Data Hiding in Journaling File Systems

By

Knut Eckstein and Marko Jahnke

Presented At

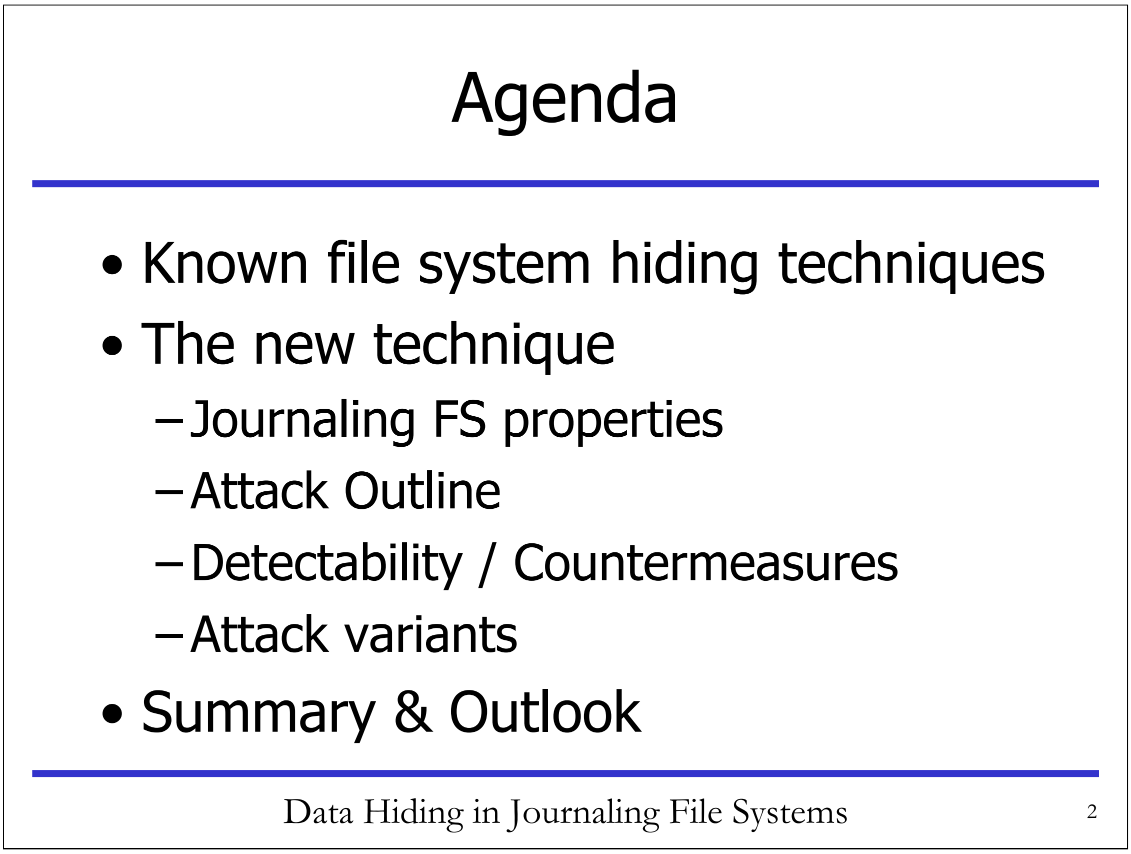
The Digital Forensic Research Conference

DFRWS 2005 USA New Orleans, LA (Aug 17th - 19th)

Data Hiding in Journaling File Systems

Dr. Knut Eckstein
knut at acm dot org

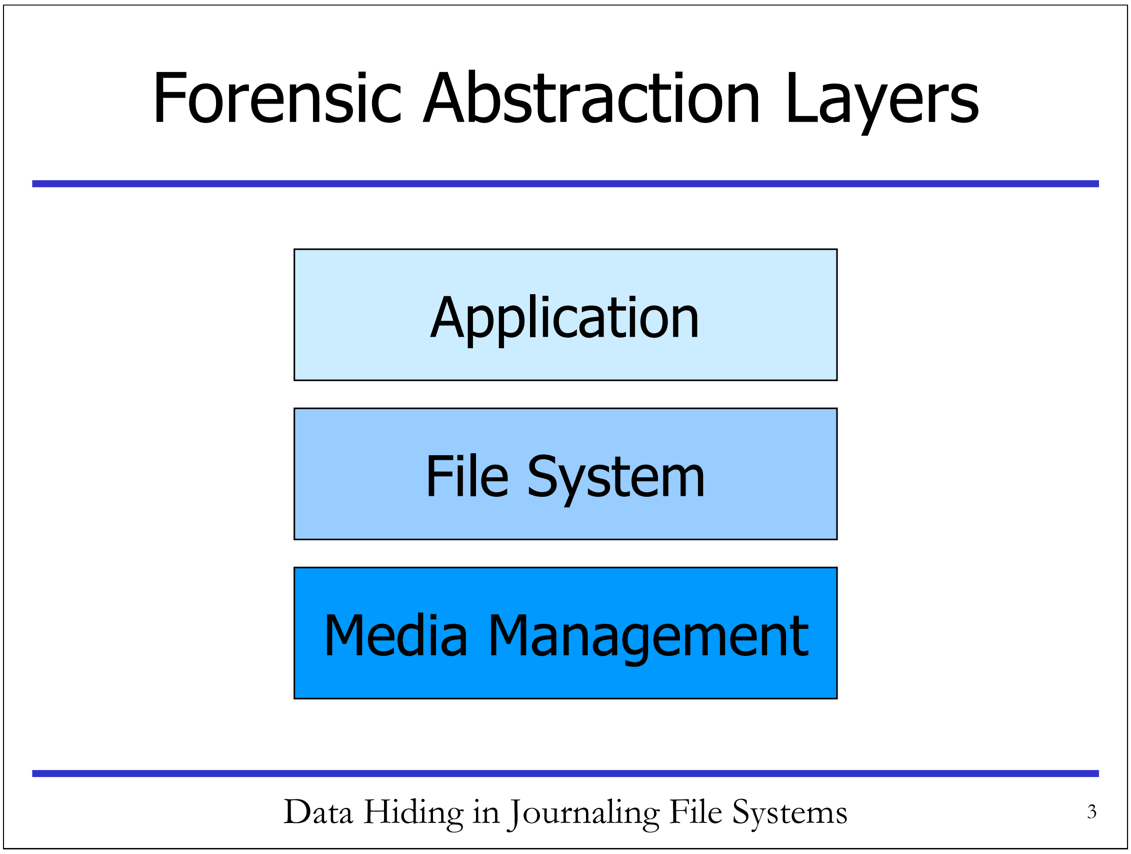
NC3A
P.O. Box 174
The Hague, Netherlands



Agenda

- Known file system hiding techniques
- The new technique
  - Journaling FS properties
  - Attack Outline
  - Detectability / Countermeasures
  - Attack variants
- Summary & Outlook

Known hiding techniques are covered extensively in the paper, this presentation is focusing largely on the new technique.



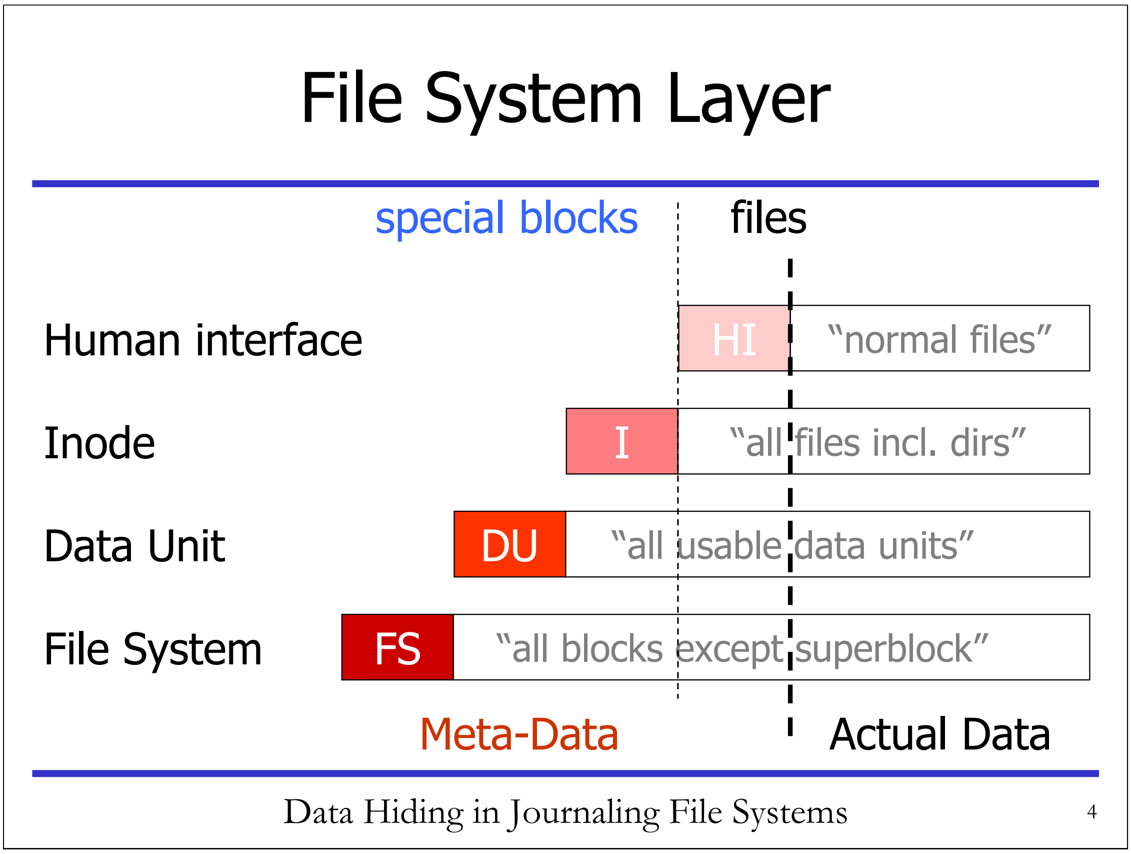
Forensic Abstraction Layers

Application

File System

Media Management

Forensic abstraction layers are also useful for classifying data hiding methods. E.g. steganography or storing information in JPEG header comments falls into the application layer, “playing” with the “host protected area (HPA) of an IDE hard disk falls into the media management layer.



File System Layer

special blocks | files

Human interface — HI — “normal files”

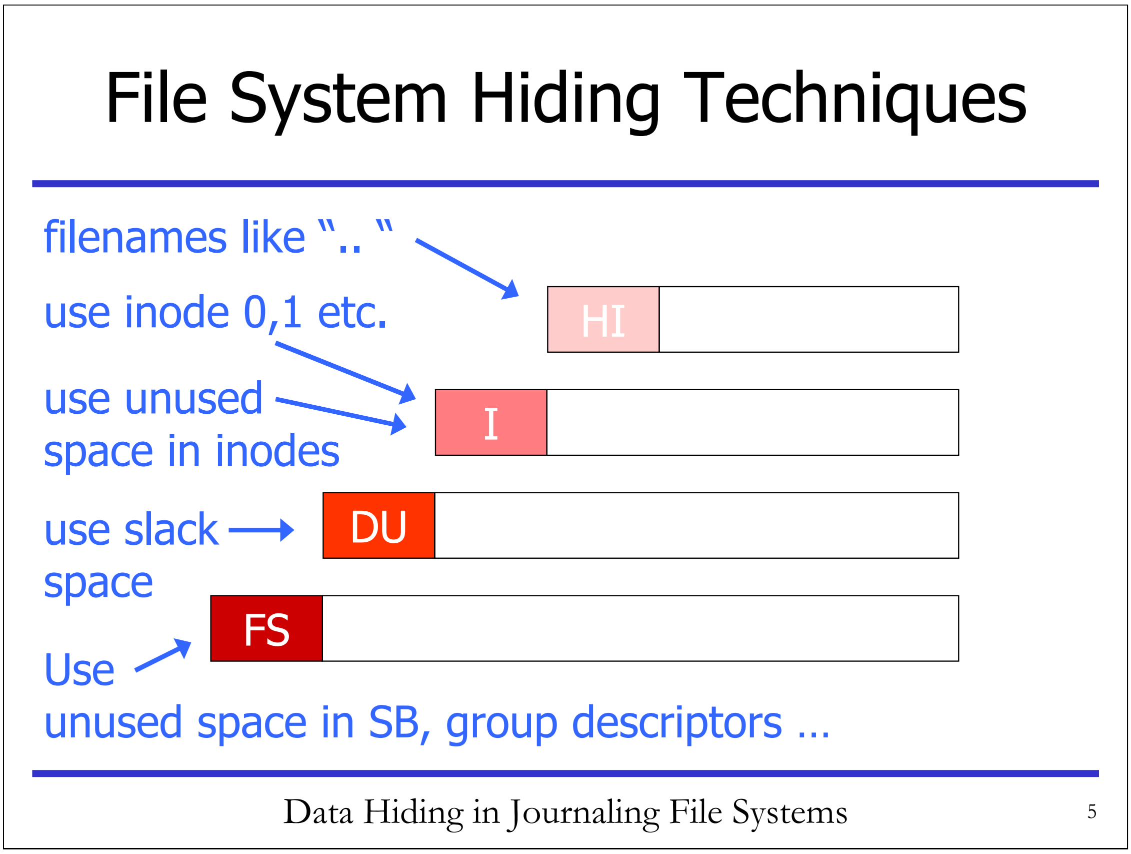
Inode — I — “all files incl. dirs”

Data Unit — DU — “all usable data units”

File System — FS — “all blocks except superblock”

Meta-Data | Actual Data

The filesystem abstraction layer can again be subdivided into multiple sublayers or categories, by looking at the different kind of meta-data used by the file system to store the actual data. These are not necessarily abstraction layers but certainly layers of processing inside the file system. The diagram on this slide (intentionally) looks a little bit like an OSI stack, but depending on the actual file system implementation, file systems are somewhat “messier” in their “layering” in that the multiple encapsulations drawn here are not as perfect in a real file system. But the general idea holds, that on each sub-layer additional meta-data is added/taken away depending on the direction of processing through the stack. The comments in quotes in each layer are specific to the disk layout employed by ext2/3, UFS and similar. One can nicely observe that for those file systems the line between files and “special blocks” on disk to store meta-data is drawn between the human-interface sublayer and the inode sublayer. That means that the only meta data in such a file system which is stored in a file is the HI meta data, in that diretories are nothing but files of a certain type. More recent file system designs like JFS or VxFS move that “demarcation line” substantially to the left, e.g. for JFS the inode list and the block allocation list are also stored in a file leaving – simply put – the super block as the only kind of “other area” on disk, i.e the line between files and “special blocks” is drawn between FS and DU.



File System Hiding Techniques

filenames like “.. “

use inode 0,1 etc.

use unused space in inodes

use slack space

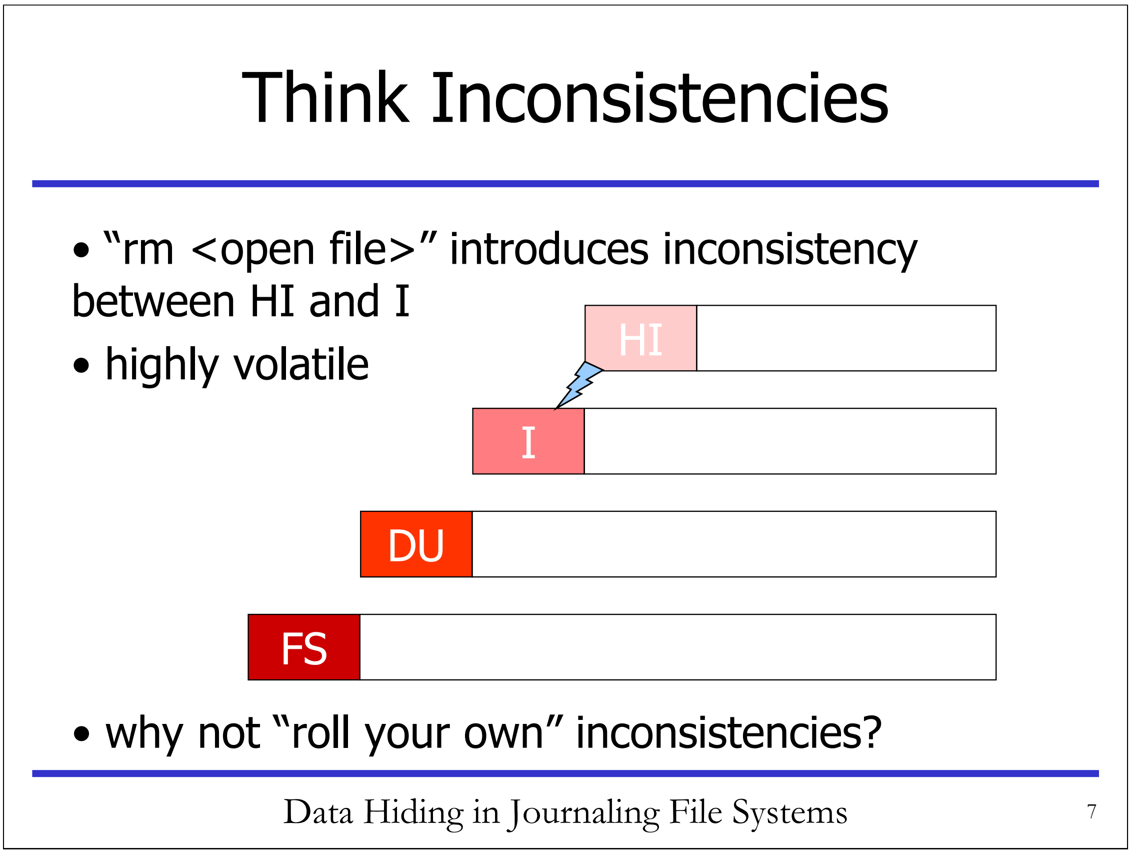
Use unused space in SB, group descriptors …

This slide briefly shows known hiding techniques sorted by file system sublayer they apply to.

Disadvantages

Pick at least one ☺

- Volatility of hidden data
- Easy discovery
- Low amount of hiding space

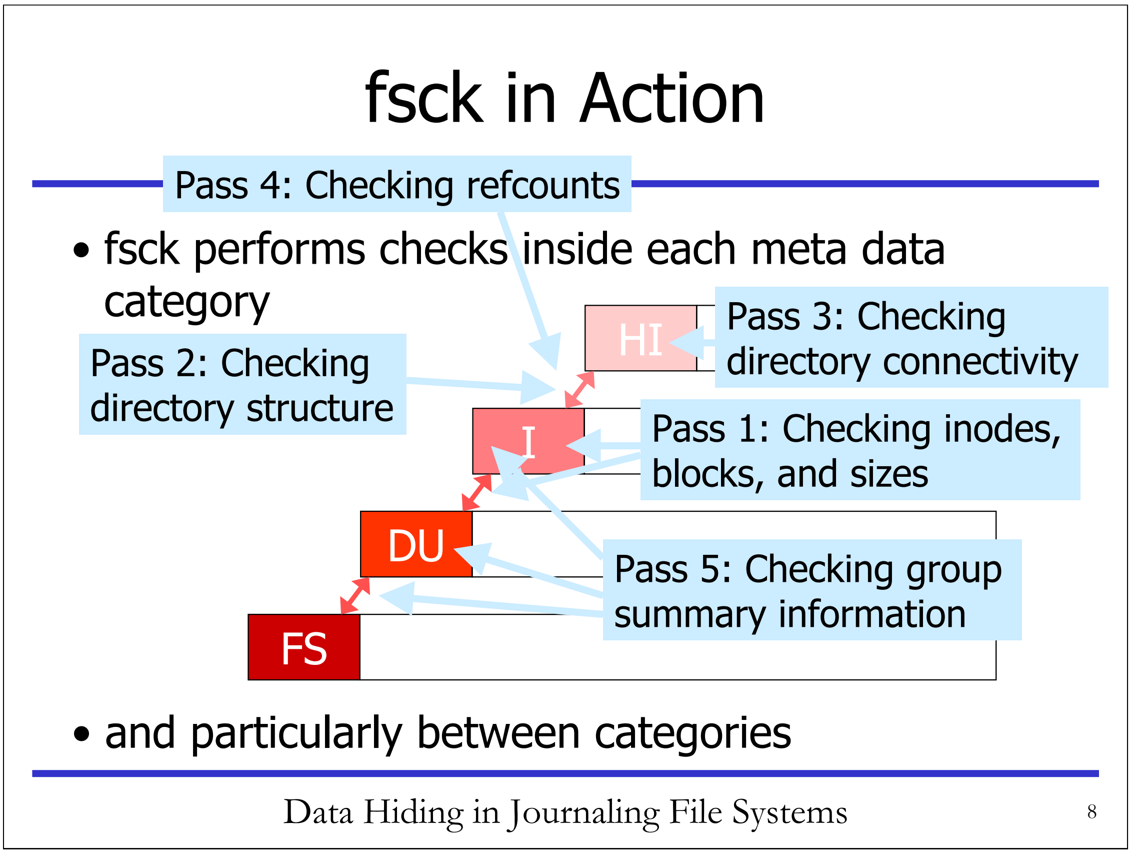


Think Inconsistencies

- “rm <open file>” introduces inconsistency between HI and I
- highly volatile

- why not “roll your own” inconsistencies?

While the volatility of the “rm <open file>” method can be overcome (as discussed in the paper) by a “customized” fsck program, this is a nontrivial step that effectively means trojanizing the host system. But it leads to the idea of an attacker creating custom inconsistencies in the file system to his advantage, exploiting new properties of journaling file systems.



fsck in Action

- fsck performs checks inside each meta data category

Pass 4: Checking refcounts

Pass 3: Checking directory connectivity

Pass 2: Checking directory structure

Pass 1: Checking inodes, blocks, and sizes

Pass 5: Checking group summary information

- and particularly between categories

Data Hiding in Journaling File Systems 8

Inconsistencies in “classical” file systems are going to be detected by the fsck program, either after a system crash or after a certain number of days or mount operations have gone by. This slide illustrates the multi-stage work plan of an fsck program.

This fsck explanation is particular to the “classical” ext2/3/UFS/FFS file systems. Other file systems have their own checkers which may employ completely different phases…

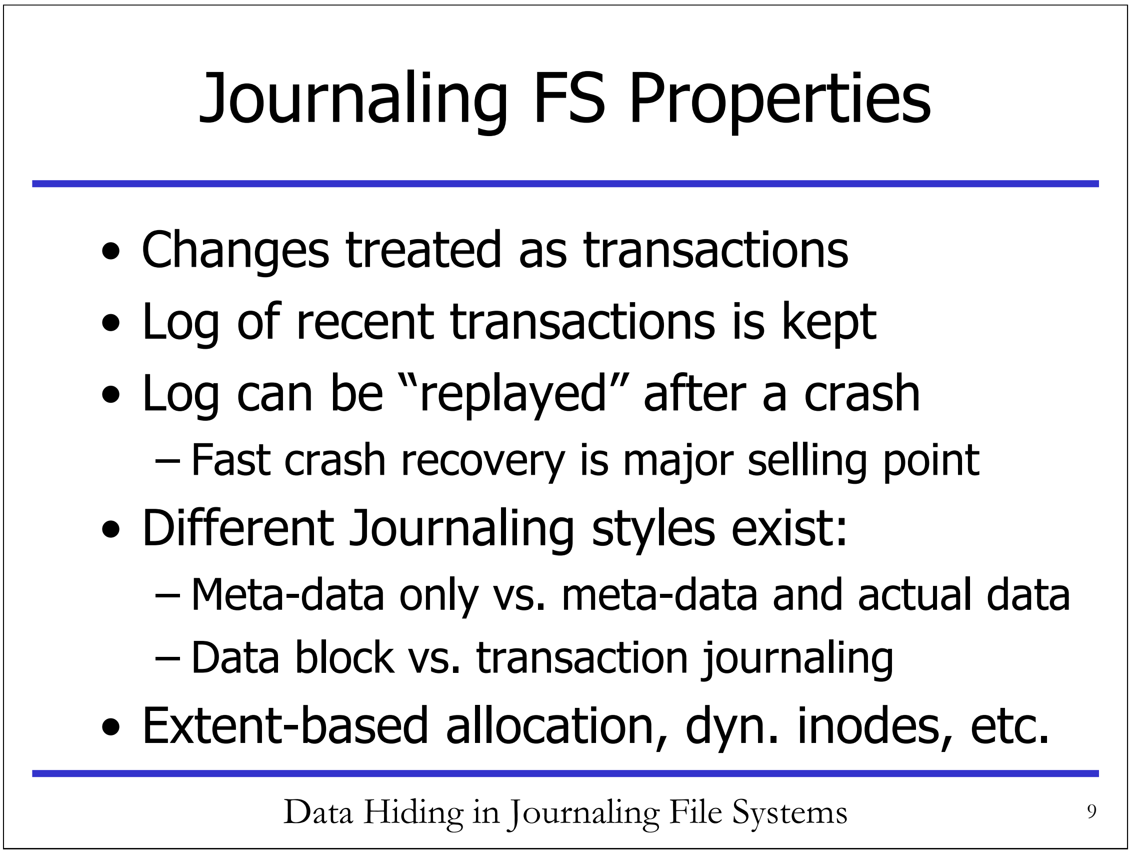
1. Check blocks and inodes
The **fsck** utility checks the inode list for invalid entries. It compares the inode entries to the blocks that the actual files use.

2. Check pathnames
The **fsck** utility removes directory entries discovered from bad inodes and looks for directories that have inode pointers that index out-of-range or point to bad inodes.

3. Check connectivity
The **fsck** utility looks for unreferenced or orphaned directories. Should one be found, it is placed in the **lost+found** directory.

4. Check reference counts
Using information from pass two and three, the **fsck** utility looks for unreferenced files and bad link counts.

5. Check cylinder grouping
The last pass looks at the on-disk free blocks and inode maps comparing them to an updated map from the corrections made during pass one through four. This also explains why inconsistencies in bitmaps are only reported in Pass 5 (as can be seen in the screenshot on slide 12): Earlier passes only check for consistency between certain data and pointer fields themselves, thereby building the fsck-internal allocation bitmaps “on the fly”, which only in Pass 5 are compared to the existing inode and block allocation bitmaps on disk.

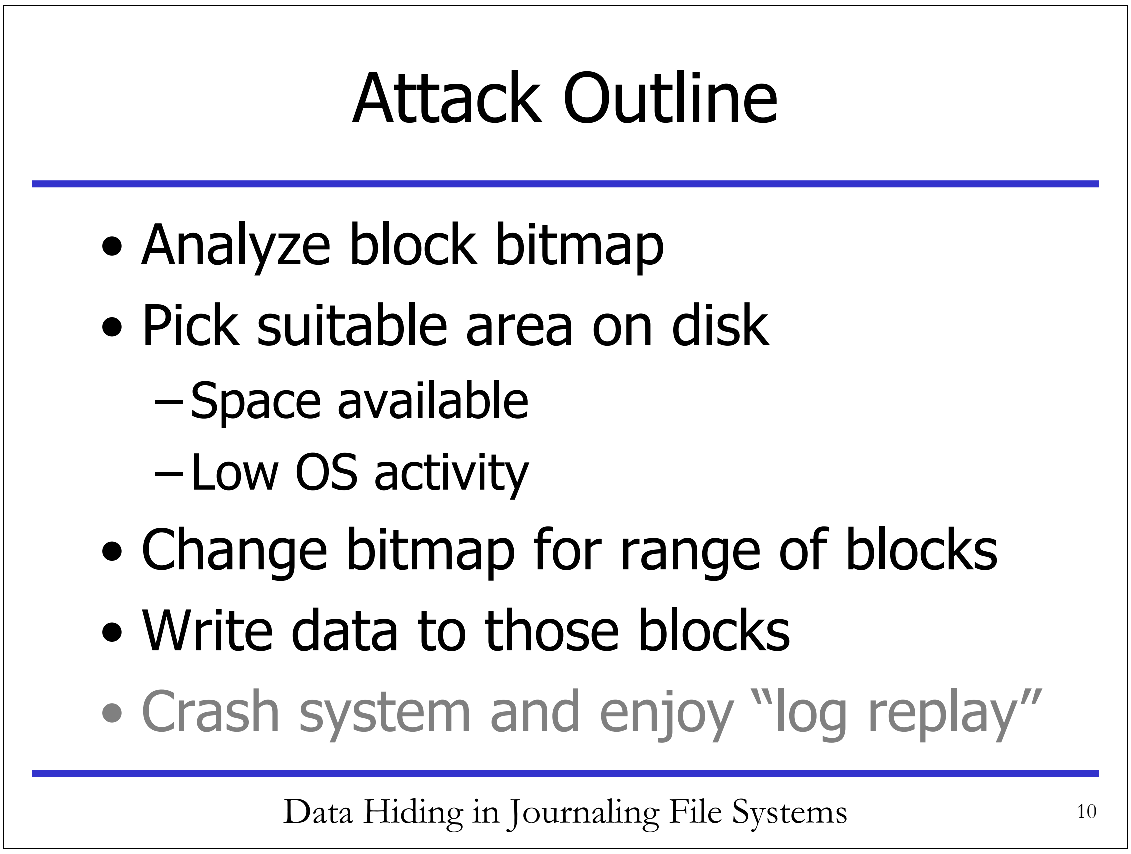


Journaling FS Properties

- Changes treated as transactions
- Log of recent transactions is kept
- Log can be “replayed” after a crash
  - Fast crash recovery is major selling point
- Different Journaling styles exist:
  - Meta-data only vs. meta-data and actual data
  - Data block vs. transaction journaling
- Extent-based allocation, dyn. inodes, etc.

Instead of fsck’ing for several hours on large systems, journaling file systems “by design” can perform crash recovery in seconds, because the log replay time depends on the size of the log but not on the size of the file system (like fsck).

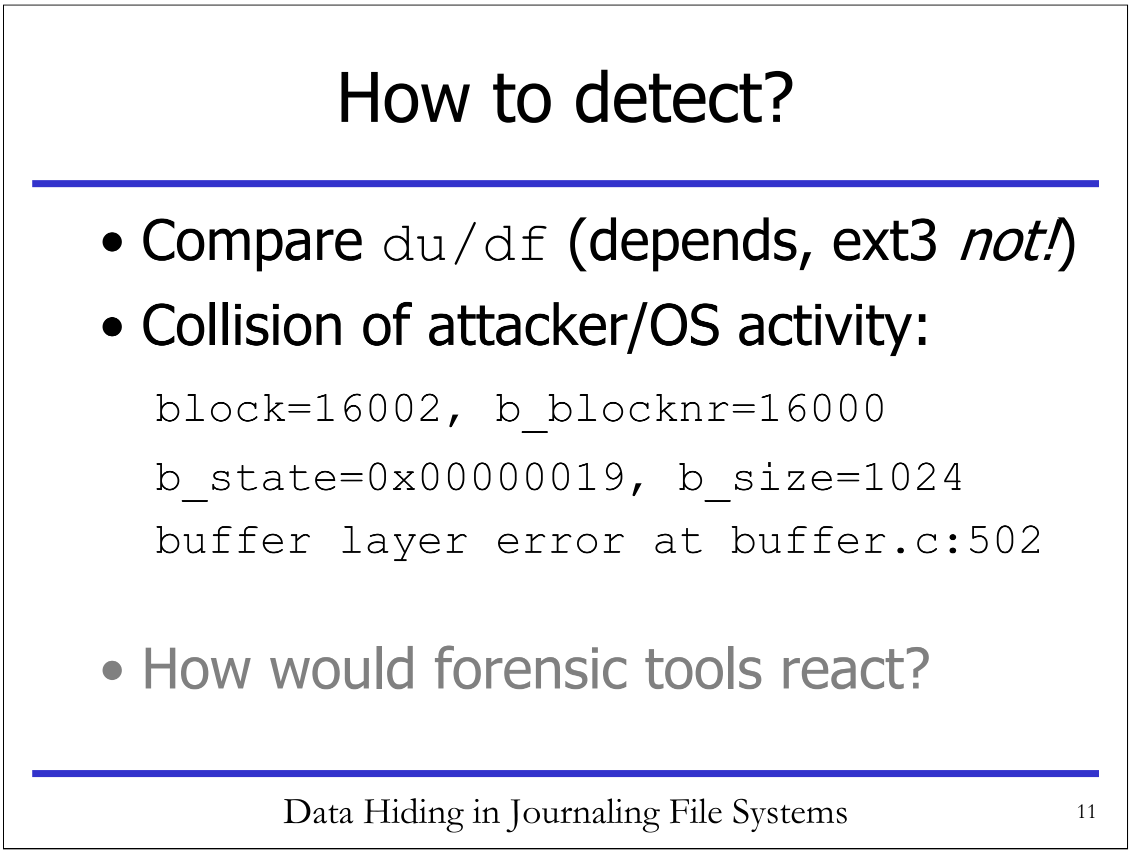
The final bullet point on this page covers other innovations in “modern” file systems which are not linked to their journaling property.



Attack Outline

- Analyze block bitmap
- Pick suitable area on disk
  - Space available
  - Low OS activity
- Change bitmap for range of blocks
- Write data to those blocks
- Crash system and enjoy “log replay”

This is covered in detail (incl screenshots) in the paper.



How to detect?

- Compare `du/df` (depends, ext3 *not!*)
- Collision of attacker/OS activity:

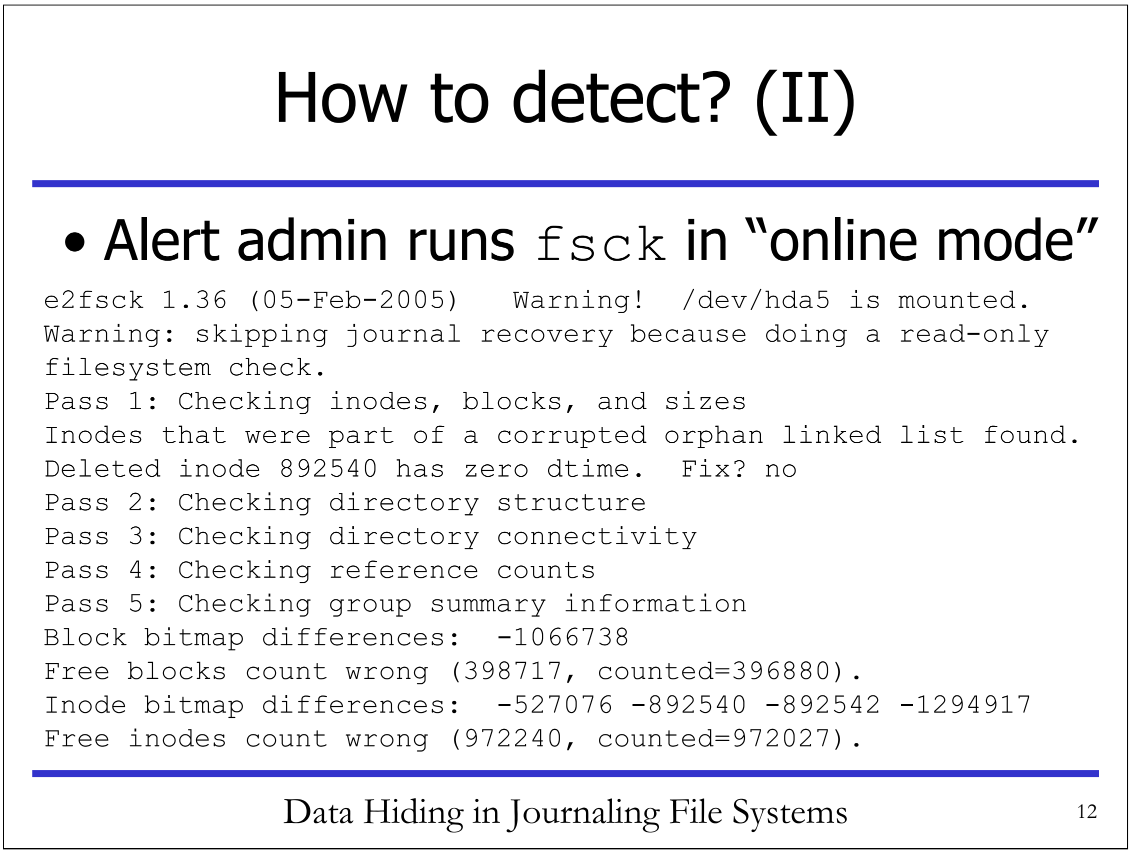
```
block=16002, b_blocknr=16000
b_state=0x00000019, b_size=1024
buffer layer error at buffer.c:502
```

- How would forensic tools react?

Whether comparing du/df output can give a hint to this attack being employed, depends on now the file system generates the summary information used by df. If the free space info is kept current during file allocations and deallocations, but never looks at actual bitmaps on disk, then df will will report what the attacker wants the administrator to believe, like in the case of ext3.

If an attacker chooses a “busy” area on disk, error messages like the one depicted above can show up in the system log when both the attacker and the OS try to modify a bitmap at the same time.

The question how standard forensic tools would react to such an inconsistent file system has not been analyzed yet.

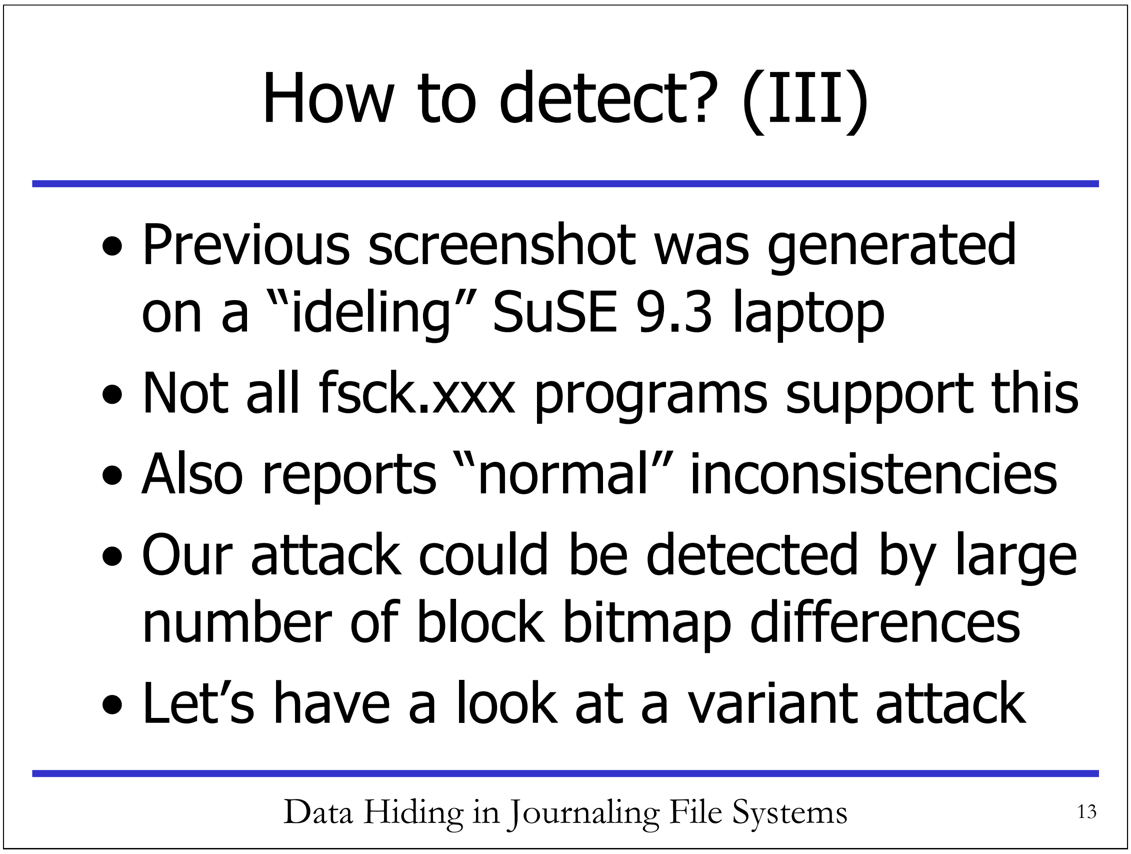


How to detect? (II)

- Alert admin runs `fsck` in "online mode"

```
e2fsck 1.36 (05-Feb-2005)   Warning!  /dev/hda5 is mounted.
Warning: skipping journal recovery because doing a read-only
filesystem check.
Pass 1: Checking inodes, blocks, and sizes
Inodes that were part of a corrupted orphan linked list found.
Deleted inode 892540 has zero dtime.  Fix? no
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
Block bitmap differences:  -1066738
Free blocks count wrong (398717, counted=396880).
Inode bitmap differences:  -527076 -892540 -892542 -1294917
Free inodes count wrong (972240, counted=972027).
```

Comments on this screenshot can be found on the next slide. "Online mode" essentially means that a system administrator who has become alerted/suspicious decides to manually run a "read-only" fsck against a read-write mounted file system.



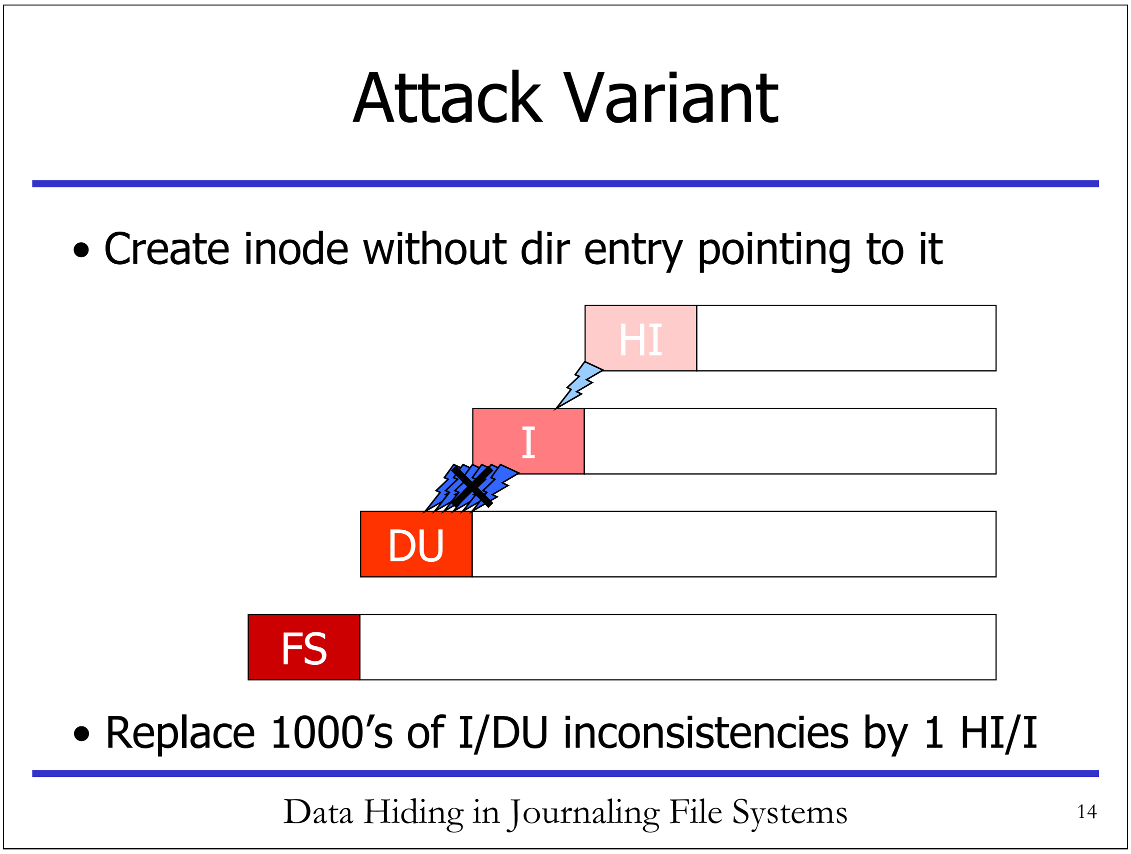
How to detect? (III)

- Previous screenshot was generated on a “ideling” SuSE 9.3 laptop
- Not all fsck.xxx programs support this
- Also reports “normal” inconsistencies
- Our attack could be detected by large number of block bitmap differences
- Let’s have a look at a variant attack

The previous screenshot shows the online-check of a “ideling” laptop’s root file system.

fsck.reiserfs for example does not allow the checking of a rw mounted file system in principle, even when only a read-only check is specified.

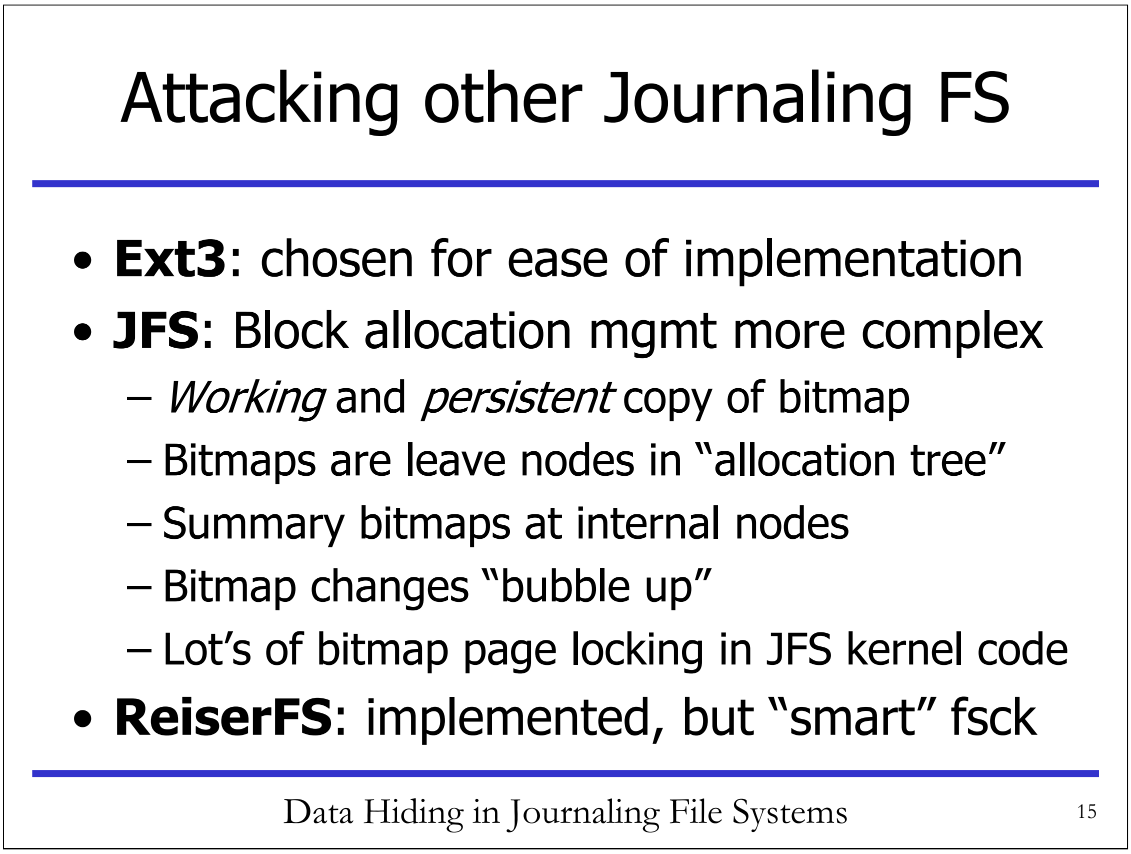
The previous slides shows one block allocation bitmap inconsistency and four inode allocation bitmap differnces resulting from ongoing “routine” file creation, resizing or deletion activity on the “ideling” laptop. Clearly our demo attack which created several thousand block bitmap inconsistencies on purpose would “light up like a Christmas tree” in this onine check



Attack Variant

- Create inode without dir entry pointing to it
- Replace 1000's of I/DU inconsistencies by 1 HI/I

This attack variant again allocates 1000s of disk blocks but now also allocates a single inode and fills its data structures to reference the disk blocks correctly. Thereby 1000s of inconsistencies between the DU and I sublayer are replaced by a single inconsistency bewtween the HI and I sublayer. This single inconsistency is the manually created inode which has no directory entry pointing to it. Unlike the attack discussed so far, this variant is very difficult to detect using an online fsck run, since every online fsck I've run so far reported at least a few HI/I inconsistencies during normal file system operation, so with this attack variant the attacker can hide in the "inconsistency noise" of normal fs operation. The price he has to pay for this advantage is the the higher implementation complexity. A side advantage of this approach is that now only one inode number has to be "memorized" by the attacker in order to store an almost arbitrary amount of information.



Attacking other Journaling FS

- **Ext3**: chosen for ease of implementation
- **JFS**: Block allocation mgmt more complex
  - *Working* and *persistent* copy of bitmap
  - Bitmaps are leave nodes in "allocation tree"
  - Summary bitmaps at internal nodes
  - Bitmap changes "bubble up"
  - Lot's of bitmap page locking in JFS kernel code
- **ReiserFS**: implemented, but "smart" fsck

Ext3 was chosen for the initical proof of concept coding, since it is fully supported by TSK and the attack code is mainly a modified version of TSK code for ext3 which not only reads but also writes block bitmaps as well as blocks themselves.

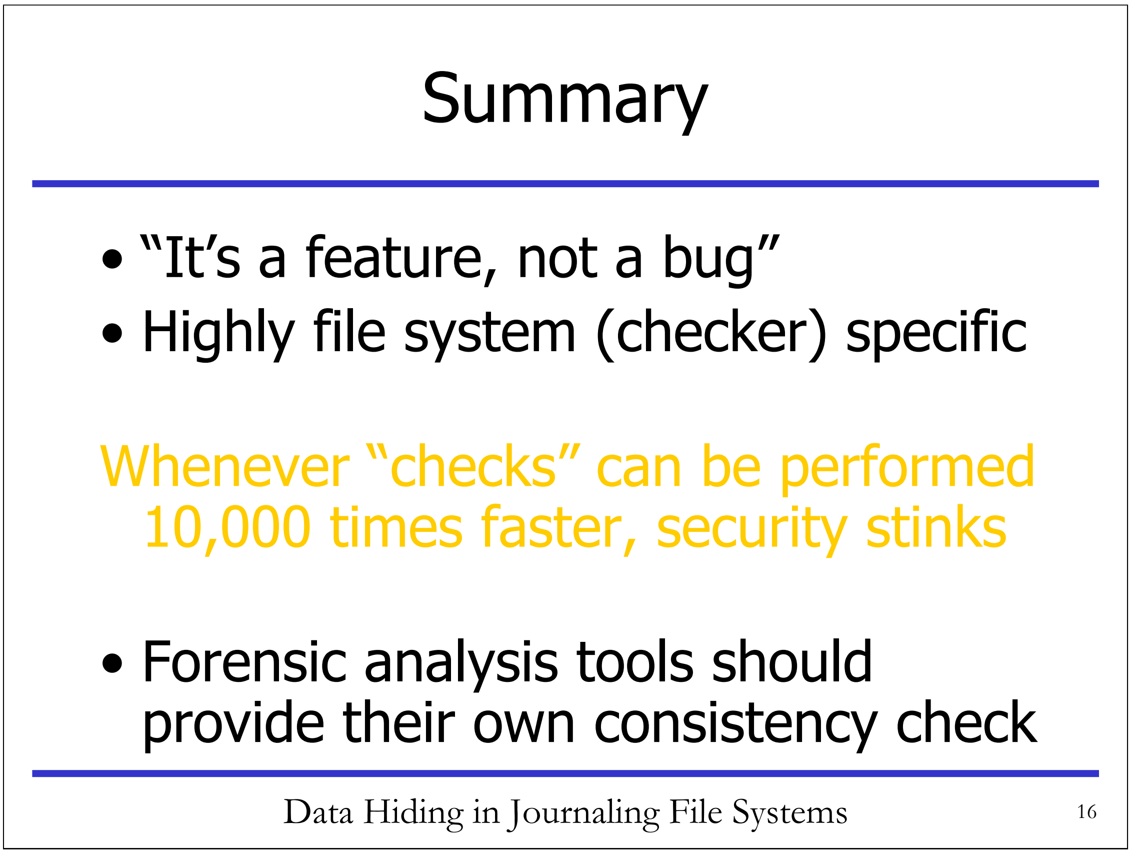
Although I meanwhile have implemented full support for JFS for Linux in TSK, given the complexity of the data structures that hold the block allocation bitmap, the level of effort to create the proof of concept code appears significantly higher than that for other file systems.

Just before the DFRWS 2005 was held I implemented the attack for ReiserFS, so this is not covered in the paper. Injecting the inconsistent bitmaps and hiding the data worked fine in general, but an interesting feature of the fsck.reiserfs program combinded with the boot action sequence of SuSE 9.2 on the test system lead to detection at ***every*** reboot, not just reboots after crashes, provided that the file system is listed in /etc/fstab.

If mounted manually or through another script during startup, detection of the attack fails, because the mount code for ReisesFS will itself replay the log and only the log upon detecting that the file system is not clean.

If the file system on the other hand is listed in /etc/fstab, what happens is that due to the way fsck.reiserfs is called by /etc/init.d/boot.localfs during every boot it not only replays the journal during startup to determine whether the file system is clean, but also affords some time to a few consistency checks e.g. checking the block allocation bitmaps, which leads to the inconsistency being discovered and fixed. The hidden information does not get erased, it just is put in danger of being erased in the future since the blocks it uses are now again marked as free. That means that a bold attacker could also go ahead and change the bitmaps again to suit his purpose, especially since in the system tested (SuSE 9.2) no traces of the inconsistency being detected and fixed appear in the system log files, i.e the administrator has to see them scrolling past on the console or he will never know…

But kudos to the ReiserFS team for taking a few extra security-relevant checks… A very determined attacker would now start analysing the source code of fsck.reiserfs in order to find out where the consistency checks stop, so that he can determine an inconsistency that would not be detected :-}}

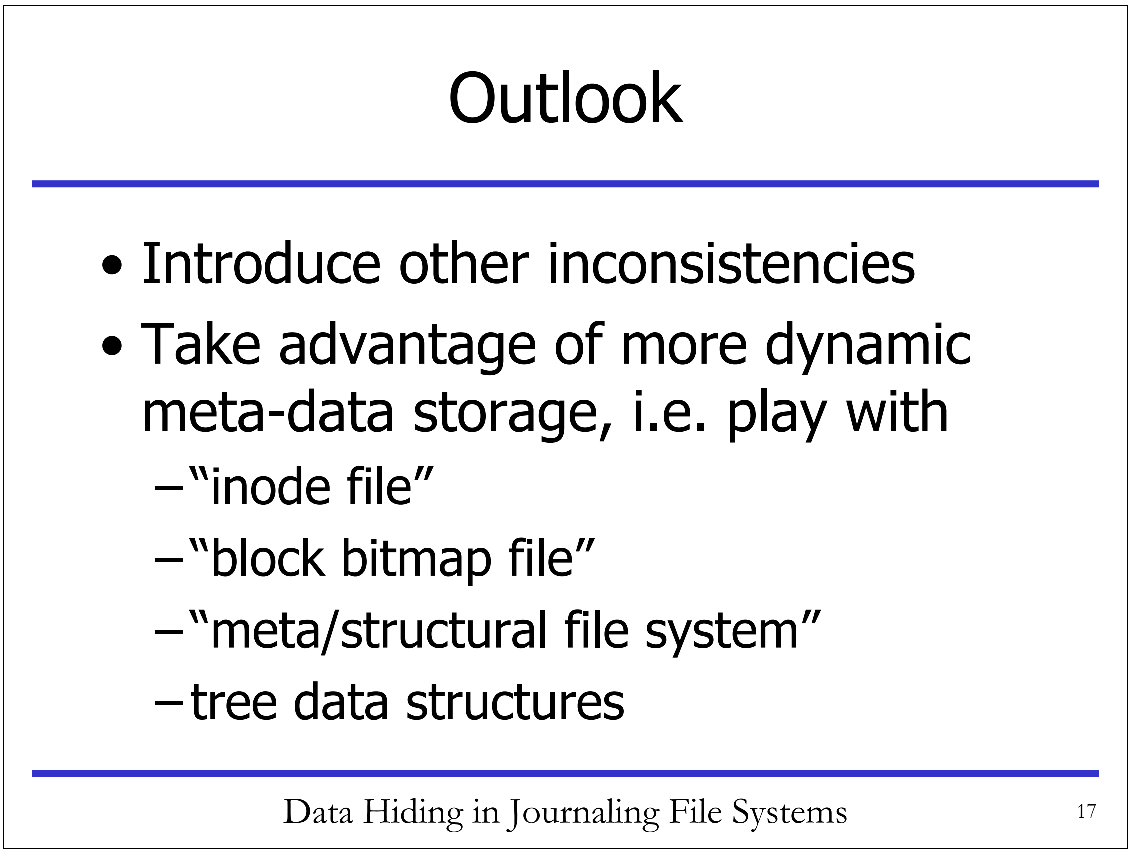


Summary

- "It's a feature, not a bug"
- Highly file system (checker) specific

Whenever "checks" can be performed 10,000 times faster, security stinks

- Forensic analysis tools should provide their own consistency check



Outlook

- Introduce other inconsistencies
- Take advantage of more dynamic meta-data storage, i.e. play with
  - “inode file”
  - “block bitmap file”
  - “meta/structural file system”
  - tree data structures

Questions