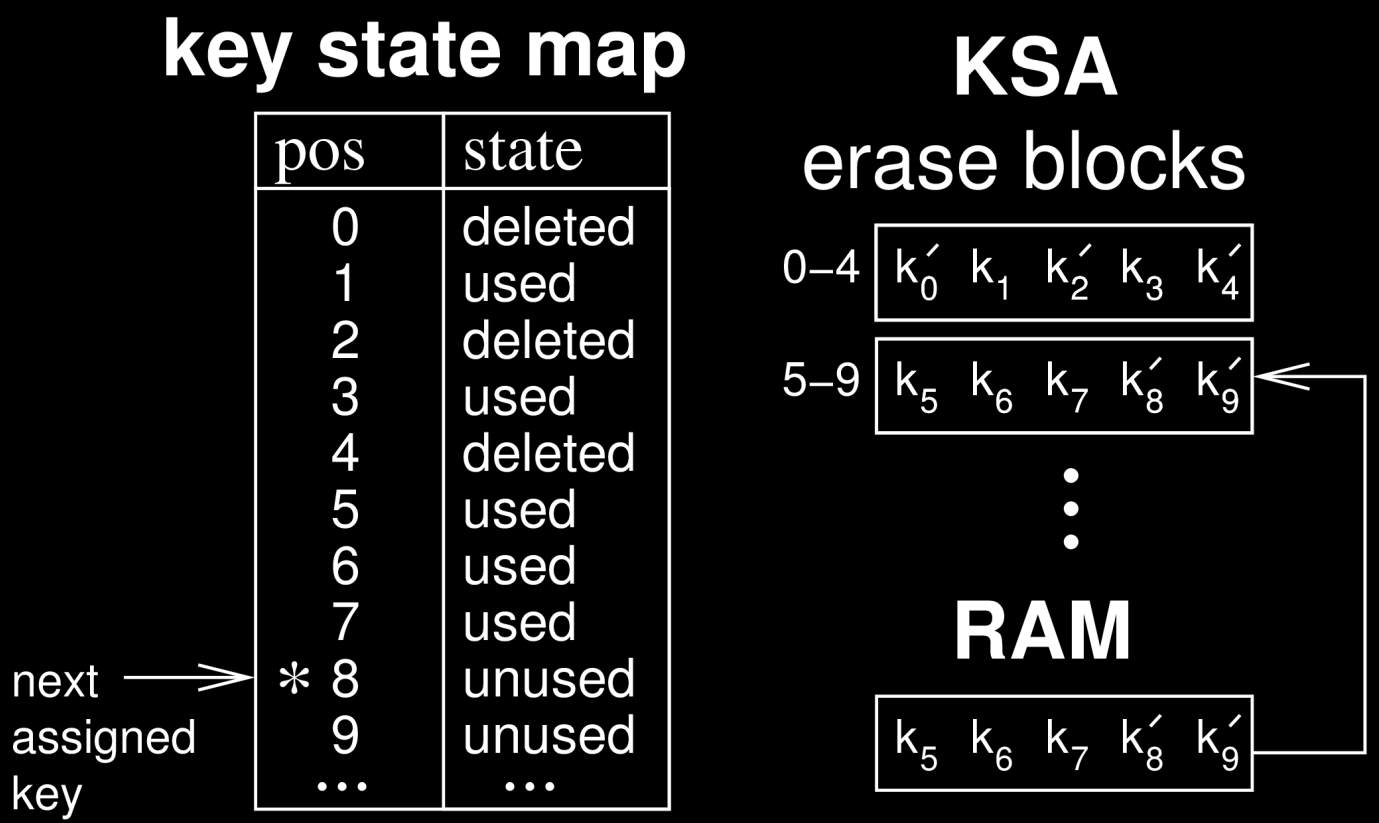
Key State Map and KSA



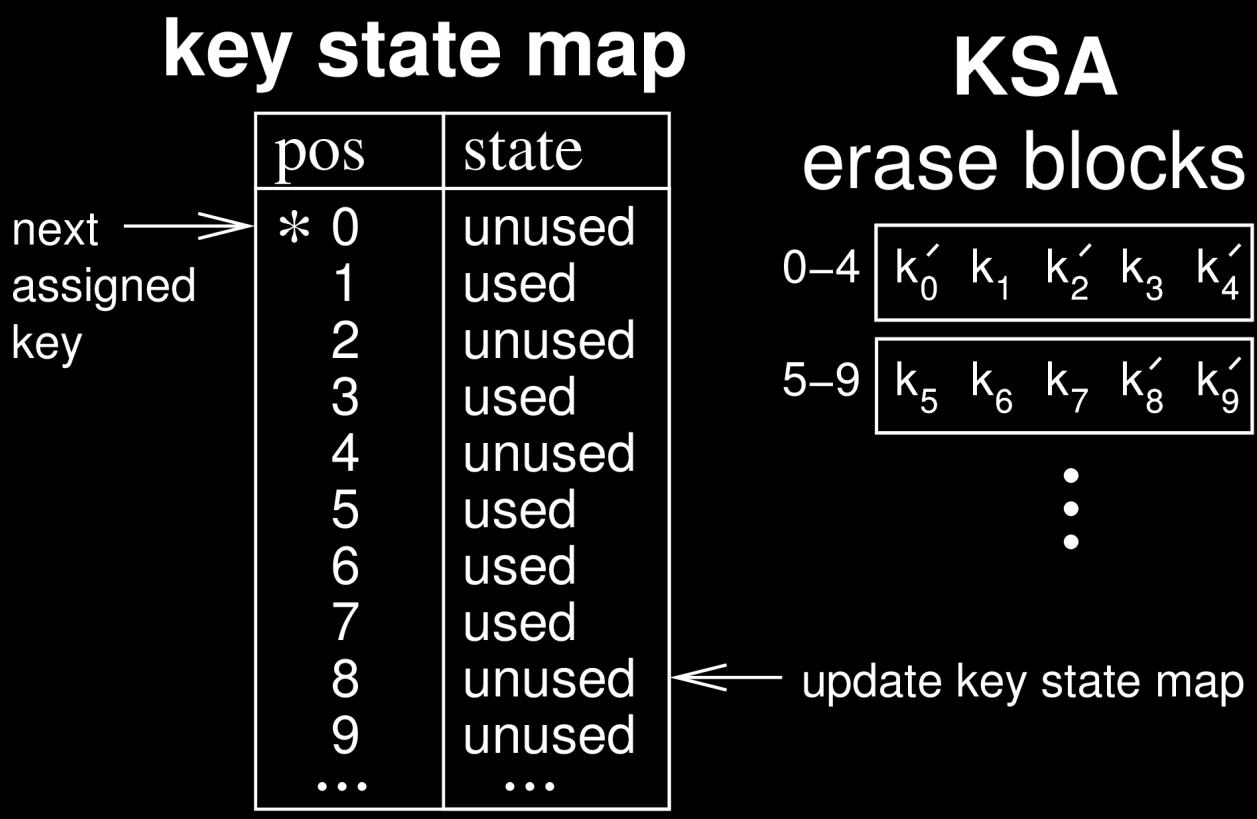
Key State Map and KSA



Key State Map and KSA

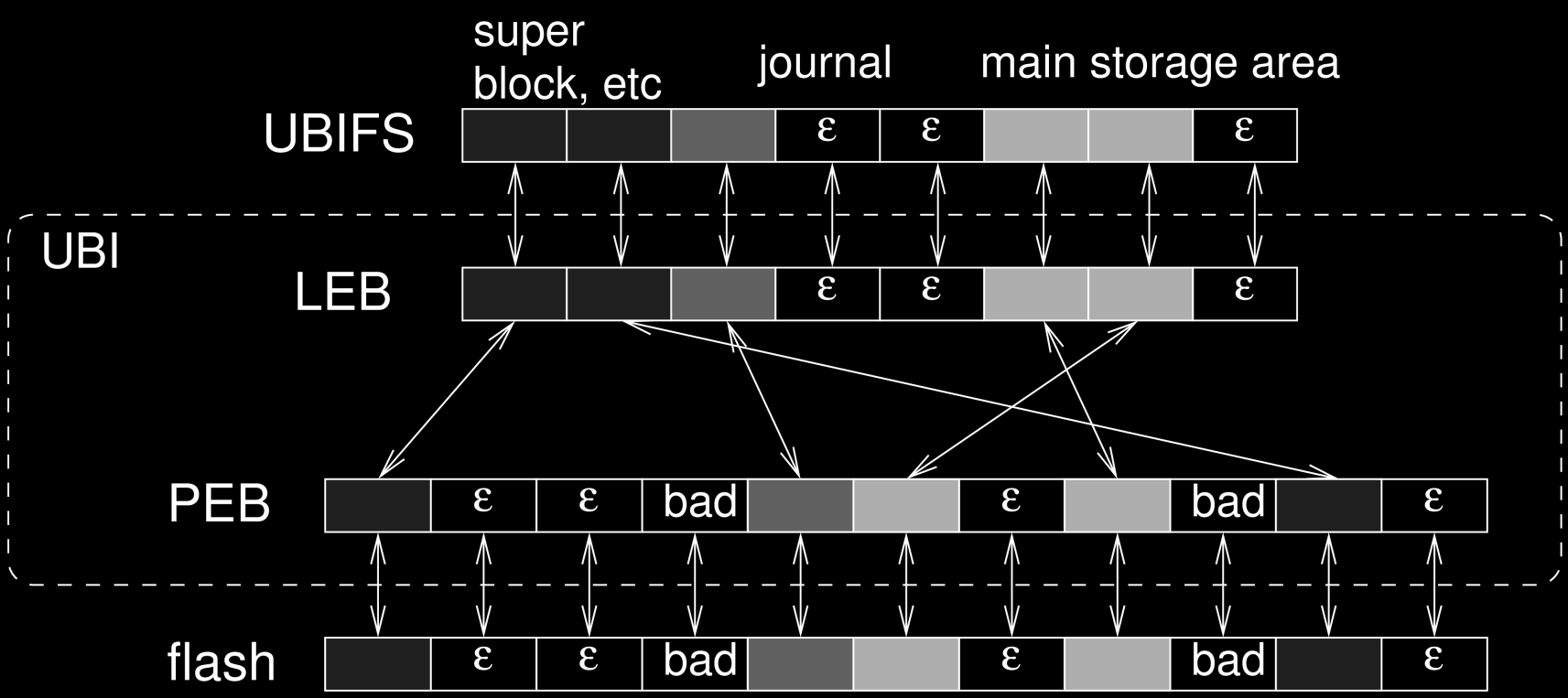


Key State Map and KSA

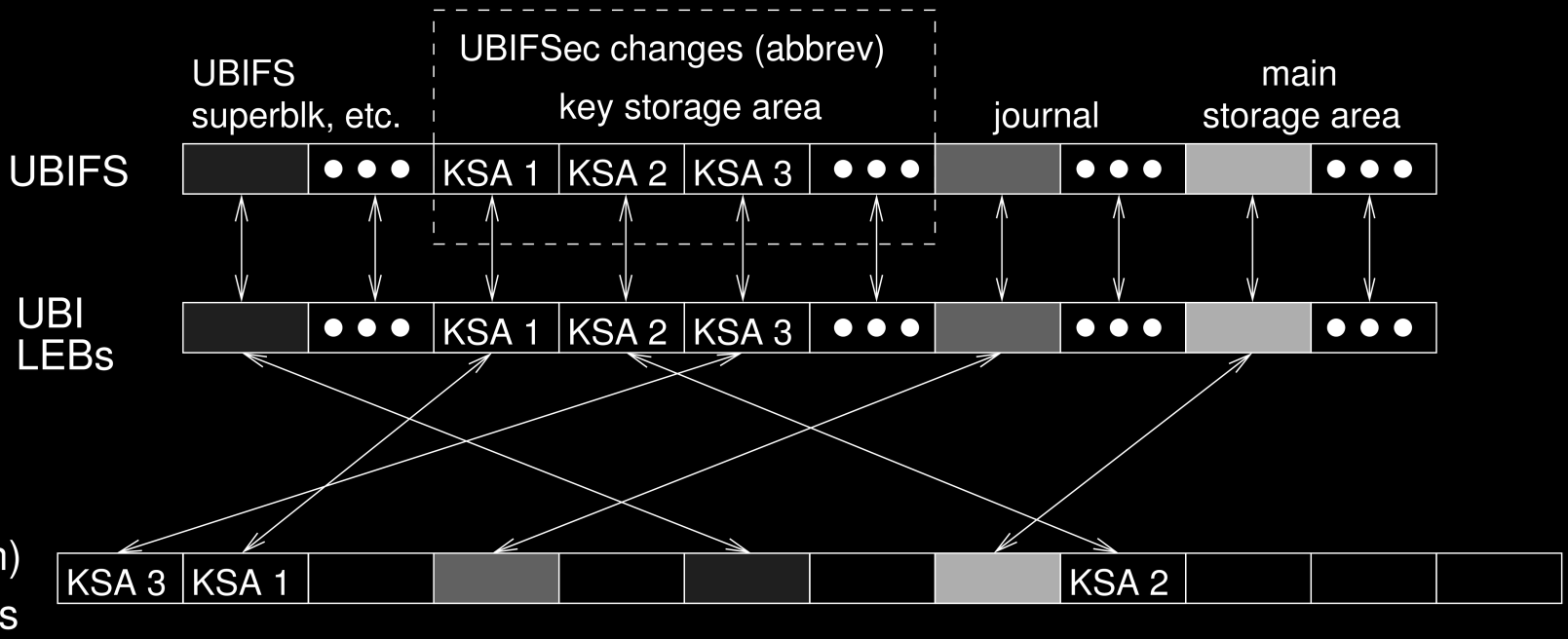


Introducing UBIFSec: our secure deletion implementation for the UBI file system (UBIFS)

UBIFS: on top of UBI



UBIFSec


UBIFSec changes (abbrev)
key storage area
UBIFS
superblk, etc.
journal
main
storage area
UBIFS
KSA 1
KSA 2
KSA 3
UBI
LEBs
KSA 1
KSA 2
KSA 3
(flash)
PEBs
KSA 3
KSA 1
KSA 2

UBIFSec Implementation

- UBI provide logically-referenced KSA, atomic updates with deletion, automatic wear levelling
- DNEFS cryptographic operations during UBIFS compression
- DNEFS integrated with the checkpoint and replay mechanism in UBIFS
- DNEFS key states managed by UBIFS's index
- fully implemented as a single patch, incremental patching ongoing

We tested UBIFSec in simulations and running as file system for a Google Nexus One Android phone.

Erase Block Wear

Purge period	PEB erasures per hour	Lifetime (years)
Stardard UBIFS	21.3	841
60 minutes	26.4	679
30 minutes	34.9	512
15 minutes	40.1	447
5 minutes	68.5	262
1 minute	158.6	113

Throughput and Power Consumption

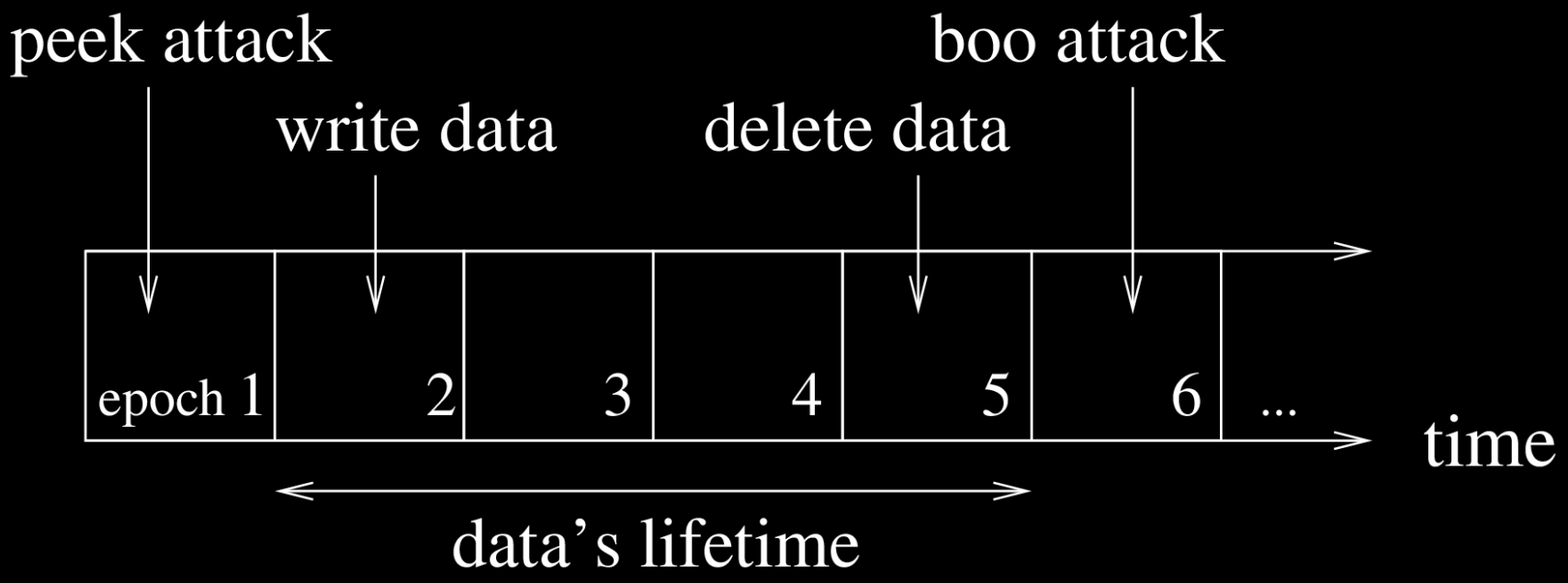
	YAFFS	UBIFS	UBIFSec
Read rate (MiB/s)	4.4	3.9	3.0
Power usage (mA)	39	39	39
GiB read per %	5.4	4.8	3.7
Write rate (MiB/s)	2.4	2.1	1.7
Power usage (mA)	30	46	41
GiB written per %	3.8	2.2	2.0

Summary

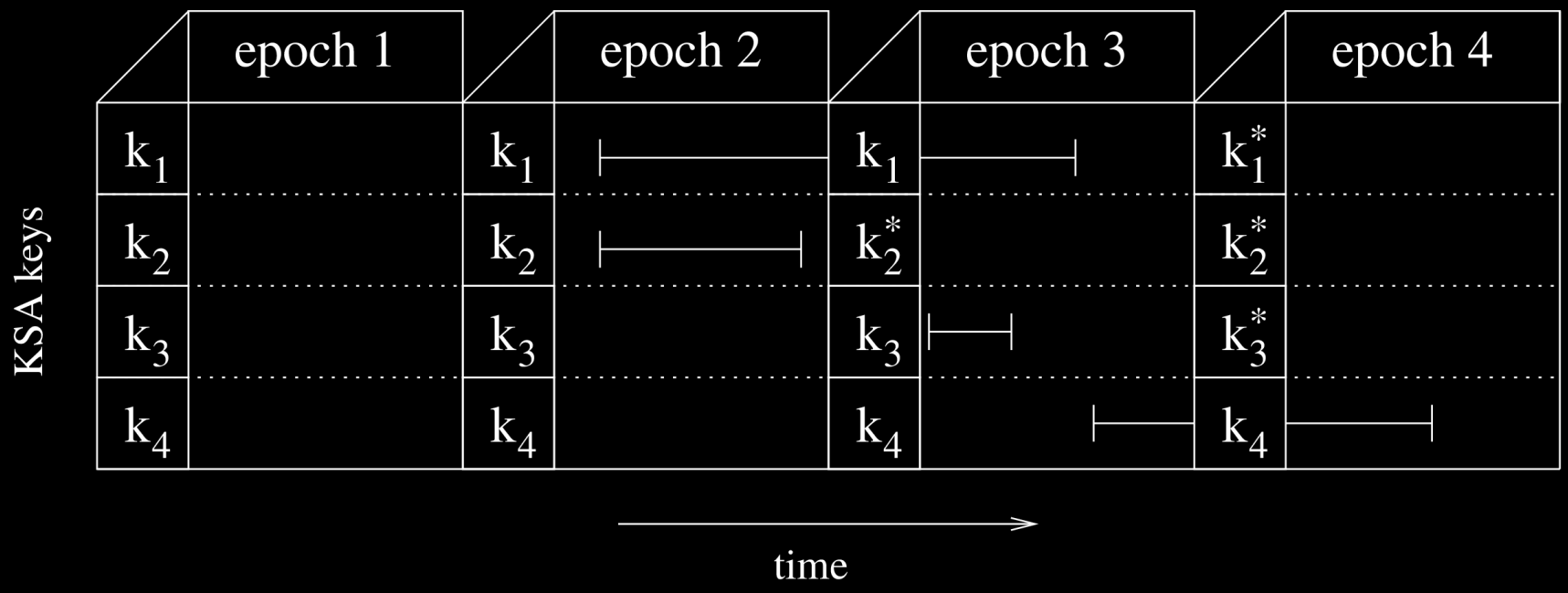
- secure deletion for flash memory problem is not straightforward
- we propose DNEFS: secure deletion by periodic purging of a small key area
  - each data node is stored encrypted, with its key in the key area
  - provides guaranteed fine-grained secure deletion against computationally-bounded adversary
- we implement DNEFS: UBIFSec extends UBIFS to include our design
  - fully implemented into the file system without sacrificing features
  - additional wear, space and computation are reasonable
  - UBIFSec runs normally on an Android phone
- DNEFS can also be integrated into hardware flash controllers as well as software flash file systems
- extended to an encrypted file system by simply encrypting the KSA with a passphrase

Why do we replace both unused and deleted keys with new random values?

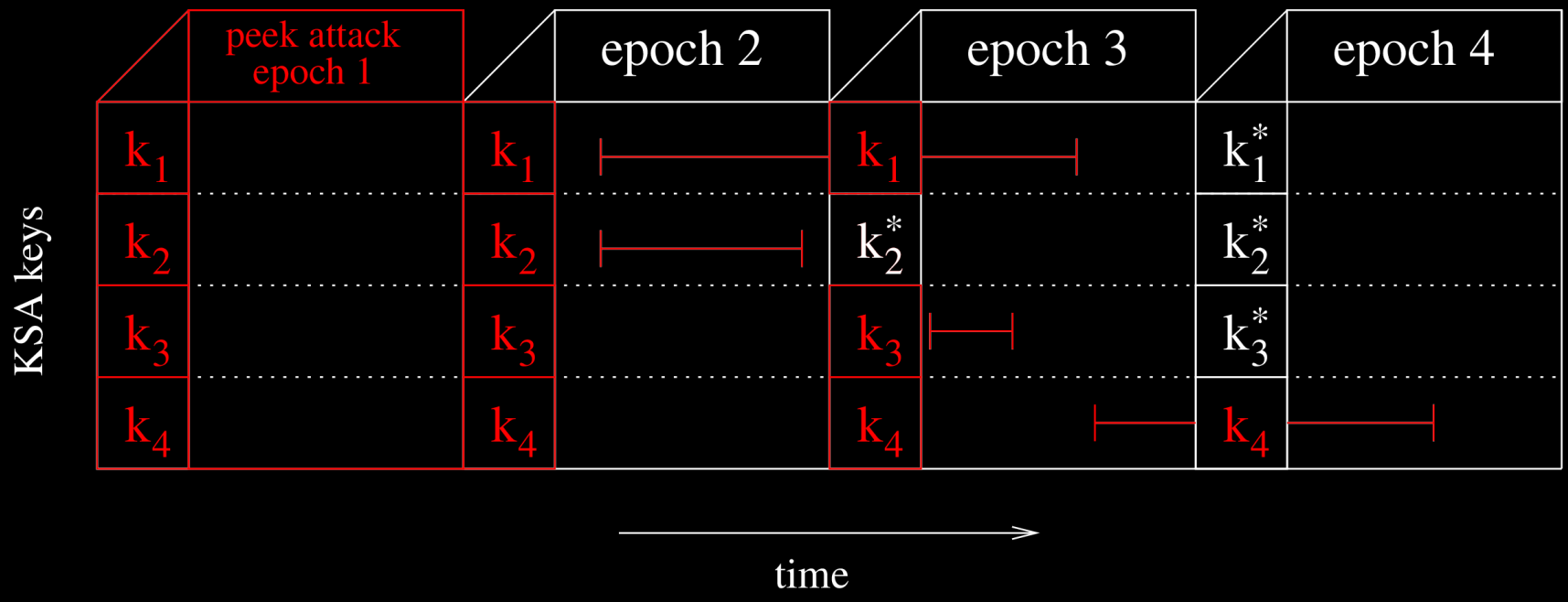
Peek-a-boo Attacker



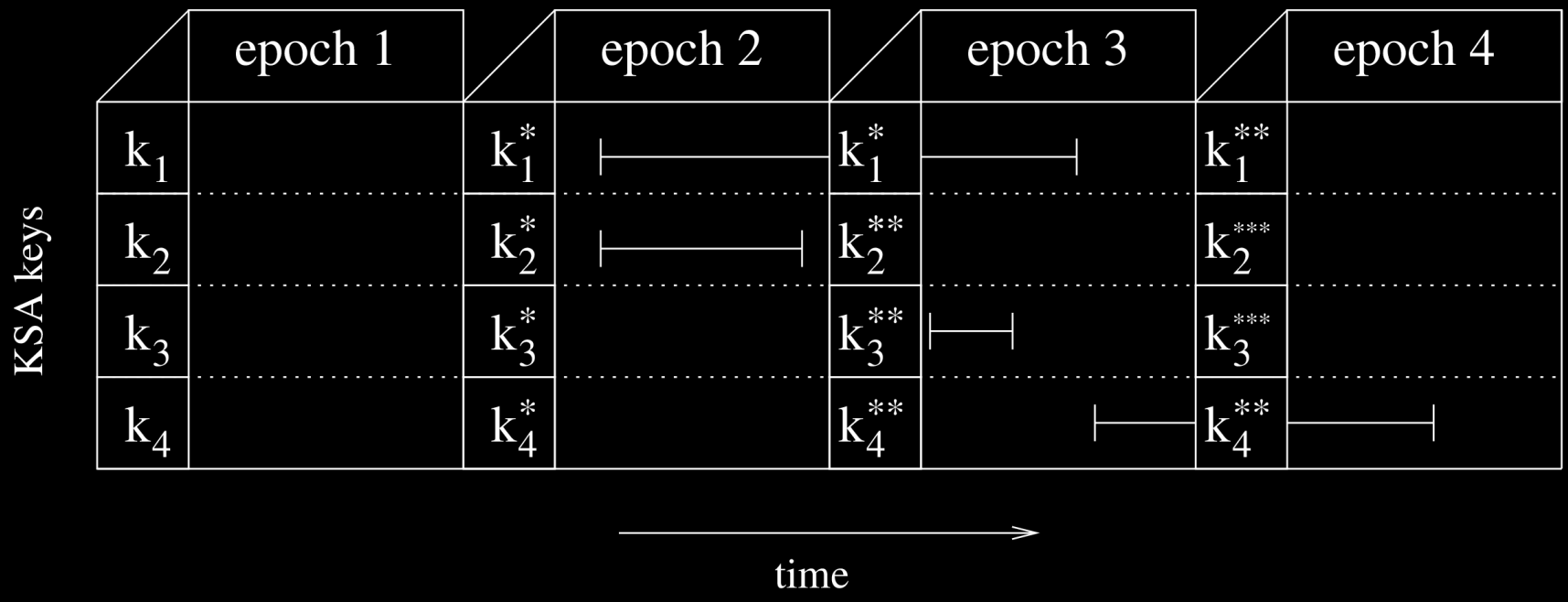
Peek-a-boo attacker without unused replacement



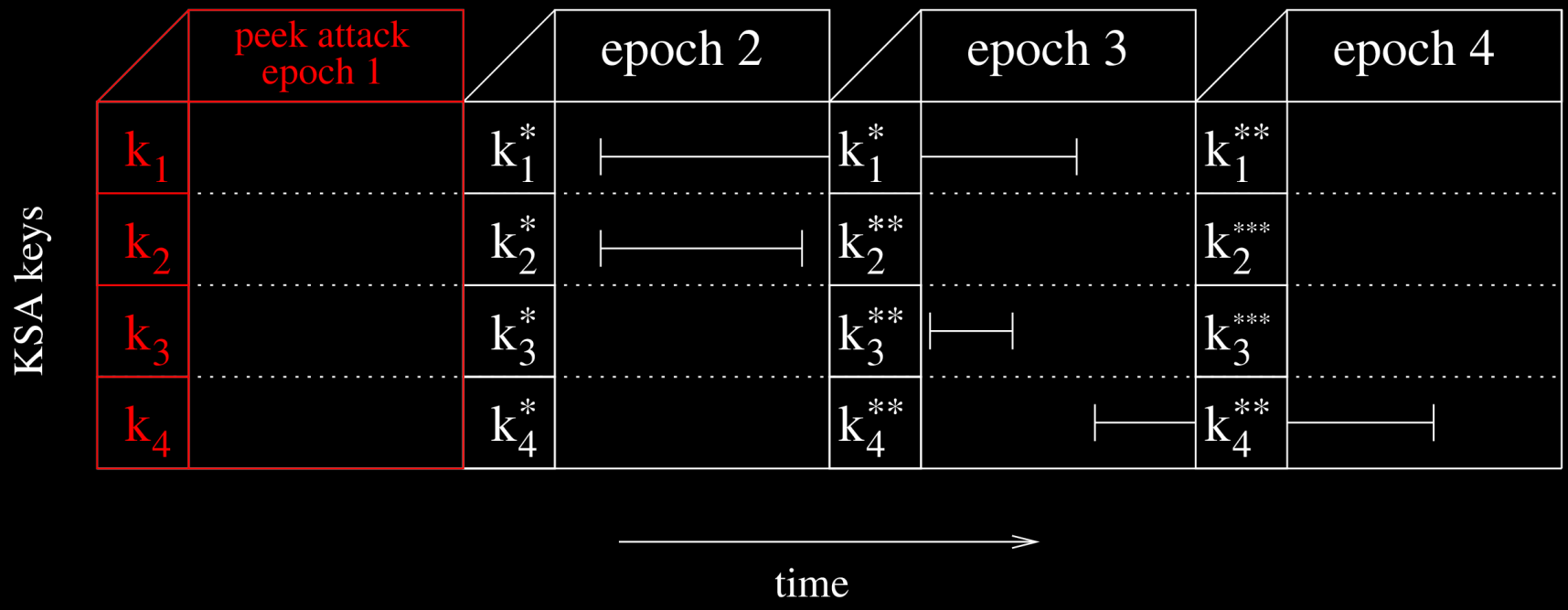
Peek-a-boo attacker performs a peek attack



Peek-a-boo attacker with unused replacement



Peek-a-boo attacker performs a peek attack



Encrypting Whole File

- we still need seek()
- ECB: semantic security
- CTR-like, CBC-like: efficient modifications
- CTR-like, CBC-like with IVs per datanode: our solution

Data Node Size

Data node size (flash pages)	KSA size (EBs per GiB)	Copy cost (EBs)
1	64	0
8	8	0.11
64	1	0.98
512	0.125	63.98
4096	0.016	511.98

Optimization: long-term and short-term data

- each time we GC a data node, we may promote it to a higher range of KSA
  - KSA is divided into ranges of expected life time
- we promote by heuristics: how many times we've had to copy the data around
- getting a new key is low-cost: we have to anyway copy the data

Adding support for encrypted file system

- currently, all the data is encrypted, but the keys are plaintext
- trivial change to turn it into password-protected volume
  - encrypt the entire KSA with a single key derived from a password
  - more efficient than to have a second encryption layer on top

Generalizing to FTL

- our solution is a general technique
  - encrypt blocks at smallest granularity
  - colocate keys in a logically-referenced migrating KSA
  - periodically update the KSA's blocks to delete data
- could be extended to Flash Translation Layer (FTL)
  - used for SD card, USB sticks, etc.
  - maps logical sectors to flash addresses
  - allows normal (e.g., FAT) file systems to be mounted
  - vary in implementation, but all the same principle

Generalizing to FTL

- in the mapping of sector to flash address, also put a key position
  - when mounting, after this mapping is built, then determine the set of used keys
- reserve a set of erase blocks for storing keys
  - last page of each block has a magic number, logical KSA number and purging epoch number
  - periodically purge the KSA
- file system must issue TRIM commands to the FTL to notify unused sectors
  - should be the case regardless