File systems

CSE 536 Fall 2026

jedimaestro@asu.edu

Forensics and TOCTTOU are good ways to learn about filesystems...

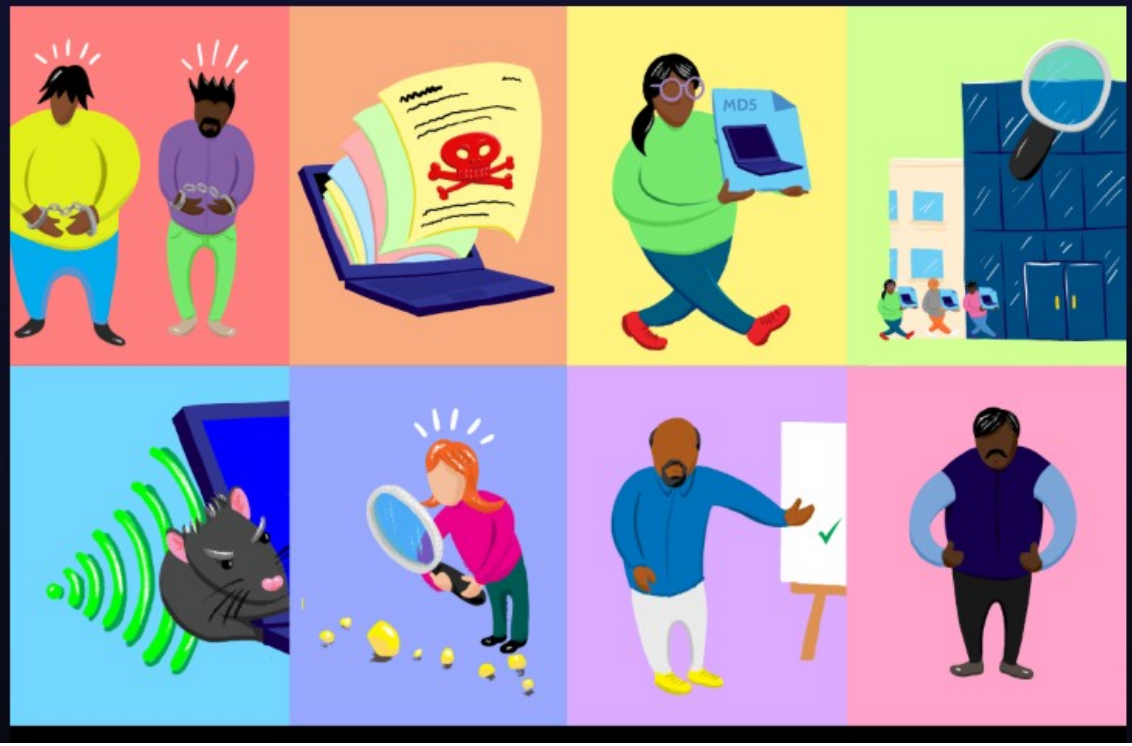
NET ALERT

ALERT!

Arrests Made Over Evidence Planted by Hackers

Also available in

Français বাংলা मराठी తెలుగు हिन्दी اُردُو ગુજરાતી

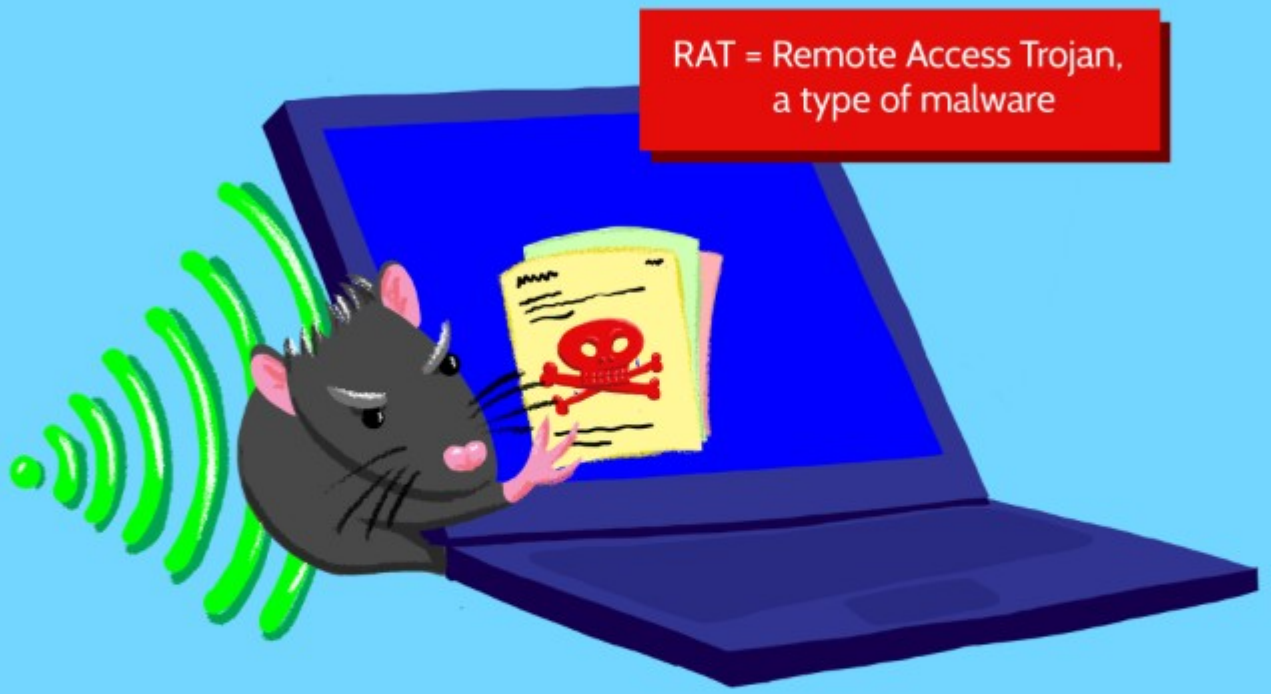


Between 2018 and 2020, sixteen activists, academics, lawyers and poets were arrested in India after authorities discovered incriminating documents on their laptops.

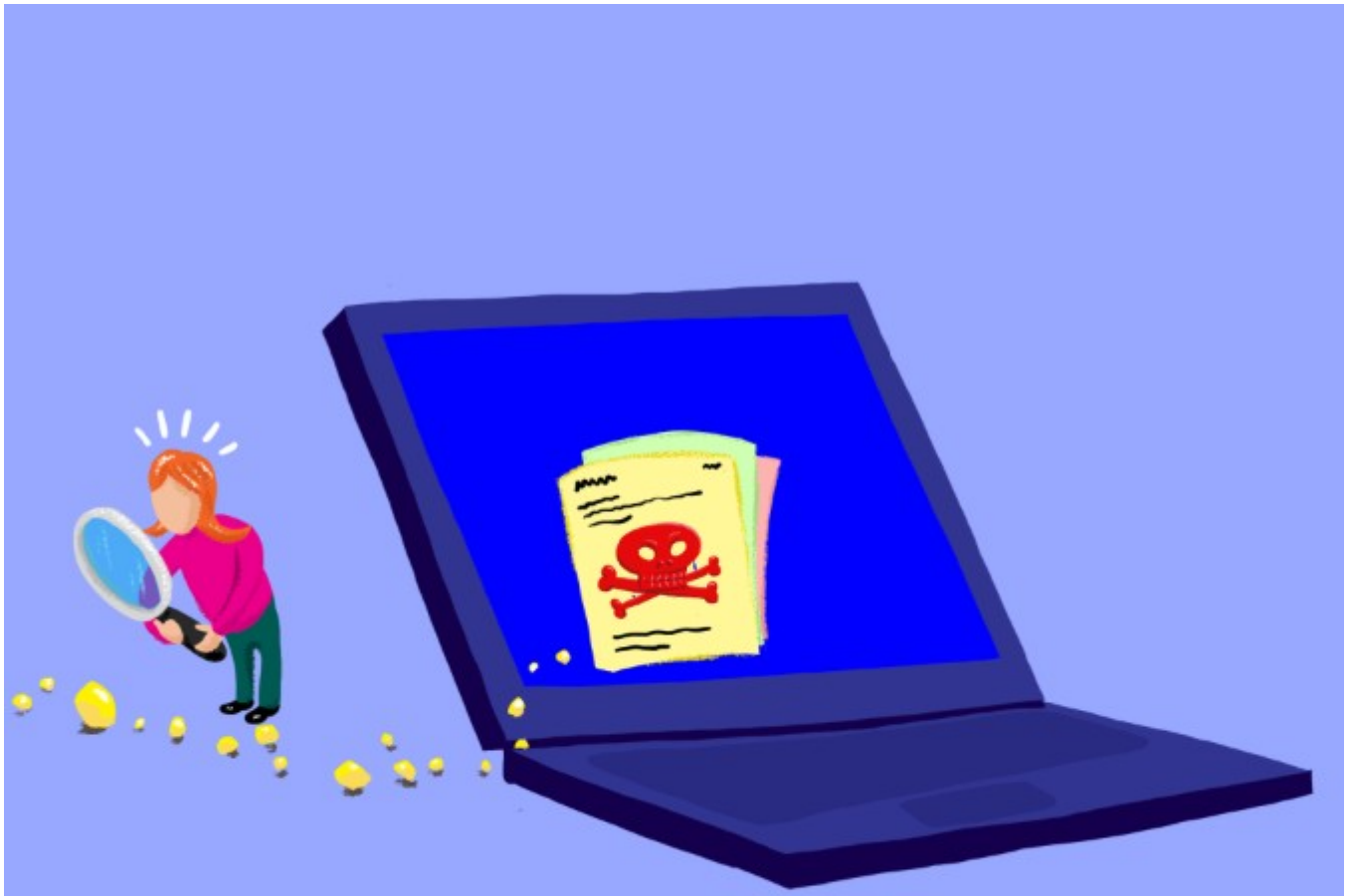
These documents called for violence and included an exchange about firearms with the banned Maoist party.

But defense lawyers helping those arrested begged to differ. They gathered up those laptops with incriminating evidence...

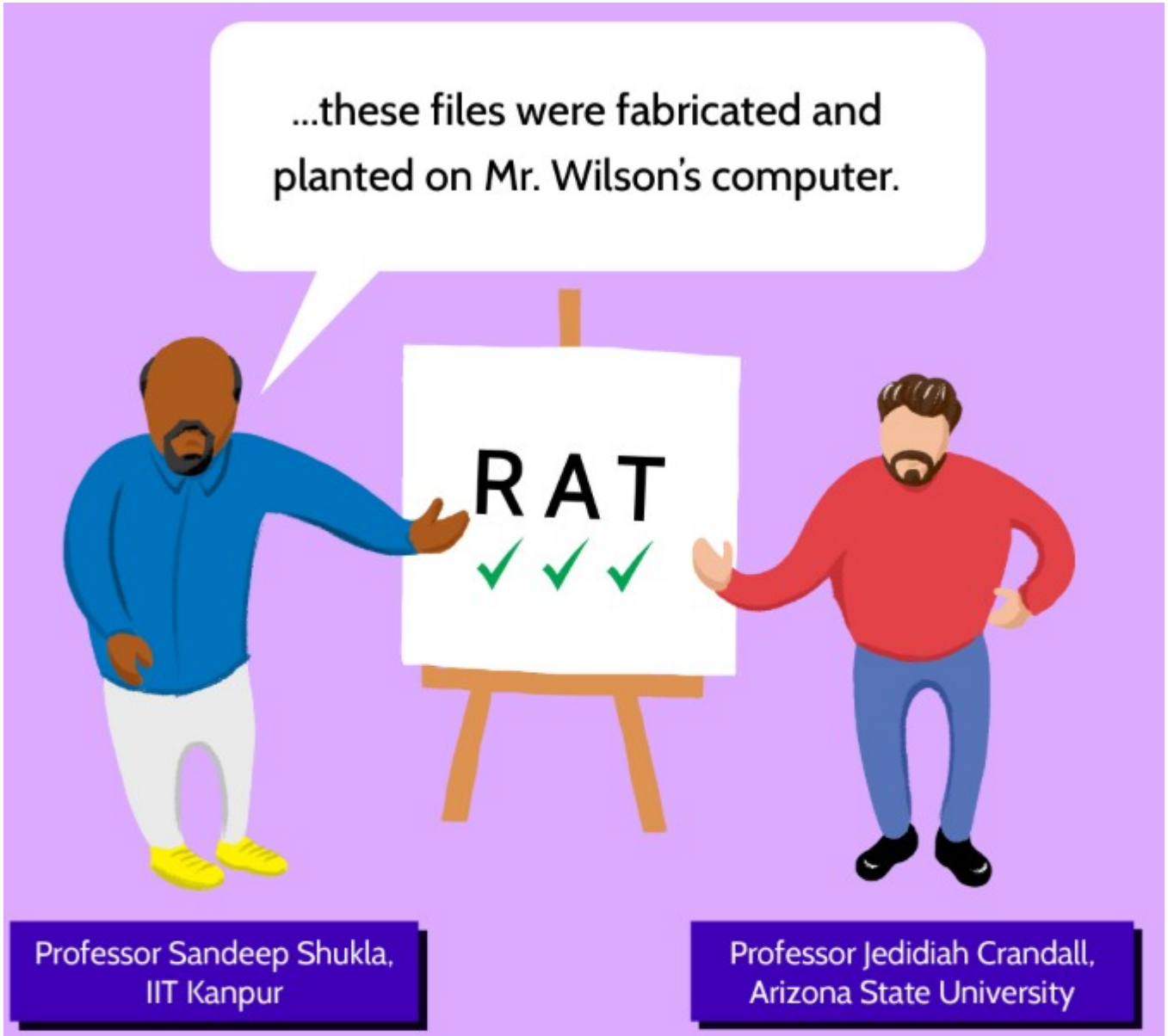
And sent it to Arsenal, a digital forensics firm in the US, for a second opinion.



They found evidence that RATs had secretly slipped those documents into the laptops over the internet.



Luckily for the forensics team, the RAT left a trail of crumbs in its wake.



Other experts agree with Arsenal's findings, which were first published in February of 2021.

Yet the National Investigation Agency disagrees. They maintain that they see no evidence of hacking.

So the activists remain in jail, awaiting trial, while the RAT hackers are still on the loose.

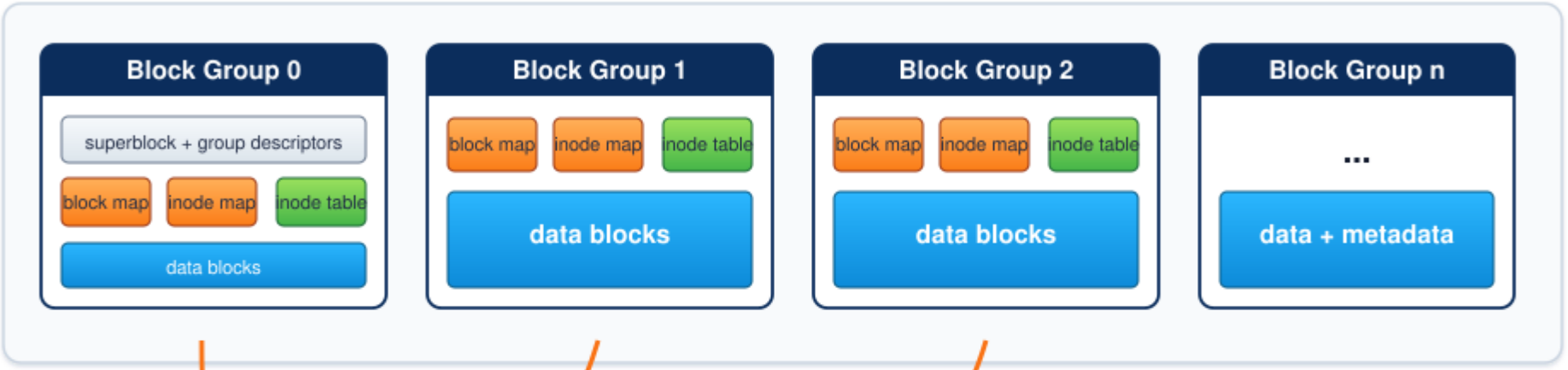
Was a Windows laptop, but lets look at the ext4 filesystem and UNIX data structures for filesystems...

ext4 Filesystem: Bird's-Eye View

A disk is organized into block groups, with inodes, extents, bitmaps, data, and a journal

On-disk volume layout

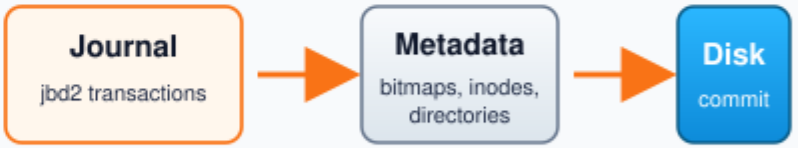
not drawn to scale



How a file is found

Big idea: inodes describe files; extents describe where file bytes live.

Crash recovery path



Usually metadata is journaled; data journaling is configurable.

Major points

Block groups
locality and repeated layout

Bitmaps
track free blocks/inodes

Inodes
own metadata and extents

Extents
map file ranges to blocks

Journal
replays committed updates

data blocks · allocation maps / journal · inode structures · filesystem descriptors / metadata

Journal the metadata

The 4 Steps to Safely Commit Data

- **1. Journal Write (The Plan):** The filesystem writes the intended changes (metadata, and sometimes the actual data) into a dedicated, hidden log file called the journal. At this stage, the main filesystem storage remains untouched.
- **2. The Commit Marker (The Guarantee):** Once the write to the journal is entirely complete, the filesystem appends a special "commit marker" to the end of the journal entry. This marker proves that the entire plan was safely recorded.
- **3. Main Storage Write (The Execution):** Now that the plan is safe, the filesystem writes the changes to their permanent locations in the actual storage blocks.
- **4. Checkpoint (The Cleanup):** After the main storage is successfully updated, the filesystem marks that journal entry as clean or deleted. The space in the journal can now be reused for future operations.

ZFS: Bird's-Eye View

Datasets sit on a pooled, checksummed, copy-on-write storage engine

Logical view

Dataset
filesystem + properties

zvol
block device volume

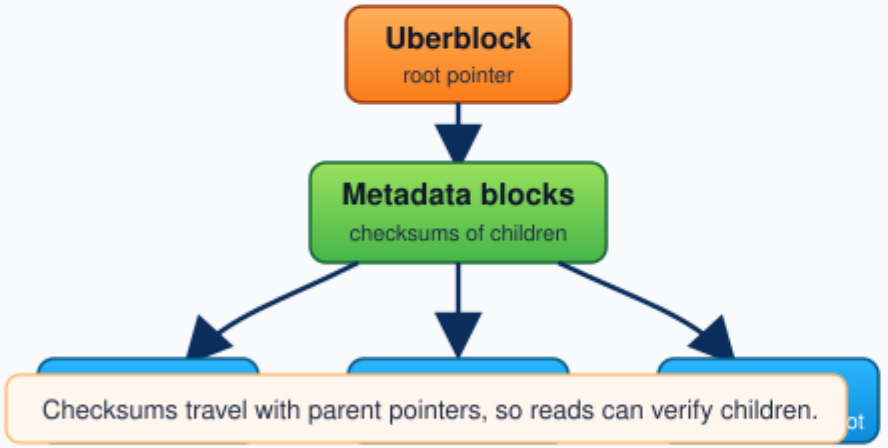
Snapshot
cheap point-in-time view

Clone
writable snapshot child

Properties
compression, quotas, encryption

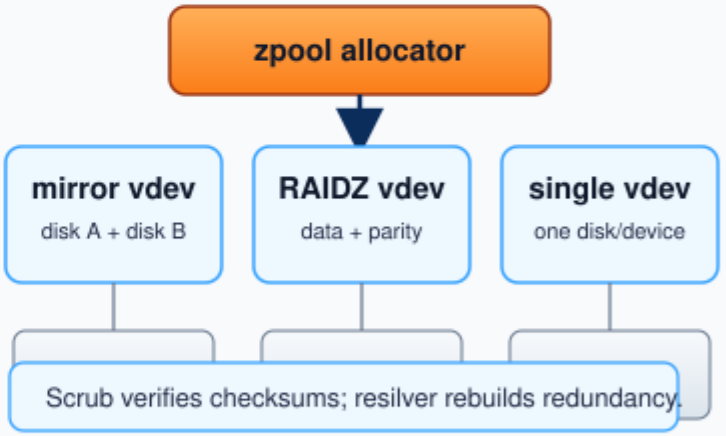
Copy-on-write block tree

Writes allocate new blocks, then atomically swing the root to the new tree.



Storage pool

A zpool aggregates one or more vdevs; redundancy lives inside vdevs.



Operational pieces

ARC
RAM read cache

L2ARC
optional cache device

ZIL / SLOG
sync-write intent log

Checksums
detect bad reads

Snapshots
share old blocks

data blocks · roots / logs / allocation · metadata / checksums · datasets and volumes · snapshots and clones

The Big ZFS Point

ZFS treats storage as unreliable, then verifies and repairs it

Sun
original home

Filesystem + Volume Manager + Integrity Engine
ZFS owns layout, verification, and repair.

FreeBSD
major adopter

Filesystem
datasets, files, snapshots

→

Volume Manager
zpools, vdevs, RAIDZ

→

Integrity Engine
checksums and repair

The lecture hook: ZFS checks whether bytes are still the right bytes

data block → parent checksum → verify read → repair

Designed for huge scale
128-bit storage design;
max file size commonly cited as 16 EiB.

One sentence to remember
ZFS integrates layers so it can detect, locate, and fix storage errors.

Logo boxes are simplified placeholders; attach official logo files if exact trademark art is required.

Btrfs: Bird's-Eye View

A copy-on-write filesystem built from B-trees, extents, checksums, and flexible device mapping

Logical view

Subvolume — independent file tree

Snapshot — CoW point-in-time tree

Send / receive — snapshot replication

Properties — compression, qgroups

Files — extents, not block lists

Copy-on-write B-tree forest

Metadata is organized as B-trees; updates write new nodes and roots.

Root tree

Filesystem trees — one per subvolume/snapshot

Extent tree — file ranges

Checksum tree — data verification

Chunk tree — logical to devices

Snapshots share old extents until writes create new extents.

Space and device mapping

Btrfs allocates large chunks, then maps extents into those chunks.

Chunk allocator

Data chunks — file contents

Metadata — tree blocks

System — chunk tree info

device 1 — single/dup/RAID

device 2 — RAID profile

device n — balance can move

Scrub verifies checksums; balance rewrites chunk placement.

Major points

B-trees — roots for metadata views

Copy-on-write — new blocks before roots

Checksums — stored separately

Chunks — logical space to devices

Balance — relocates chunks

extents / data-related trees

roots / allocation / CoW

metadata and properties

subvolumes and snapshots

B-tree vs B+ Tree

Both keep indexes shallow by putting many sorted keys in each node

B-tree

Keys and records may appear in internal nodes or leaves.

Good mental model

Search can stop at an internal node if the record is there.

Range scans work, but leaves are not necessarily a simple chain.

B+ tree

Internal nodes route the search; records live in the leaves.

linked leaves make range scans fast

Good mental model

Search always descends to a leaf to fetch the record.

Then the leaf chain can stream nearby keys in order.

Why storage systems care

Wide nodes
one disk/page read fans out to many keys

Shallow height
huge indexes still take few hops

B+ trees favor scans
databases and filesystems often need ranges

internal index node · B-tree record node · B+ leaf record node

XFS: Bird's-Eye View

A high-throughput extent filesystem built around allocation groups and B+trees

On-disk layout

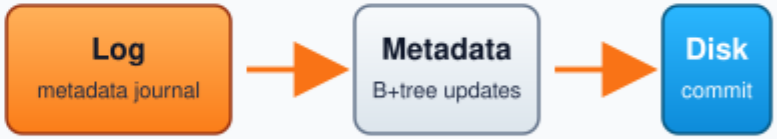
How files map to space

Directories name inodes; inodes point to extents, not one block at a time.

Delayed allocation chooses extents later for better placement.

Journaling and scalability

XFS journals metadata updates; allocation groups reduce global contention.



AG-local B+trees track free space and inode allocation.

Major points

- **Allocation groups** — parallel local regions
- **B+trees** — free space and indexes
- **Extents** — large contiguous ranges
- **Metadata log** — fast crash recovery
- **Realtime option** — separate data volume

An **extent** in XFS is a compact record saying:

```
file offset range -> disk block range
```

So instead of storing “this file uses block 100, 101, 102, 103, ...” one at a time, XFS can say:

```
file blocks 0-999 -> disk blocks 50000-50999
```

NTFS: Bird's-Eye View

Almost everything is a file; records in the Master File Table describe names, attributes, and data runs

Volume metadata files

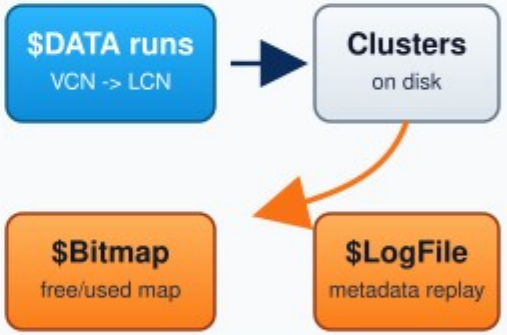
- **$Boot** — volume boot data
- **$MFT** — master file table
- **$Bitmap** — cluster allocation
- **$LogFile** — redo/undo journal
- **$Secure + $AttrDef** — security + attribute defs

Master File Table centric design

Each file or directory has an MFT record made of typed attributes.

Cluster mapping and recovery

Nonresident data uses runlists that map file ranges to disk clusters.



NTFS journals metadata, not ordinary file data by default.

Major points

- **MFT** — central table of records
- **Attributes** — names, data, security
- **Runlists** — extents over clusters
- **Bitmap** — free cluster tracking
- **Log file** — metadata recovery

NTFS: Everything Is a File

The Master File Table stores records for user files and NTFS's own metadata files

$MFT: Master File Table

one record per file or directory

records are bundles of typed attributes

Ordinary file record

A document, program, or directory is described by attributes.

- $FILE_NAME
- $STANDARD_INFO
- $SECURITY_DESC

Security metadata travels as an attribute of the file.

Filesystem metadata files

NTFS's bookkeeping also appears as reserved files.

- **$MFT** — all records
- **$Bitmap** — free clusters
- **$LogFile** — metadata journal
- **$Boot** — volume boot data

The dollar sign marks reserved NTFS system files.

For file contents, $DATA can be resident or point to runs

Small data may fit inside the MFT record; larger data uses runlists to map file ranges.

$DATA → clusters

file data / runs · MFT and security attributes · allocation map / journal · standard volume metadata

In the BK16/Bhima Koregaon reports, Arsenal used NTFS artifacts to reconstruct how documents appeared on the machines. The big ones were:

`$UsnJrnl`
This is the NTFS **change journal**. It records filesystem transactions like `FILE_CREATE`, `DATA_EXTEND`, `DATA_OVERWRITE`, `CLOSE`, and `FILE_DELETE`. Arsenal used recovered `$UsnJrnl` records from allocated and unallocated space to show a sequence: RAR archives and UnRAR executables were temporarily created, used to unpack documents, and then deleted. That pattern supported their conclusion that documents were delivered by an attacker, not normally created by the user. The report says the NTFS transaction data "clearly demonstrates the attacker's modus operandi." Arsenal Report III via Washington Post ↗

`$OBJECT_ID`
NTFS can assign object identifiers to documents when they are created or first opened. Arsenal noted that none of the 14 important documents had object identifiers, which they treated as evidence that the documents had not been opened or legitimately interacted with on that computer. Arsenal Report III via Washington Post ↗

Unallocated/slack remnants
Even after deletion or reinstallation, NTFS/Windows artifacts could remain in unallocated space, slack space, hibernation slack, prefetch files, registry remnants, and related system traces. Arsenal used those leftovers to reconstruct malware persistence, NetWire activity, WinSCP staging, document delivery, and cleanup attempts.

So the lecture-friendly point is:

> NTFS helped because "everything is metadata-rich": file creation, modification, deletion, object IDs, paths, and system activity can leave independent traces that let investigators reconstruct a timeline even when the visible files are gone.

Filesystem Choices: Bird's-Eye View

The practical differences are mostly about allocation, integrity, snapshots, and operating-system fit

Rule of thumb: pick the filesystem whose operational model matches the machine's job.

ext4

Core idea

conservative
Linux default

Use when

general servers,
VMs, root disks

Watch for

less built-in
volume magic

XFS

Core idea

allocation groups
+ extents

Use when

large files,
parallel I/O

Watch for

cannot shrink

Btrfs

Core idea

CoW B-trees
+ subvolumes

Use when

snapshots,
send/receive

Watch for

RAID5/6
caveats

ZFS

Core idea

pooled CoW
+ checksums

Use when

data integrity,
NAS, snapshots

Watch for

vdev planning
matters

NTFS

Core idea

MFT records
+ attributes

Use when

Windows system
and data volumes

Watch for

best fit is
Windows-native

Quick use-case map

Simple Linux default: ext4. Very large local filesystems: XFS. Snapshot-heavy Linux workflows: Btrfs.
Integrity-first storage pools or NAS: ZFS. Windows-native compatibility and system volumes: NTFS.

traditional Linux extent filesystems
Linux CoW filesystem
pooled integrity-first CoW storage
Windows-native filesystem

Block Device, Swap, and Memory Artifacts

A laptop drive can contain both normal files and snapshots of memory pages

Laptop memory

Virtual memory manages RAM as pages.

RAM pages

process data | kernel objects | network state

Some pages are not ordinary files until the VM system writes them out.

page out

RAM -> swap

Laptop drive as a block device

The disk is just numbered sectors/blocks; partitions give regions meaning.

/dev/nvme0n1 or /dev/sda

partition 1: ext4 | **swap**

Forensic images preserve both filesystem and non-filesystem regions.

Why this matters in forensics

ext4 partition
files, directories, logs

→

swap partition
raw copied memory pages

→

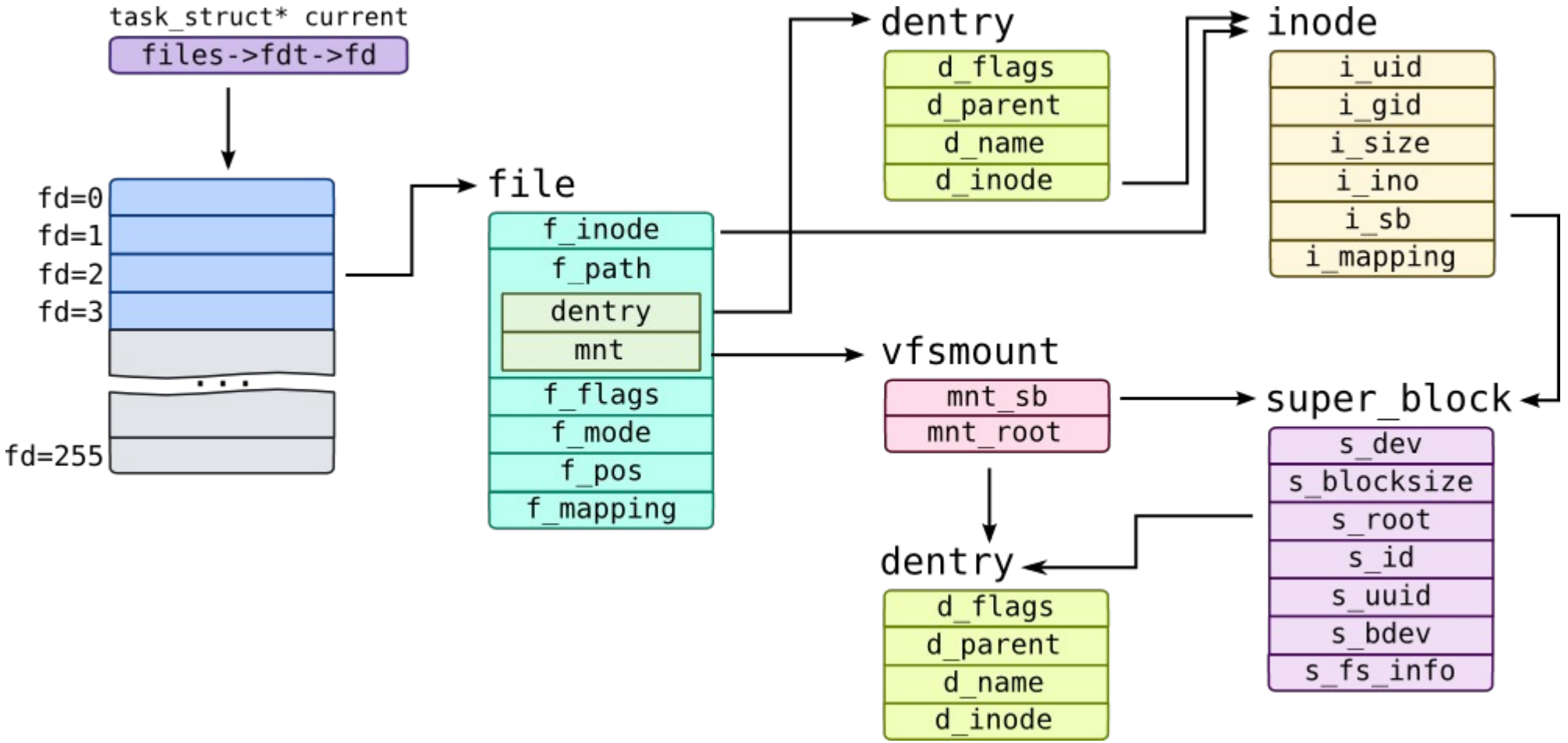
recoverable artifacts
strings, paths, keys, socket buffers, kernel structures

BK16-style lesson: investigators can correlate filesystem traces with memory remnants, hibernation/pagefile/swap data, and network activity.

Important nuance

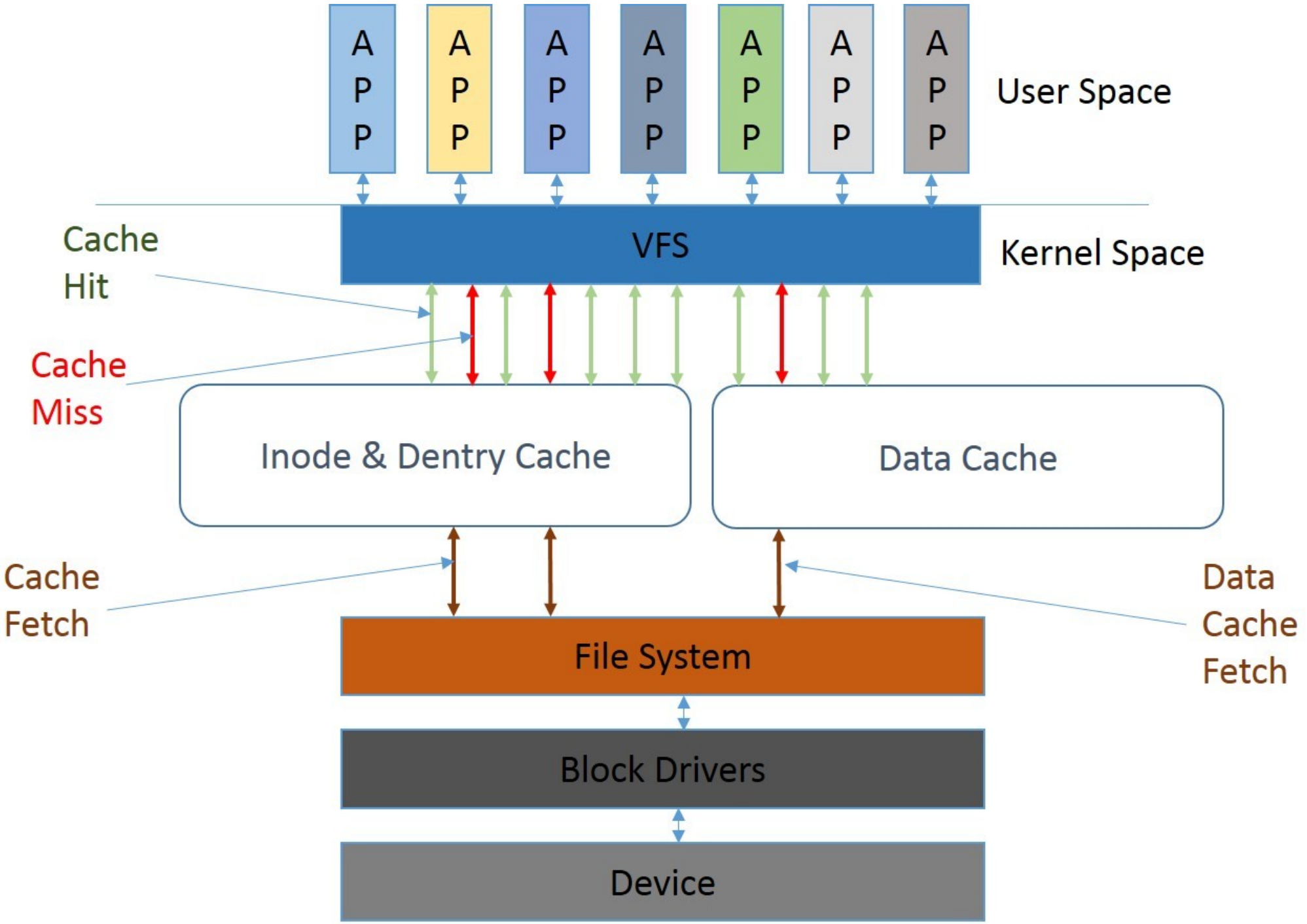
Swap does not store “files”; it stores evicted memory pages. Those pages may contain fragments of programs, documents, or OS/network state.

filesystem-backed data | swap / page-out area | kernel or forensic artifact | process/application memory



https://myaut.github.io/dtrace-stap-book/kernel/fs.html

```
man lsof
man stat
man dd
man df
man losteup
man lsblk
man mkfs.ext4
man mount
man rm
man strings
```



https://msreekan.com/tag/block-device/

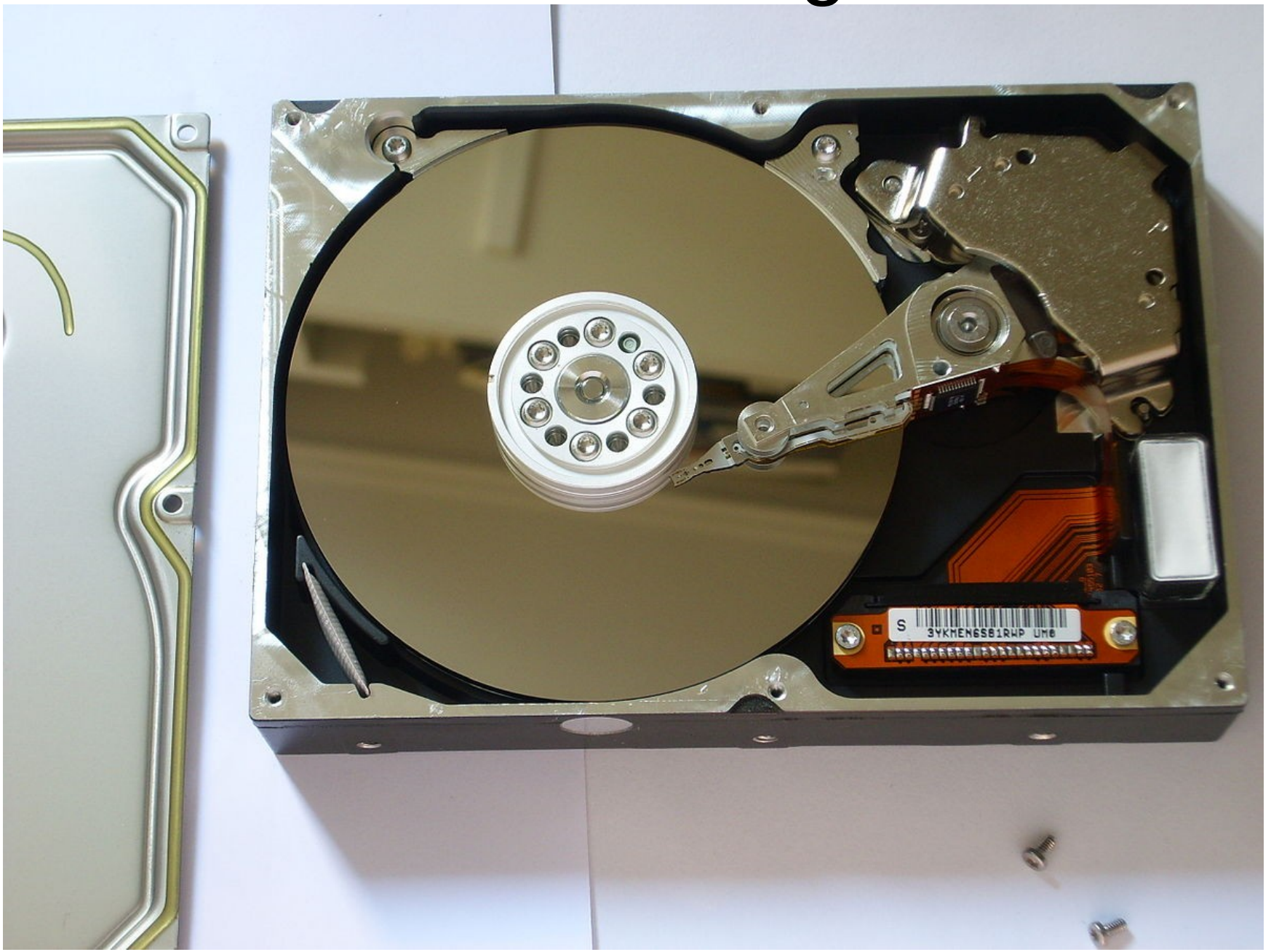
Digital forensics

- According to Wikipedia, you could be looking for: attribution, alibis and statements, intent, evaluation of source, document authentication
- File carving (*e.g.*, bifragment gap carving)
  - Electron microscopes
- Memory forensics (Volatility)
- Network forensics (PCAPs, NetFlow records, NIDS logs)
- Database forensics
- Timestamps in document or log file analysis
- Steganography
- Digital forensic processes
- Benford's law

Forensics tools

- File carvers
  - *E.g.*, Scalpel and foremost
- Log parsers
- Parsers/viewers for different kinds of files
  - SQLite, EXIF, *etc.*
- Linux commands that might be useful:
  - file, exif, sqlite3, losetup, mount, dd, ssdeep, grep, strings

File carving



Alessio Sbarbaro User_talk:Yoggysot - Own work

Memory forensics

```
$ python vol.py --profile=LinuxDebian-3_2x64 -f debian.lime linux_netstat
Proto   Source IP:Port          Destination IP:Port     State           Process
TCP     192.168.174.169:22      192.168.174.1:56705     ESTABLISHED     sshd/2787
TCP     0.0.0.0:22              0.0.0.0:0               LISTEN          sshd/2437
UDP     0.0.0.0:137             0.0.0.0:0               LISTEN          nmbd/2121
[snip]
```

Disk Carving vs. Volatility Memory Analysis

Two different evidence surfaces: raw storage bytes vs. live OS state in RAM

Traditional File Carving

From a disk image, unallocated space, or raw block device

TYPICALLY FINDS

- **Deleted files or file fragments**
  especially recognizable content in unallocated space
- **Known file types by signatures**
  JPEG, PDF, ZIP/DOCX, archives, media, databases
- **Embedded objects and payloads**
  files inside documents, archives, installers, cache blobs
- **Content artifacts**
  images, thumbnails, exports, logs, copied documents

USUAL LIMITS

Mostly recovers file content.

Original filename, path, ownership, and timestamps are often missing unless metadata is also recovered.

Volatility / Memory Forensics

From a RAM capture or hibernation-style memory image

TYPICALLY FINDS

- **Running processes and process tree**
  including command lines and parent/child relationships
- **Network sockets and connections**
  what was listening, connected, or recently active
- **Loaded modules, drivers, DLLs**
  plus suspicious injections or hidden code regions
- **Handles, registry hives, open files**
  OS objects that explain what programs were doing

USUAL LIMITS

Mostly reconstructs system state.

It is a point-in-time snapshot, and results depend on OS structures, symbols/profiles, and acquisition quality.

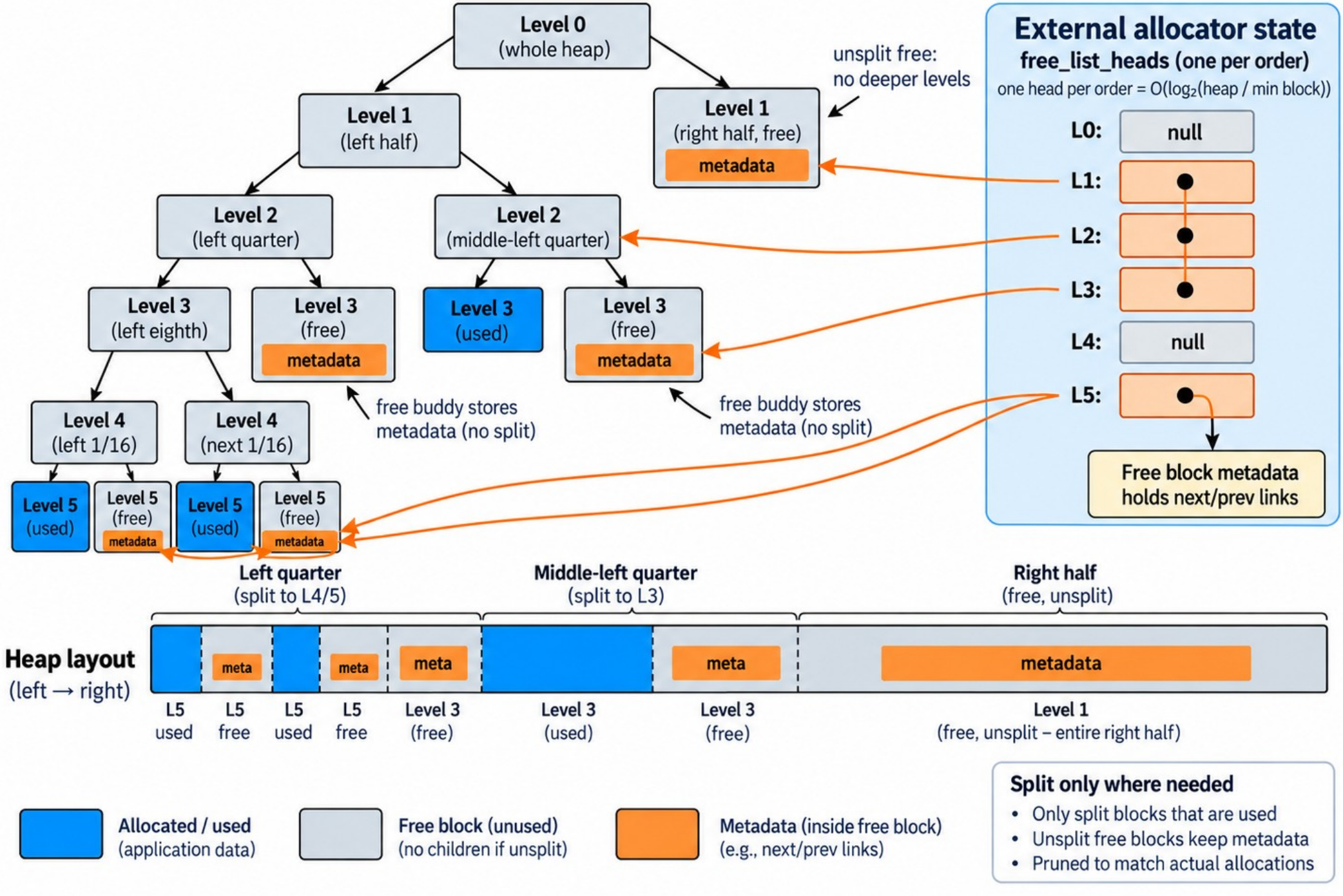
Big lecture point: carving asks "what file bytes are still on storage?" Volatility asks "what was the operating system doing in memory?"

Conceptual separation for teaching: swap/pagefile leakage from RAM into disk is intentionally left out here.

man netstat
cat /proc/NNN/maps
man pstree
man lsof

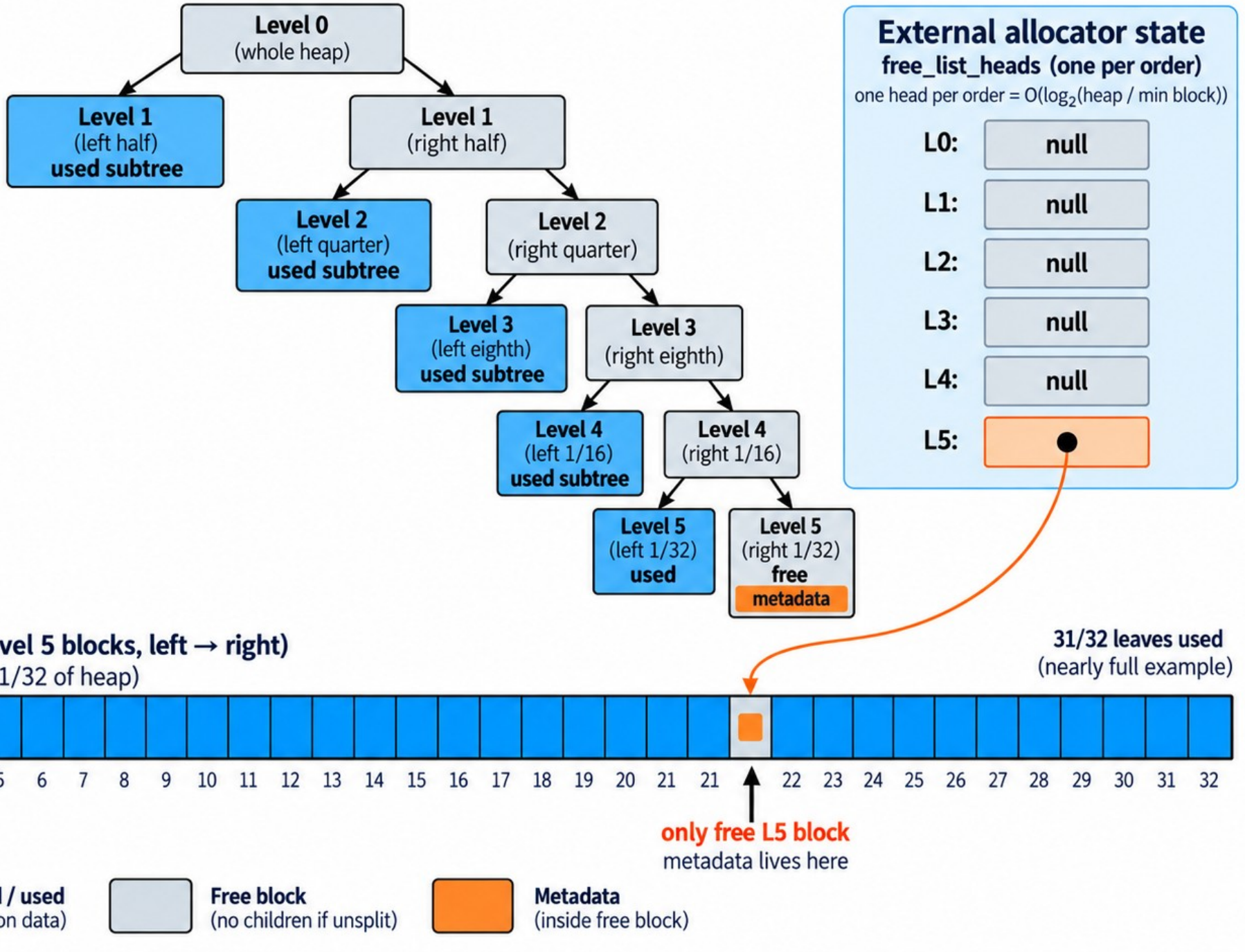
Buddy Heap Allocator (6 Levels)

Recursive division into buddy halves, with metadata in unused free halves (pruned)



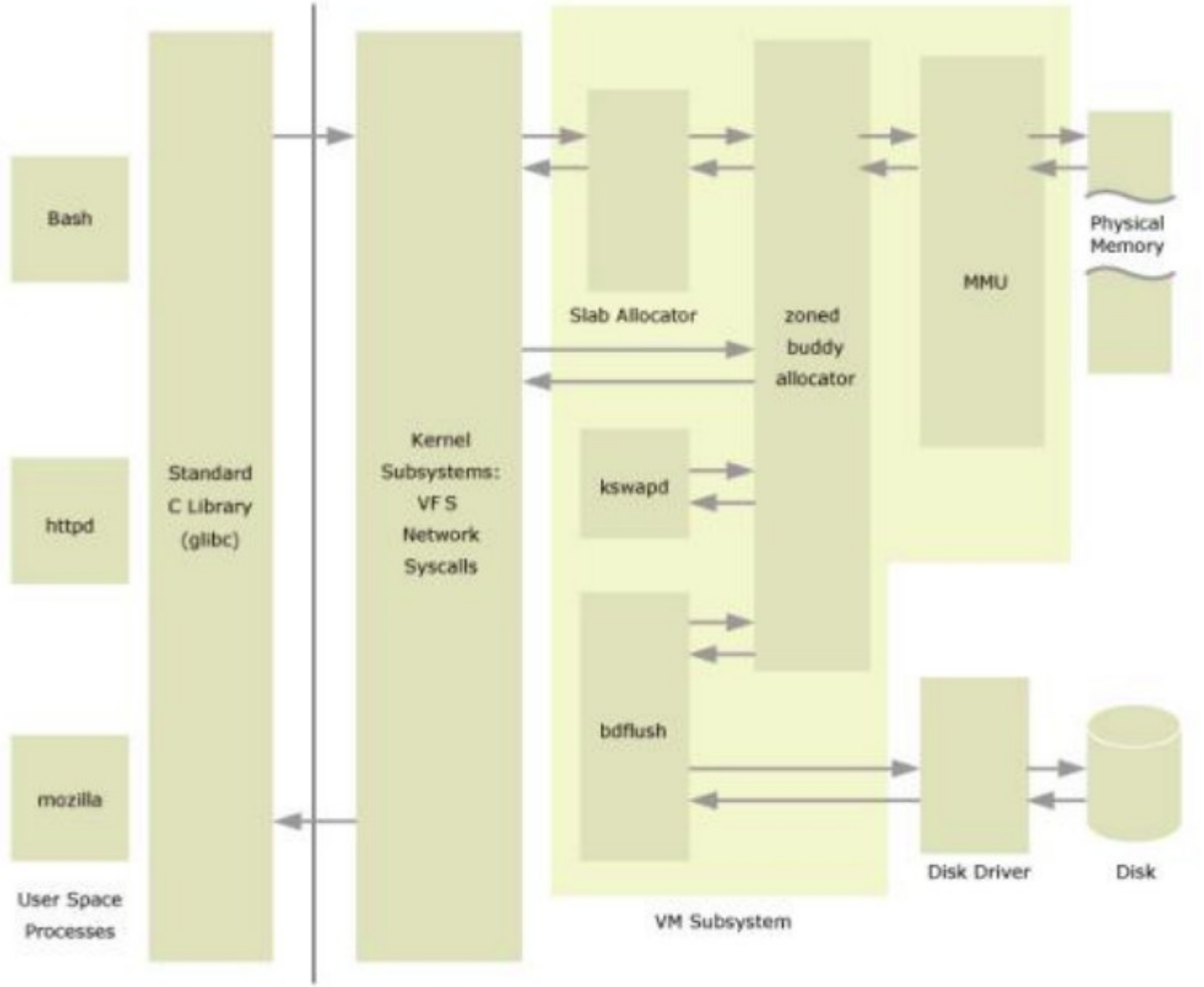
Buddy Heap Allocator (6 Levels)

Nearly full example: 31 of 32 Level 5 blocks allocated (one free)



Slides from…

https://students.mimuw.edu.pl/ZSO/Wyklady/06_memory2/BuddySlabAllocator.pdf



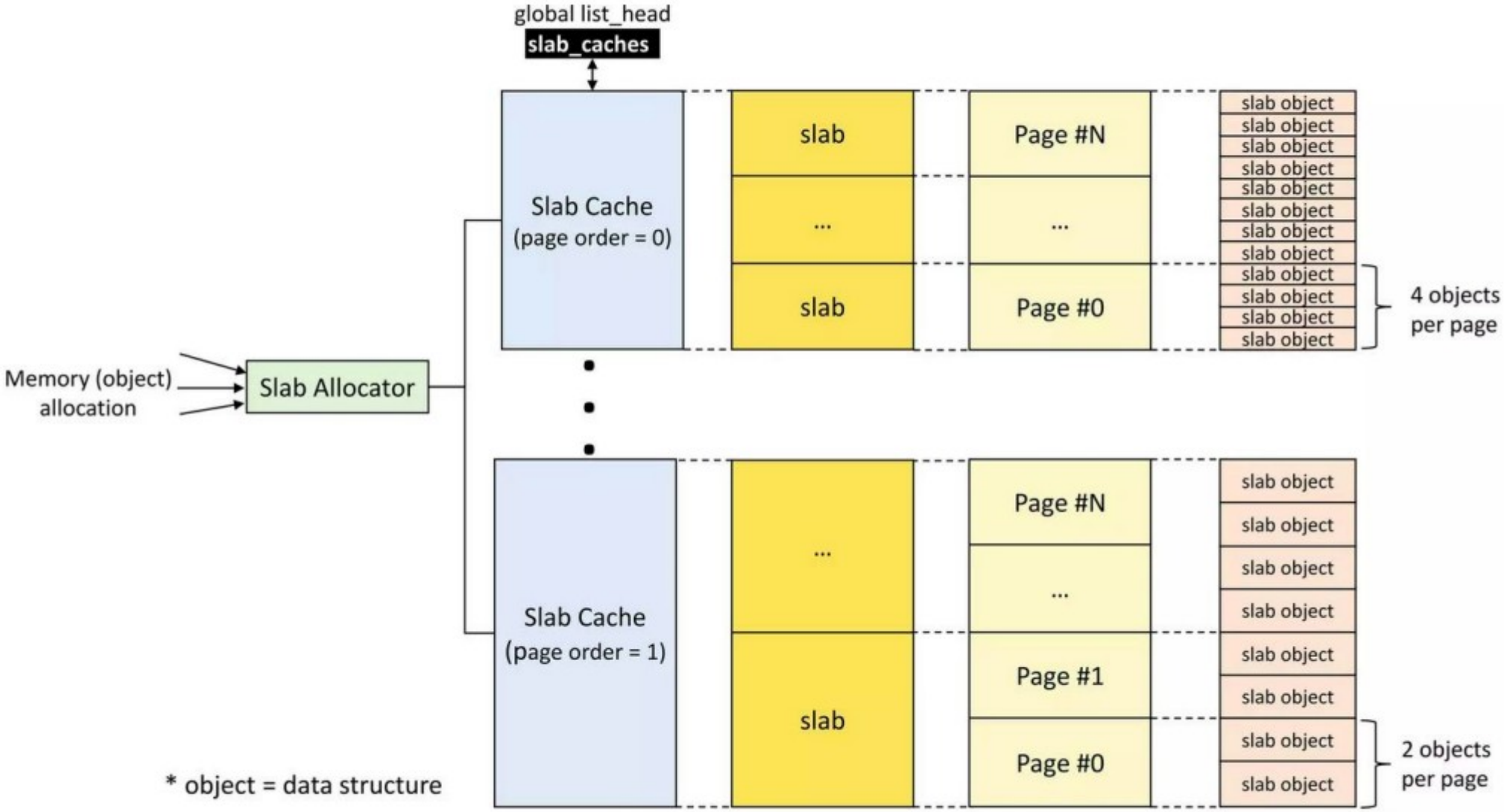
High overview of Virtual Memory subsystem
(source: N. Murray, N. Horman, Understanding Virtual Memory)

Buddy heap *vs.* slab allocator

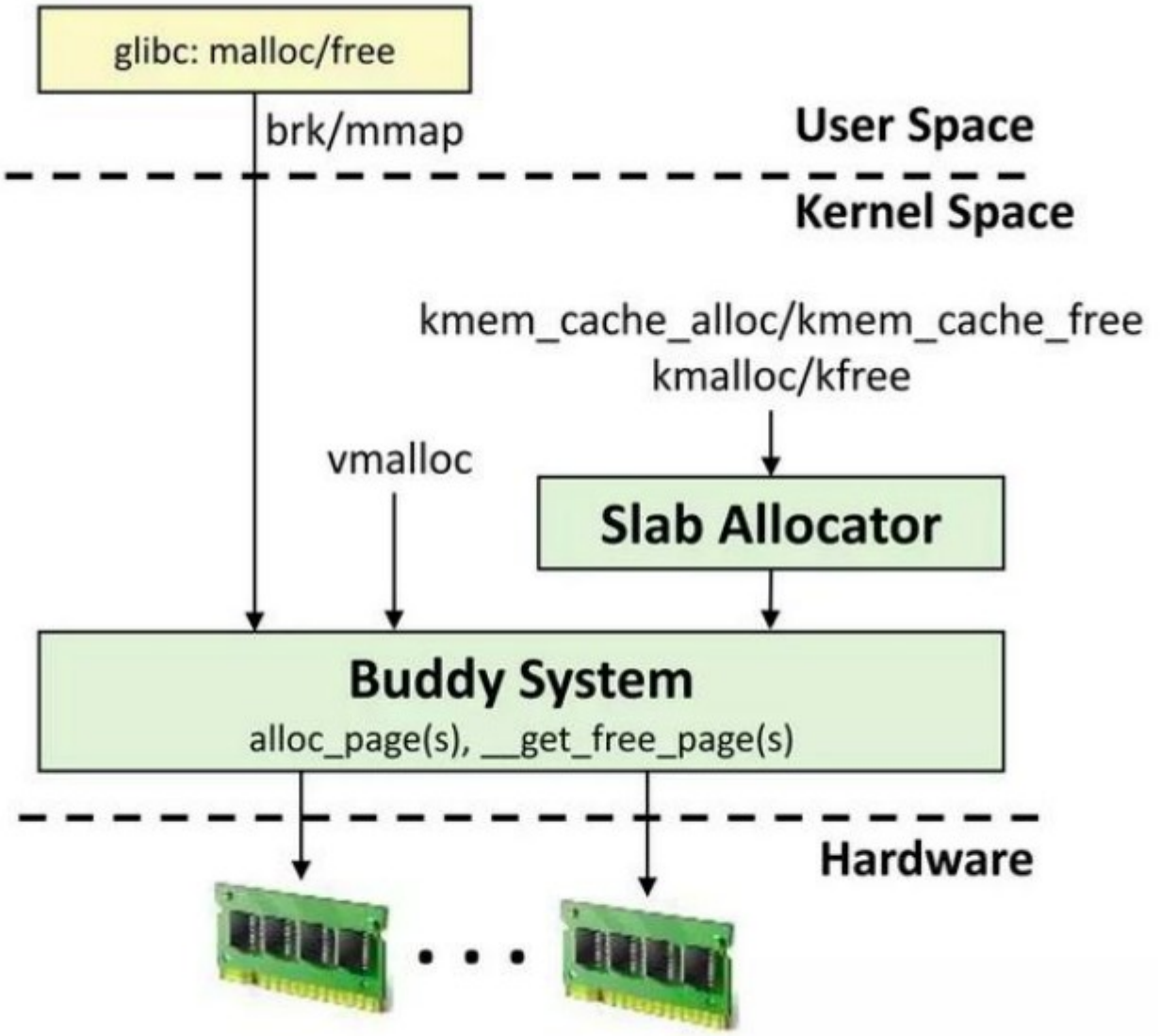
- Buddy heap
  - For grabbing 4KB pages at a time, contiguous
- Slab allocator
  - For grabbing 4KB pages at a time for common data structures

```
extern struct kmem_cache    *vm_area_cachep;
extern struct kmem_cache    *mm_cachep;
extern struct kmem_cache    *files_cachep;
extern struct kmem_cache    *fs_cachep;
extern struct kmem_cache    *sighand_cachep;
```

Slab allocator components



(source: Adrian Huang, Slab Allocator in Linux Kernel, 2022)



(source: Adrian Huang, Slab Allocator in Linux Kernel, 2022)

https://unix.stackexchange.com/questions/385323/is-the-size-of-inodes-fixed

So we *could* indeed make the inodes as large as the blocks, but in a real system this is probably not the case. In case you're wondering, the inode *structure* talks only about the pointers to the data blocks. So we have **156** bytes of pointers to the actual file contents, but the whole inode takes up 256 bytes - basically, we have 100 bytes we can use at our leisure, for whatever metadata we desire.

Treasure and tragedy in *kmem_cache* mining for live forensics investigation

Andrew Case[a], *Lodovico Marziale*[a], *Cris Neckar*[b], *Golden G. Richard, III*[c,*]

[a] *Digital Forensics Solutions, LLC, United States*
[b] *Neohapsis Inc., United States*
[c] *Dept. of Computer Science, University of New Orleans, Lakefront Campus, New Orleans, LA 70148, United States*

```
debian:~/slabwalk# insmod ./slabwalk.ko
debian:~/slabwalk# head -5 /var/log/messages
kernel: [35566.045181] inode: 106310 0 0
kernel: [35566.059469] inode: 106312 0 0
kernel: [35566.071471] inode: 139091 0 0
kernel: [35566.082007] inode: 106308 0 0
debian:~/slabwalk# ffind /dev/sda1 106310
/usr/share/zoneinfo/posix/America/Fortaleza/tmp
/cceZLcAc.o
debian:~/slabwalk# ffind /dev/sda1 139091
/var/run/sshd
debian:~/slabwalk# ffind /dev/sda1 106308
/usr/share/zoneinfo/posix/America/Fortaleza/tmp
/cceoInI5.o
```

Fig. 4 – Traversing the ext3 inode cache and using the Sleuthkit to obtain filenames.

been deleted since they were last used. Unfortunately, the normal algorithm for gathering the names of opened files, walking the list of the inode's directory entries, is not usable since the list is cleared on deallocation. Recovery of the names can still be achieved though with the help of the Sleuthkit, however, since the inode number is still accurate in the structure. Fig. 4 shows the results of traversing the free inode cache and then passing the inode number of free objects to *ffind* of the Sleuthkit, to perform a "dead" analysis of the inode in a filesystem image.

When testing this module, the ext3 filesystem was used and its inode cache, *ext3_inode_cachep*, was traversed. This cache actually contains *ext3_inode_info* structures, and these structures embed a regular inode structure. The *inode* structure allows the module to gather an inode's owner, group, mode, and inode number. Investigators can use this information to discover recently opened files, the permissions the file was opened with, and which user opened the file.

4.5. Socket buffers

4.6. Bound sockets

Before network servers can start accepting connections, they must *bind()* to a network port. In order to facilitate fast lookups of open and in-use ports, the Linux kernel tracks each in-use port within a quickly searchable data structure. For the TCP protocol, each port is kept within the *bind_bucket_cachep* of the *tcp_hashinfo* structure. This cache holds *inet_bind_bucket* structures whose member *port* contains the listening network port. This structure also contains a list of *sock* structures that reference the port, but unfortunately this list is cleared on deallocation. Enumeration of inactive entries of this cache will reveal ports that were previously used to accept network connections. In investigations where network capable malware was either present on the system or suspected to be present, this cache may be useful to help prove or disprove its existence. The port numbers can also be used to quickly point an investigator to interesting data within a network capture. Finally, combined with the network caches presented in the rest of this section, an investigator can gather a large amount of information related to past network activity solely from the machine under investigation. This will save investigative time normally spent examining server, IDS, and router logs.

4.7. Netfilter NAT table

The last cache explored, *nf_conntrack_cachep*, stores the Netfilter connection tracking information in *nf_conn* structures. Netfilter (netfilter.org) is the underlying framework for packet filtering in the Linux kernel, and the popular firewall technology, IPTables, is built upon it. In order to provide NAT capabilities, the connection tracking module must store the source and destination IP address and port for each translated connection. In the current Linux implementation, the incoming and outgoing translation for each connection is stored within the *tuplehash* member of each *nf_conn* structure.

Think about this if you read the BK16 report…

Suppose a process (Pegasus malware, VPN, Tor, etc.) wants to keep network activity from being logged to the hard drive. How hard is it to guarantee that?

Now let’s think about concurrency (as we did when we looked at Rowhammer and MELTDOWN)...

TOCTTOU

Time of Check to Time of Use

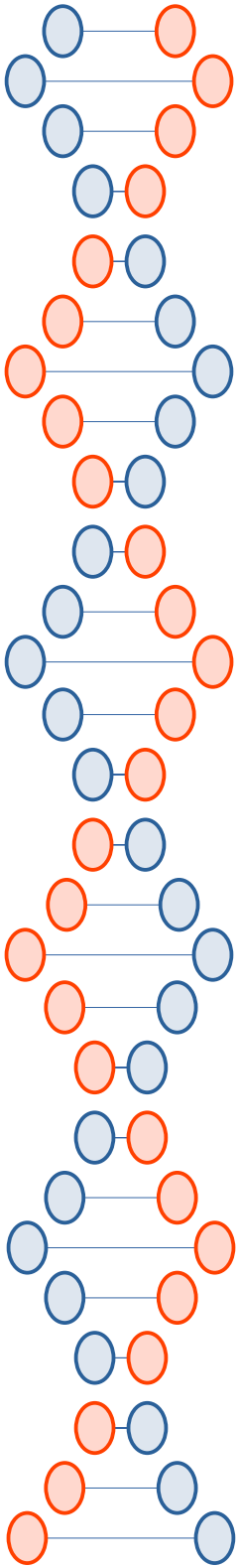
Race conditions

- Often called Time-of-Check-to-Time-of-Use (TOCTTOU)

```
if (!access(“/home/jedi/s”, W_OK))
{
  F = open(“/home/jedi/s”, O_WRITE);
  … /* Write to the file */
}
else
{
  perror(“You don't have permission to write to that file!”)
}
```

Werewolves race condition

touch moderatoronlylogfile.txt
chmod og-rw moderatoronlylogfile.txt



Filesystem TOCTTOU is a type of race condition

https://jedcrandall.github.io/courses/cse536spring2026/borisov.pdf

https://www.usenix.org/legacy/publications/compsystems/1996/spr_bishop.pdf

Fundamental issue is that something changed between when you checked the file and when you used it.

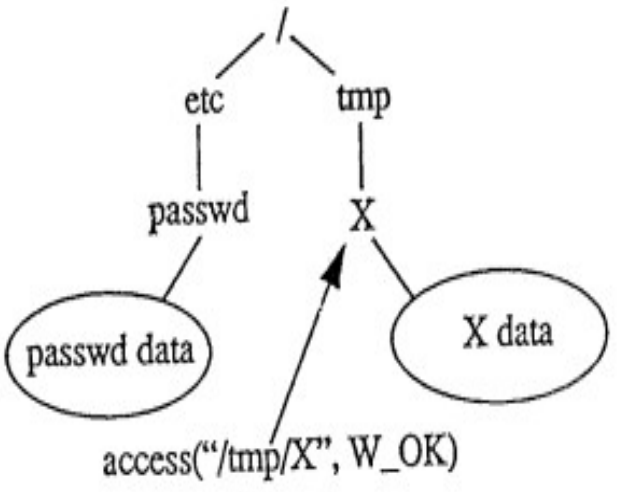


Figure 1a.

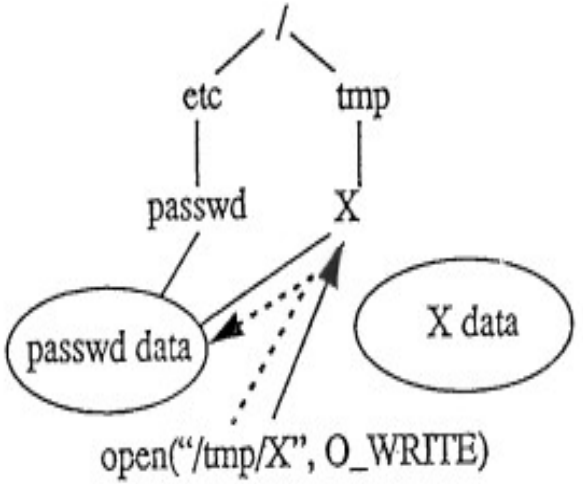
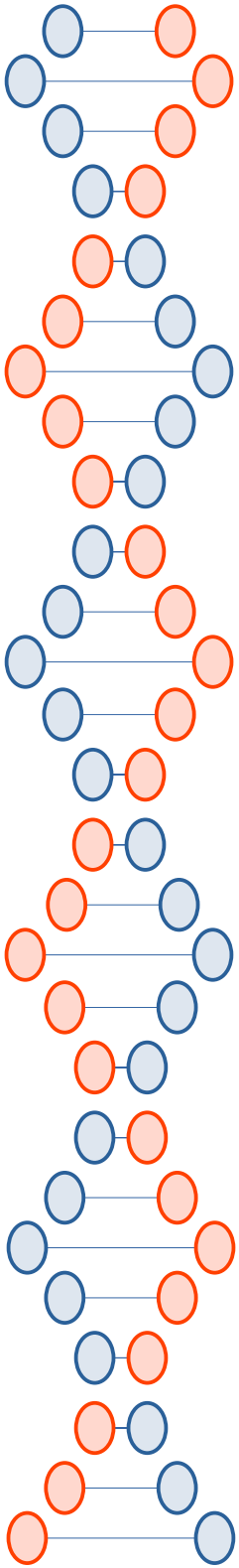
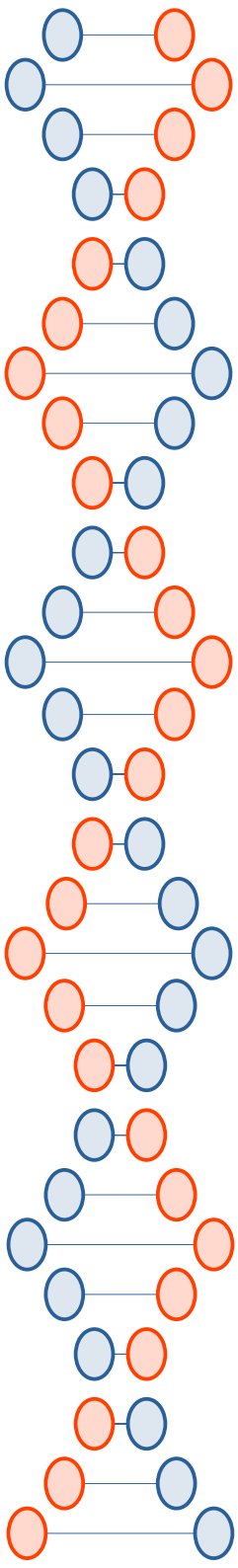


Figure 1b.

Figure 1. Example of the TOCTTOU binding flaw.



Simply dropping privileges to open the file used to be problematic...

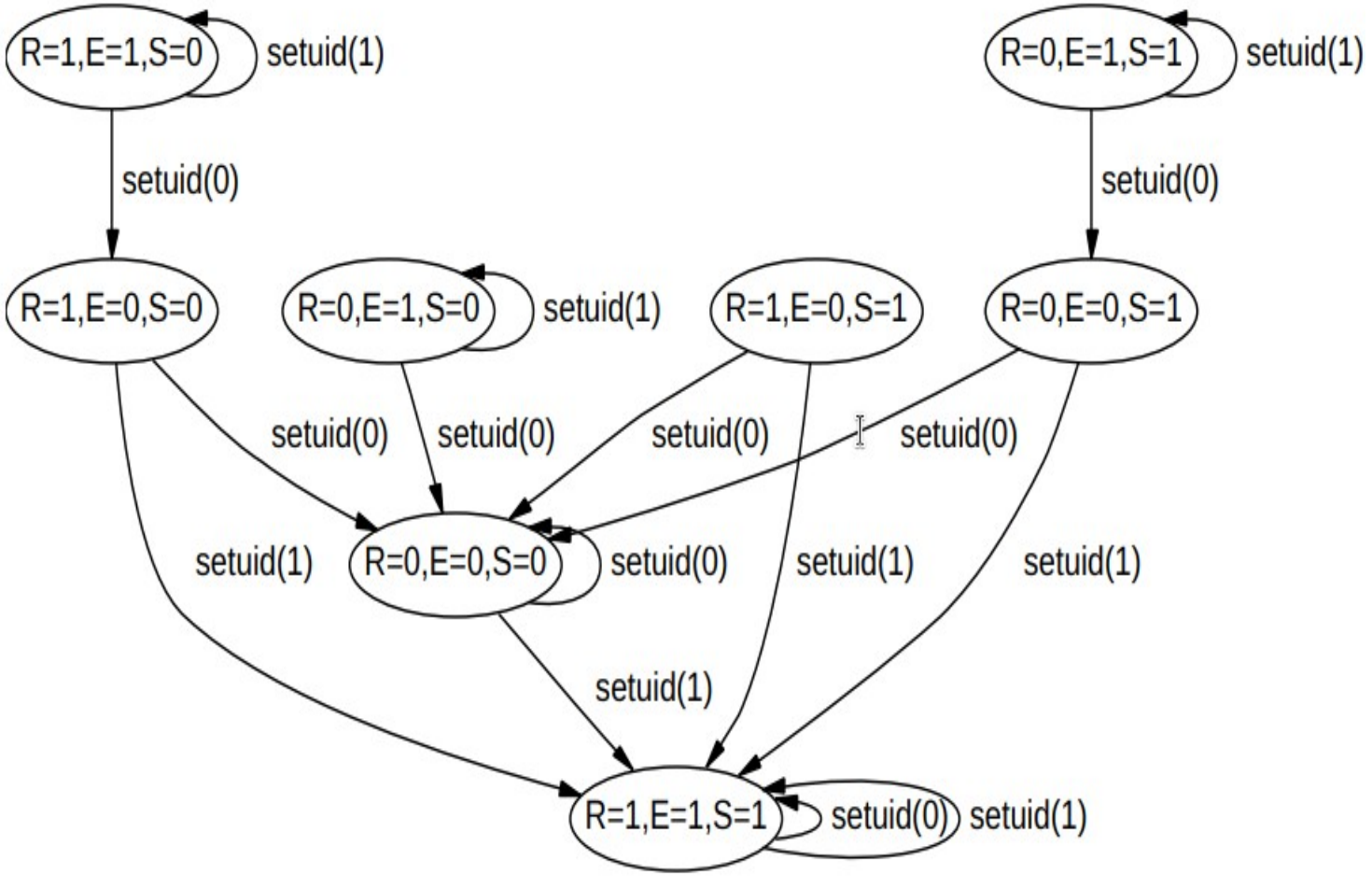


Setuid Demystified*

Hao Chen David Wagner
University of California at Berkeley
{hchen,daw}@cs.berkeley.edu

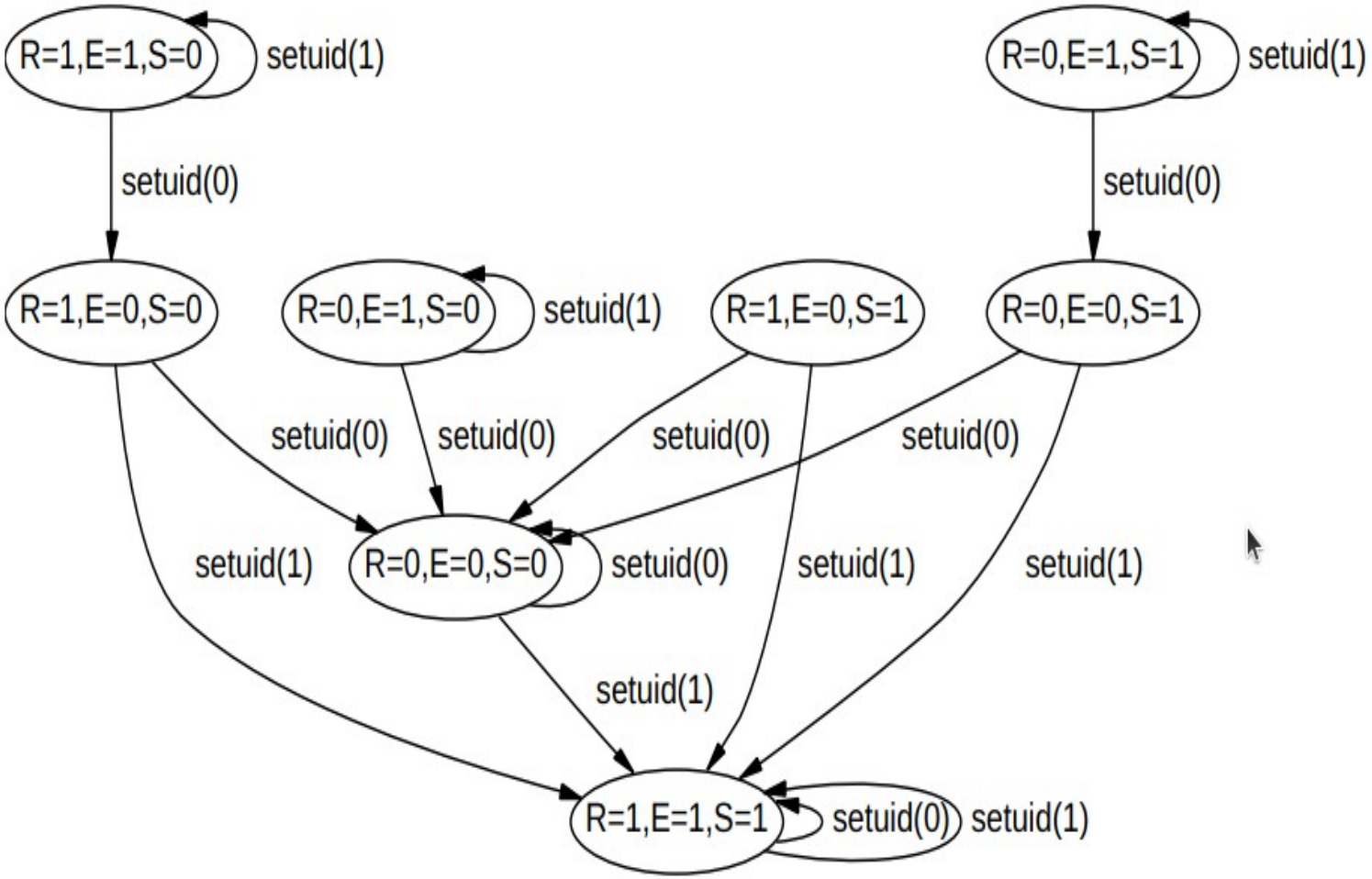
Drew Dean
SRI International
ddean@csl.sri.com

https://www.usenix.org/legacy/events/sec02/full_papers/chen/chen.pdf



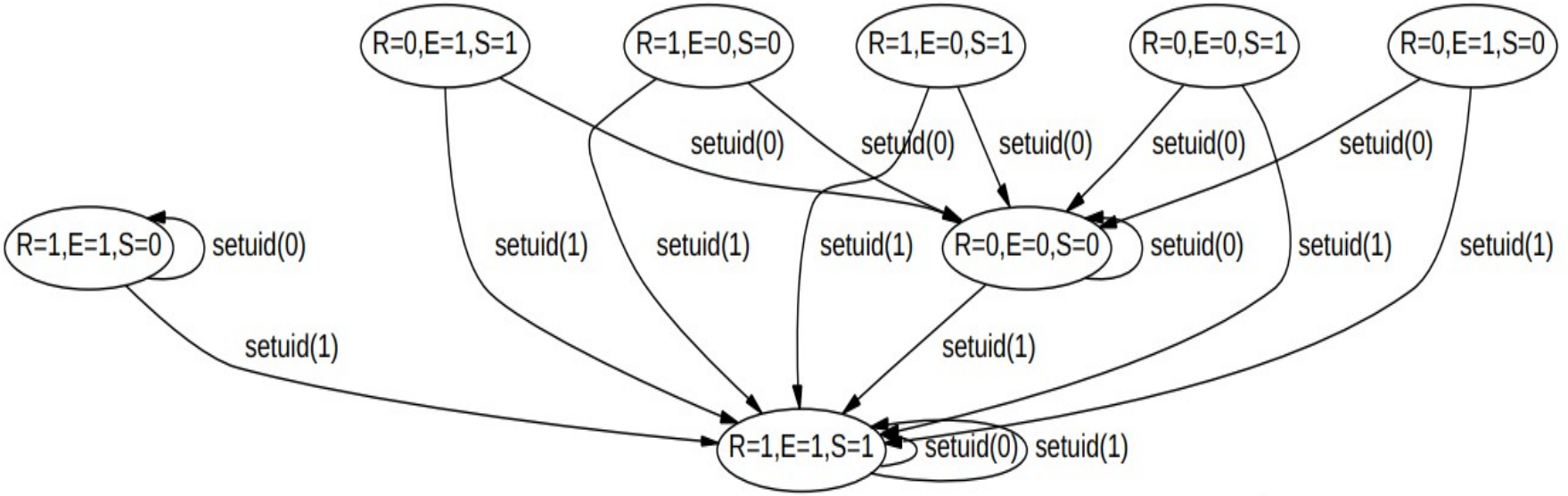
(a) An FSA describing *setuid* in Linux 2.4.18

https://www.usenix.org/legacy/events/sec02/full_papers/chen/chen.pdf

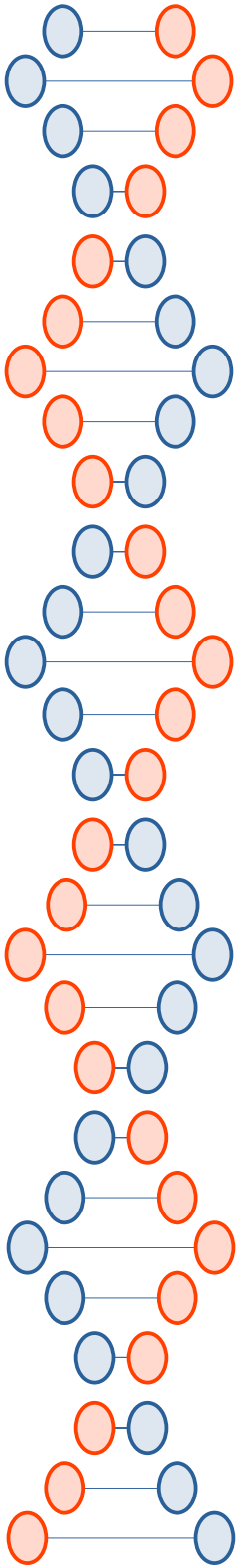


(b) An FSA describing *setuid* in Solaris 8

https://www.usenix.org/legacy/events/sec02/full_papers/chen/chen.pdf



(c) An FSA describing *setuid* in FreeBSD 4.4



Old portable solution

- Parent had to fork a child
  - Child drops privileges and opens the file
    - Sends file descriptor back to parent through a UNIX domain socket

Now you can just use `seteuid()`, but you have to be careful of other concerns…

AI to the rescue! (?)

show me an example of solving filesystem tocttou in unix with openat for euid vs ruid

c

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>

void safe_append(const char *filename, const char *data) {
    // 1. Get real UID (user) and effective UID (root)
    uid_t ruid = getuid();
    uid_t euid = geteuid();

    // 2. Drop privileges to RUID for the check
    if (seteuid(ruid) == -1) {
        perror("seteuid");
        return;
    }

    // 3. Open the file safely
    // O_NOFOLLOW: Do not follow symlinks
    // O_WRONLY | O_APPEND: Append mode
    // O_CREAT | O_EXCL: Fails if file is a link
    int fd = open(filename, O_WRONLY | O_APPEND | O_NOFOLLOW);

    // 4. Restore privileges to root
    if (seteuid(euid) == -1) {
        perror("seteuid");
        close(fd);
        return;
    }

    if (fd == -1) {
        perror("open failed (possible symlink attack)");
        return;
    }

    // 5. Use the file descriptor (not the filename)
    write(fd, data, strlen(data));
    close(fd);
    printf("Successfully appended safely.\n");
}
```

Portably Solving File TOCTTOU Races with Hardness ...

For example, many sites periodically delete files residing under the /tmp directory. If a file was...

USENIX

TOCTTOU Vulnerabilities in UNIX-Style File Systems - USENIX

TOCTTOU (Time Of Check To Time Of Use) is a well known security problem [1] in file systems...

USENIX

Modeling and Preventing TOCTTOU Vulnerabilities in Unix ...

The sendmail example shows the structural complexity of a TOCTTOU attack, which...

Knight Foundation School of Com...

Show all

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <string.h>

void safe_append(const char *filename, const char *data) {
    // 1. Get real UID (user) and effective UID (root)
    uid_t ruid = getuid();
    uid_t euid = geteuid();

    // 2. Drop privileges to RUID for the check
    if (seteuid(ruid) == -1) {
        perror("seteuid");
        return;
    }
```

```c
// 3. Open the file safely
// O_NOFOLLOW: Do not follow symlinks
// O_WRONLY | O_APPEND: Append mode
// O_CREAT | O_EXCL: Fails if file is a link
int fd = open(filename, O_WRONLY | O_APPEND | O_NOFOLLOW);

// 4. Restore privileges to root
if (seteuid(euid) == -1) {
    perror("seteuid");
    close(fd);
    return;
}

if (fd == -1) {
    perror("open failed (possible symlink attack)");
    return;
}
```

```c
    // 5. Use the file descriptor (not the filename)
    write(fd, data, strlen(data));
    close(fd);
    printf("Successfully appended safely.\n");
}
```

Caveats...

- `O_NOFOLLOW` doesn't prevent symlinks in the path, and doesn't apply to hard links
  - `filename` should not be used with root
  - Creating files and directories is a lot trickier than what AI came up with above
  - Doing things recursively requires `openat()` and working with file descriptors, starting at root (/)

https://michael.orlitzky.com/articles/posix_hardlink_heartache.xhtml

Programs written in C can pass the `O_NOFOLLOW` flag to `open` (or `stat`, or `chmod`, or...) to avoid following symlinks entirely. Those functions are all booby-trapped, however: passing `O_NOFOLLOW` only avoids symlinks in the *terminal* component of a path. If you open `/tmp/foo/bar` with `O_NOFOLLOW` and if `/tmp/foo` is a symlink, it will still be followed.

Acknowledgements

This presentation was assembled with modern scholarship techniques:

ChatGPT helped

Made slides, diagrams, and occasionally sounded more confident than wise.

Google's AI tried

Also contributed, with the energy of a helpful autocomplete.

I confirmed details

Read, checked, squinted, and said the sacred academic word: mostly.

Final verification status:
pedagogically useful, technically plausible, emotionally ready

THANK YOU

Errors remain the traditional responsibility of the nearest human.