Timestomping NTFS

(with emphasis on directory index records)

Wicher Minnaard
supervisor: Marco van Loosen (Fox-IT)

UVA/SNE MSc research project presentation
July 2nd, 2014

Research question (1)

What forms of NTFS timestamp tampering can be detected by inspecting NTFS structures?

Timestamps on NTFS (1)

MACB timestamps:

- **M**odified [1]
- **A**ccessed [2]
- **C**hanged [3]
- **B**irth [4]

[1] mOdified - cOntents
[2] updates turned of by default in recent Windows versions
[3] chAnged - metAdata
[4] but what does that mean, anyway

Timestamps on NTFS (2)

In the Master File Table entries:

- **SI** - STANDARD_INFORMATION attribute. User-modifiable through `SetFileInformationByHandle` and `ZwSetInformationFile` routines.
- **FN** - FILE_NAME attribute. Files can have multiple of these, in different namespaces. They are not exposed to userspace.
- Inside directory indices: timestamps reflecting **SI** timestamps, but embedded inside an **FN** attribute...

Maximum number of timestamps:
4 (MACB) * (1 SI, 3 FN, 3 directory index entries) = 28!

Tampering techniques

How does one tamper with timestamps ("timestomping") ?

- Through APIs, as classic `timestomp.exe`[5] does. Not perfect.
- Direct modification of on-disk NTFS structures. Current cream of the crop: later versions of `SetMace`[6].

[5] James Foster Vinnie Lin, Blackhat 2005

[6] Joakim Schicht, 2011-2014

Research question revisited

What forms of NTFS timestomping can be detected by inspecting NTFS structures?

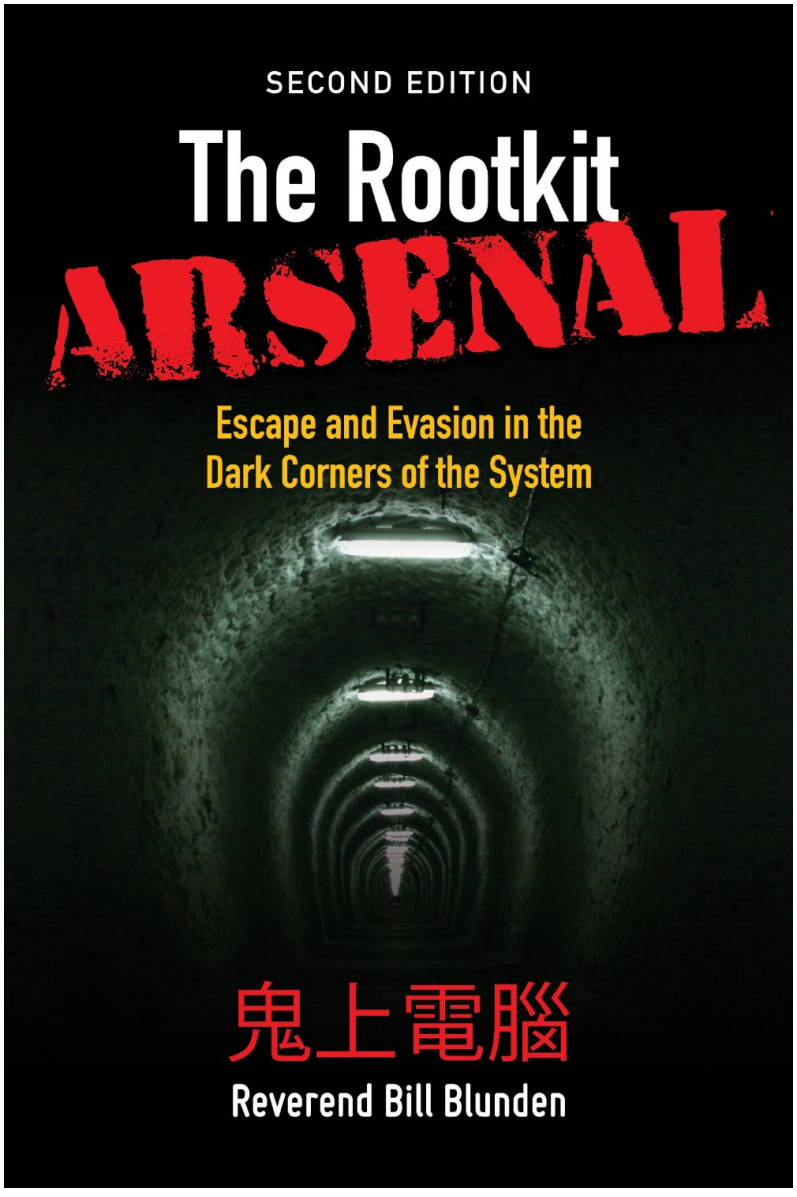
Research question revisited

What forms of NTFS timestomping can be detected by inspecting NTFS structures?

Subquestions:

- What is the form, function, and location of all these timestamps? How do they relate to each other?
- What timestomping techniques are available to modify each timestamp?
- What inconsistencies (if any) do the techniques introduce?

Timestomping detection



Timestomping detection

subtlety is key!

> This highlights the fact that you should *aspire to subtlety*, but when it's not feasible to do so, then you should at least be consistent. If you conceal yourself in such a manner that you escape notice on the initial inspection but are identified as an exception during a second pass, it's going to look bad. The investigator will know that something is up and call for backup. If you can't change time-stamp information uniformly, in a way that makes sense, then don't attempt it at all.

Timestomping detection

subtlety is key!

> This highlights the fact that you should *aspire to subtlety*, but when it's not feasible to do so, then you should at least be consistent. If you conceal yourself in such a manner that you escape notice on the initial inspection but are identified as an exception during a second pass, it's going to look bad. The investigator will know that something is up and call for backup. If you can't change time-stamp information uniformly, in a way that makes sense, then don't attempt it at all.

Common slip-up: Forgetting about the 100ns timestamp resolution: `2014-01-01 12:12:34.000000`

Timestomping detection

subtlety is key!

> This highlights the fact that you should *aspire to subtlety*, but when it's not feasible to do so, then you should at least be consistent. If you conceal yourself in such a manner that you escape notice on the initial inspection but are identified as an exception during a second pass, it's going to look bad. The investigator will know that something is up and call for backup. If you can't change time-stamp information uniformly, in a way that makes sense, then don't attempt it at all.

Generally: look for inconsistencies

Timestomping: Inconsistencies

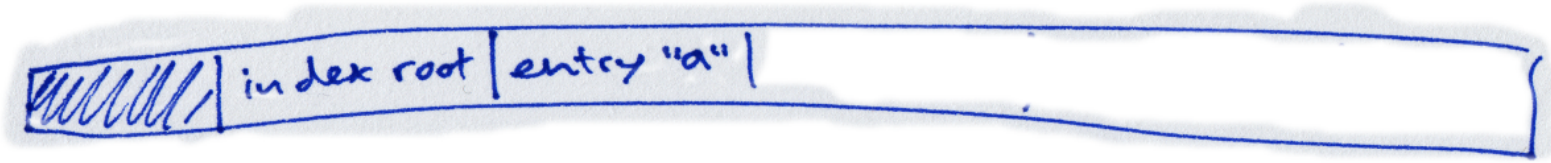
- Causal relationships (*happened-before*): allocators, sequence numbers. Willassen, 2008.
- Deriving past operations from the NTFS journal. Cho, 2012.
- Explicit second source of timestamps: directory index entries in B-tree slack (INDEX_ALLOCATION): `INDXParse.py`, Ballenthin, 2011-2014.

Parsing the INDEX_ROOT attribute

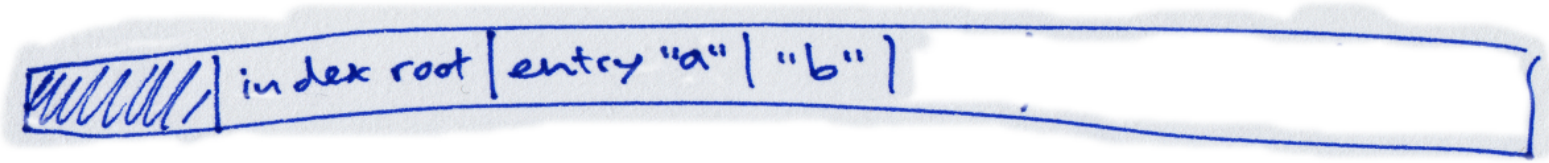
With the *Hachoir* framework:

```
181
182 class IndexRoot(FieldSet):
183     # Carrier 370 13.12
184     def createFields(self):
185         yield Enum(textHandler(UInt32(self, "ixtype", "indexed attribute type"), hexadecimal), ATTR_NAME)
186         yield UInt32(self, "collsort", "collation sorting rule")
187         yield UInt32(self,"ixrecsz(b)", "size of each index record (bytes)")
188         yield UInt8(self,"ixrecsz(c)", "size of each index record (clusters)")
189         yield PaddingBytes(self, "padding[]", 3)
190         yield IndexNodeHeader(self, 'nodeheader')
191         while self._current_size//8 <= self['nodeheader/used_offset'].value: # <=; for the lists ends with an
            empty entry with 'last' flag set
192             yield DirectoryIndexEntry(self, 'entry[]')
193
194
195 class IndexNodeHeader(FieldSet):
196     # Carrier 373 13.14
197     def createFields(self):
198         yield UInt32(self, "start_offset", "offset to start of index entry list")
199         yield UInt32(self, "used_offset", "offset to end of used portion of index entry list")
200         yield UInt32(self, "alloc_offset", "offset to end of allocated portion of index entry list")
201         yield Bit(self, "haschildren", "one or more ixentries in this node point to child nodes in $INDEX_ALLOCATION")
202         yield NullBits(self, "reserved[]", 31)
203         seekfwd = self['start_offset'].value*8 - self._current_size
204         if seekfwd:
205             yield RawBytes(self, "padding[]", seekfwd)
206
```

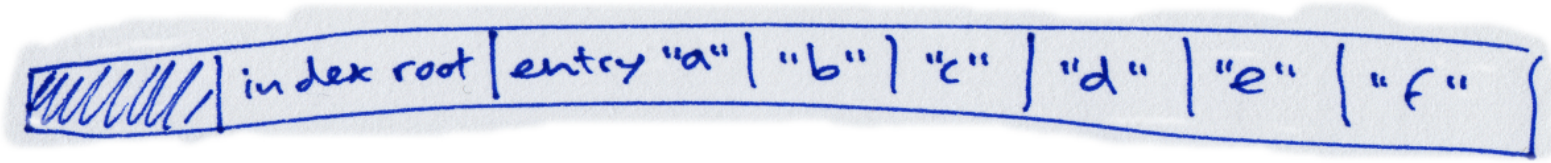
Growing a directory index



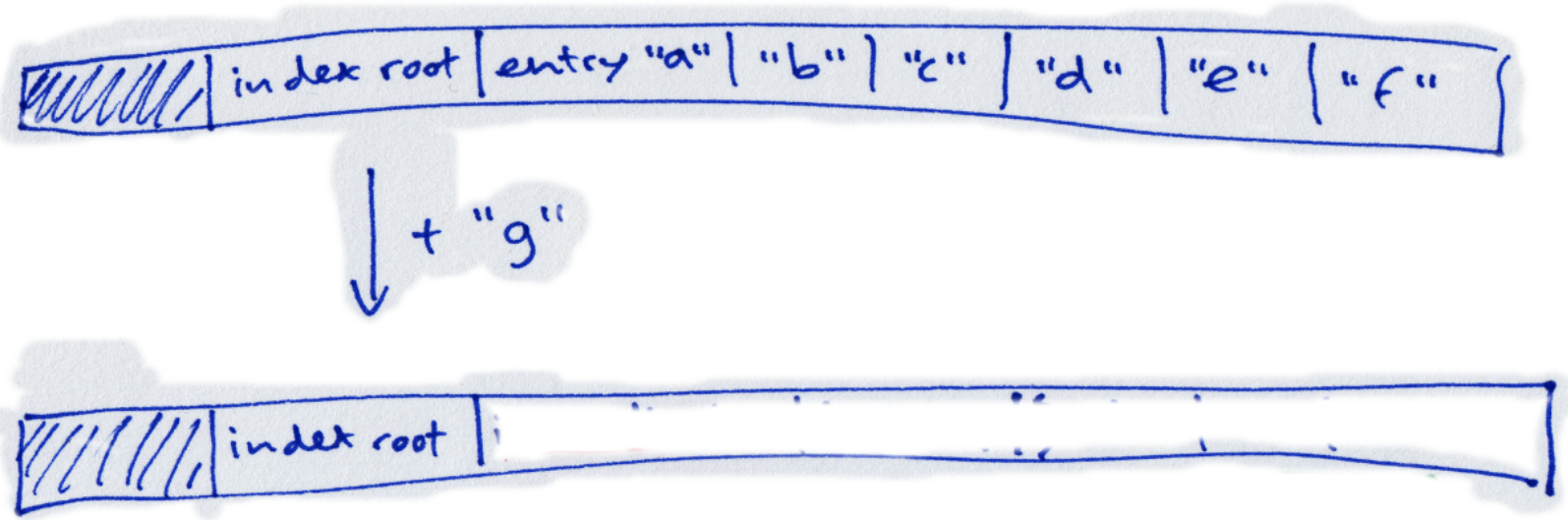
Growing a directory index



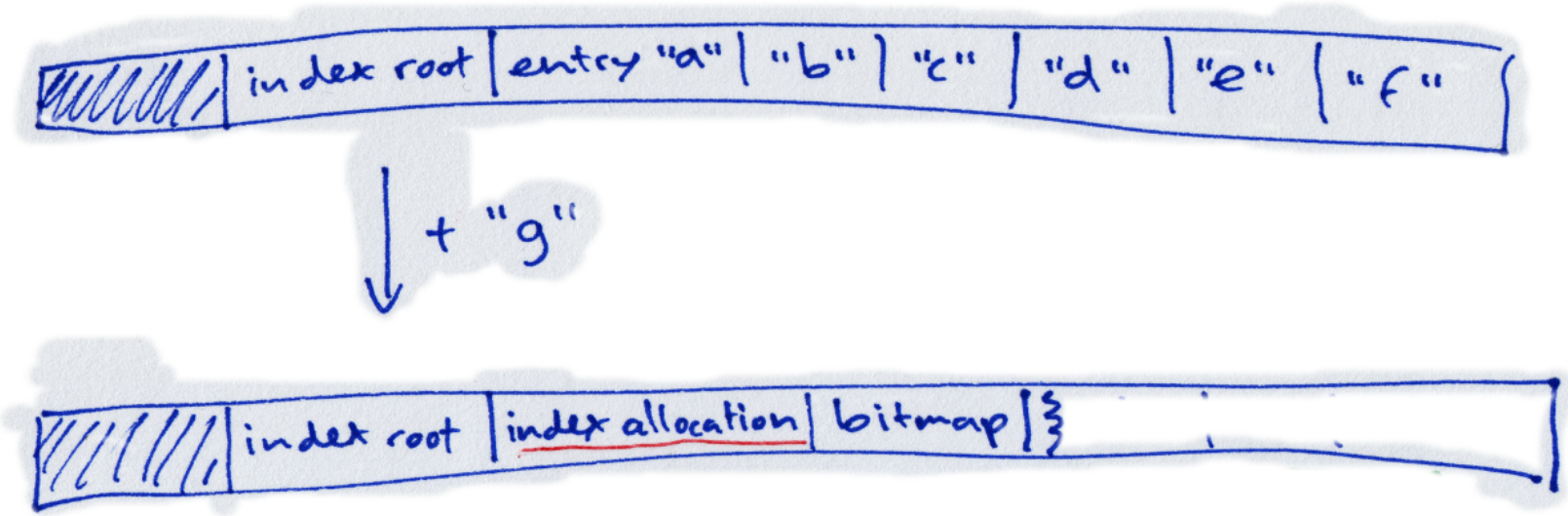
Growing a directory index



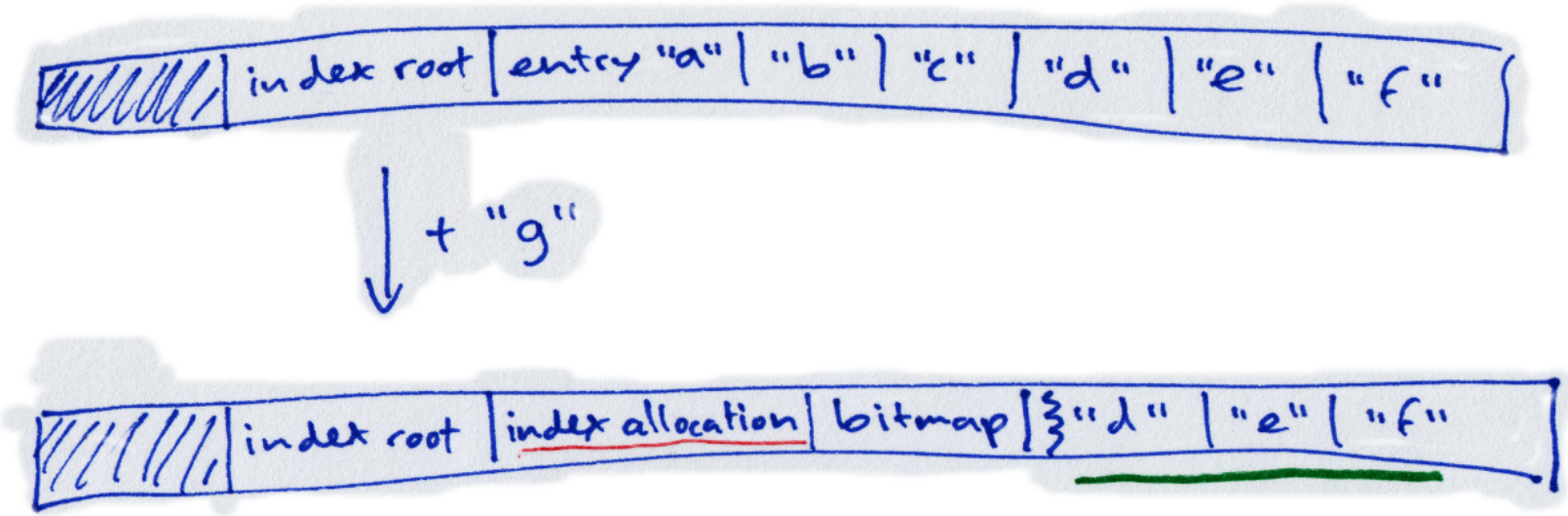
Growing a directory index



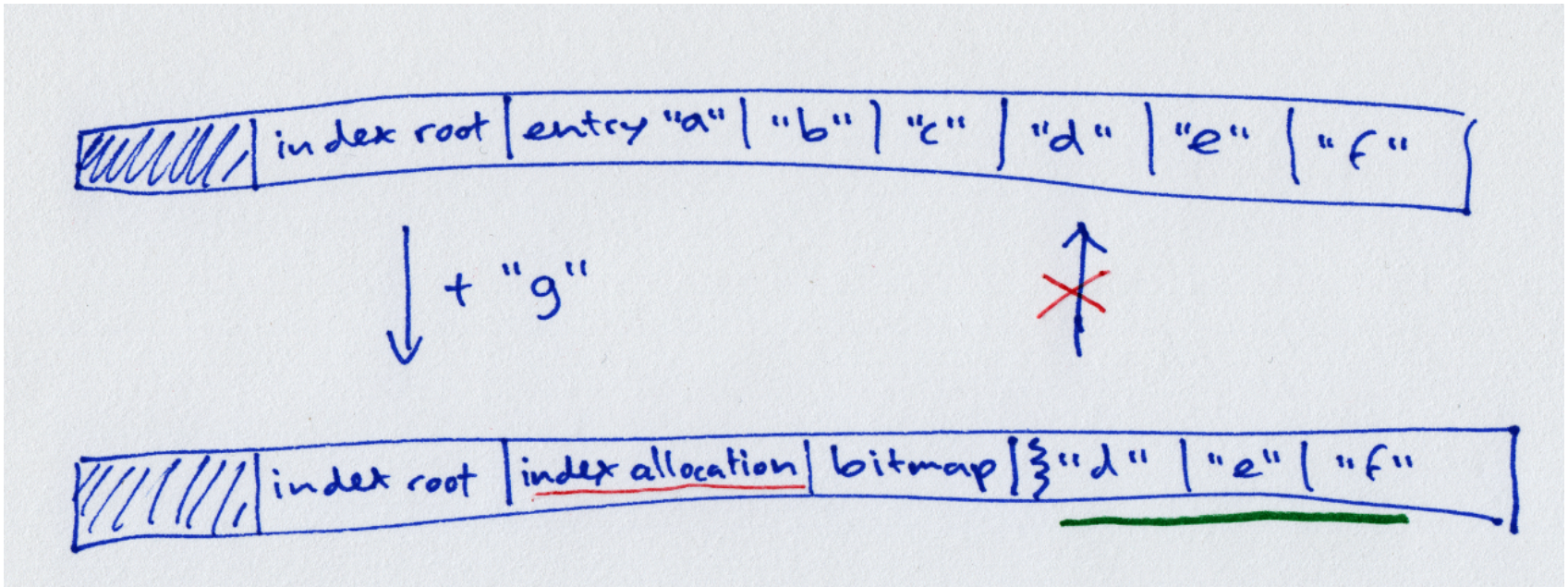
Growing a directory index



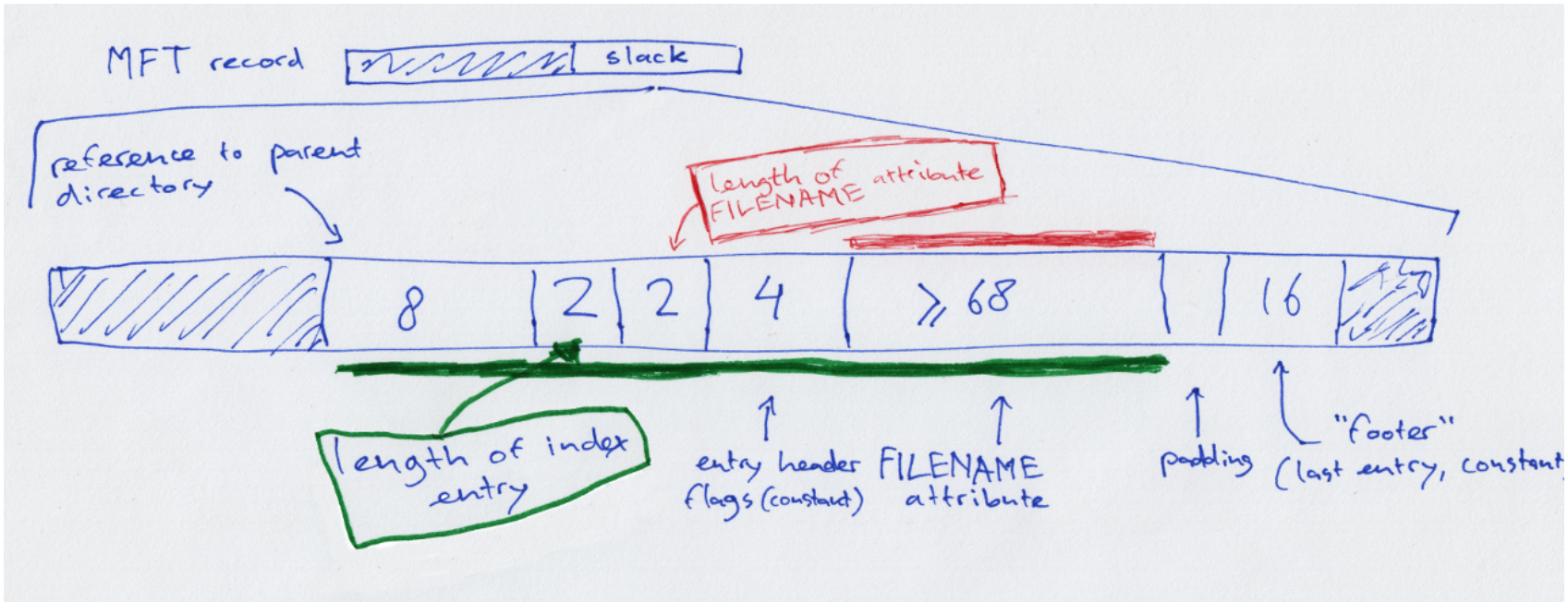
Growing a directory index



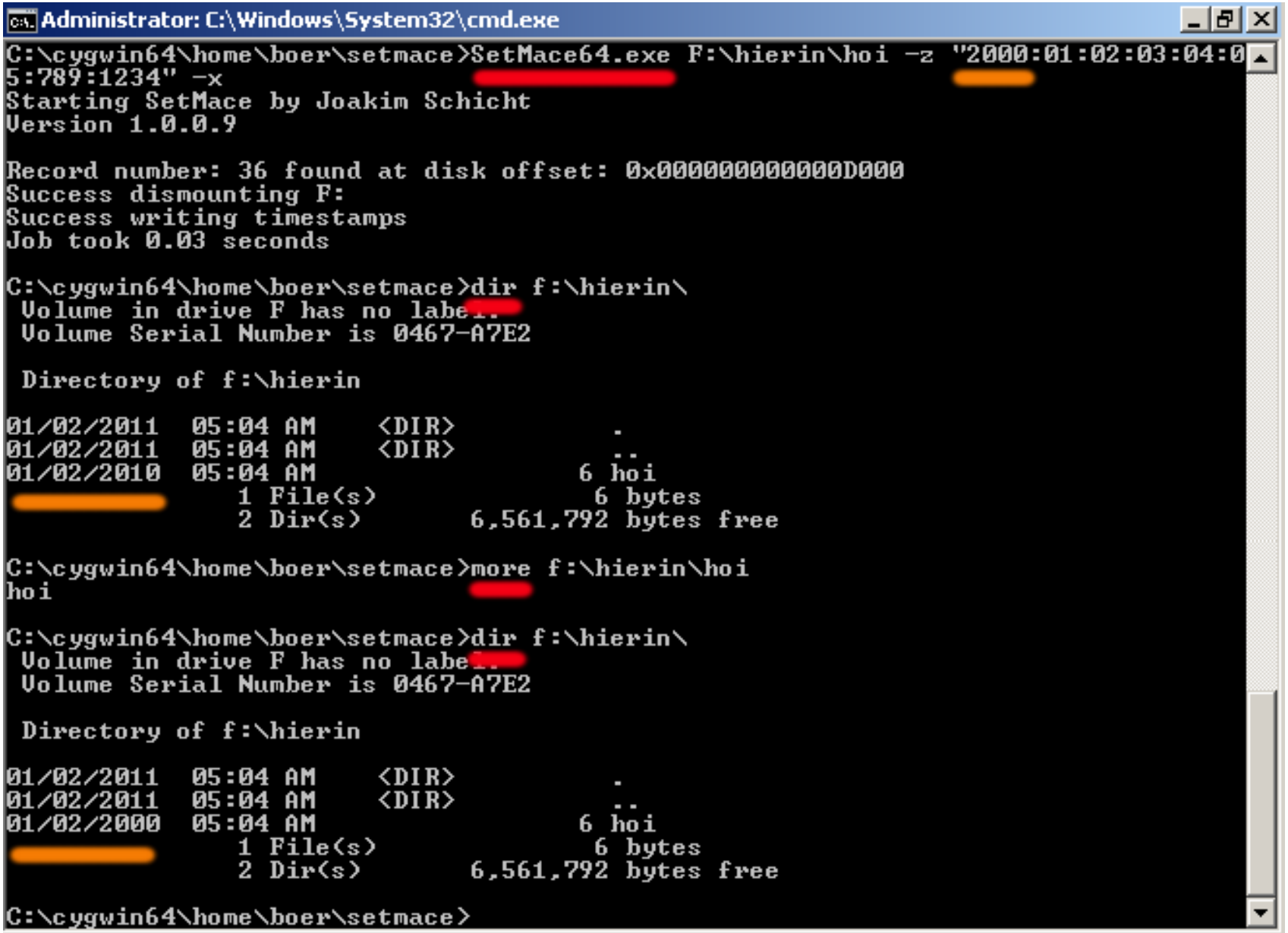
Growing a directory index



Carving root index entries from MFT slack



SetMACE directory indices



Fingerprinting timestamp relations (1)

What about self-inconsistencies in time stamps?

Fingerprinting timestamp relations (1)

What about self-inconsistencies in time stamps?
As the FILE_NAME timestamps are a snapshot of some earlier state of the STANDARD_INFORMATION timestamps... the former should always be less or equal to the latter, right?

Fingerprinting timestamp relations (1)

What about self-inconsistencies in time stamps?
As the FILE_NAME timestamps are a snapshot of some earlier state of the STANDARD_INFORMATION timestamps... the former should always be less or equal to the latter, right?

```
sqlite> .headers off
sqlite> select count() from mft
   ...> where isdir = 0 and isalloc = 1;
116660
sqlite> select count() from mft
   ...> where isdir = 0 and isalloc = 1
   ...> and sim >= fnm;
28534
sqlite> select count() from mft
   ...> where isdir = 0 and isalloc = 1
   ...> and sim < fnm;
88126
```

Fingerprinting timestamp relations (2)

An example fingerprint:

.

```
sia = sib < sim < fna = fnb = fnc = fnm < sic
```

.

Total number of possible configurations [7] :

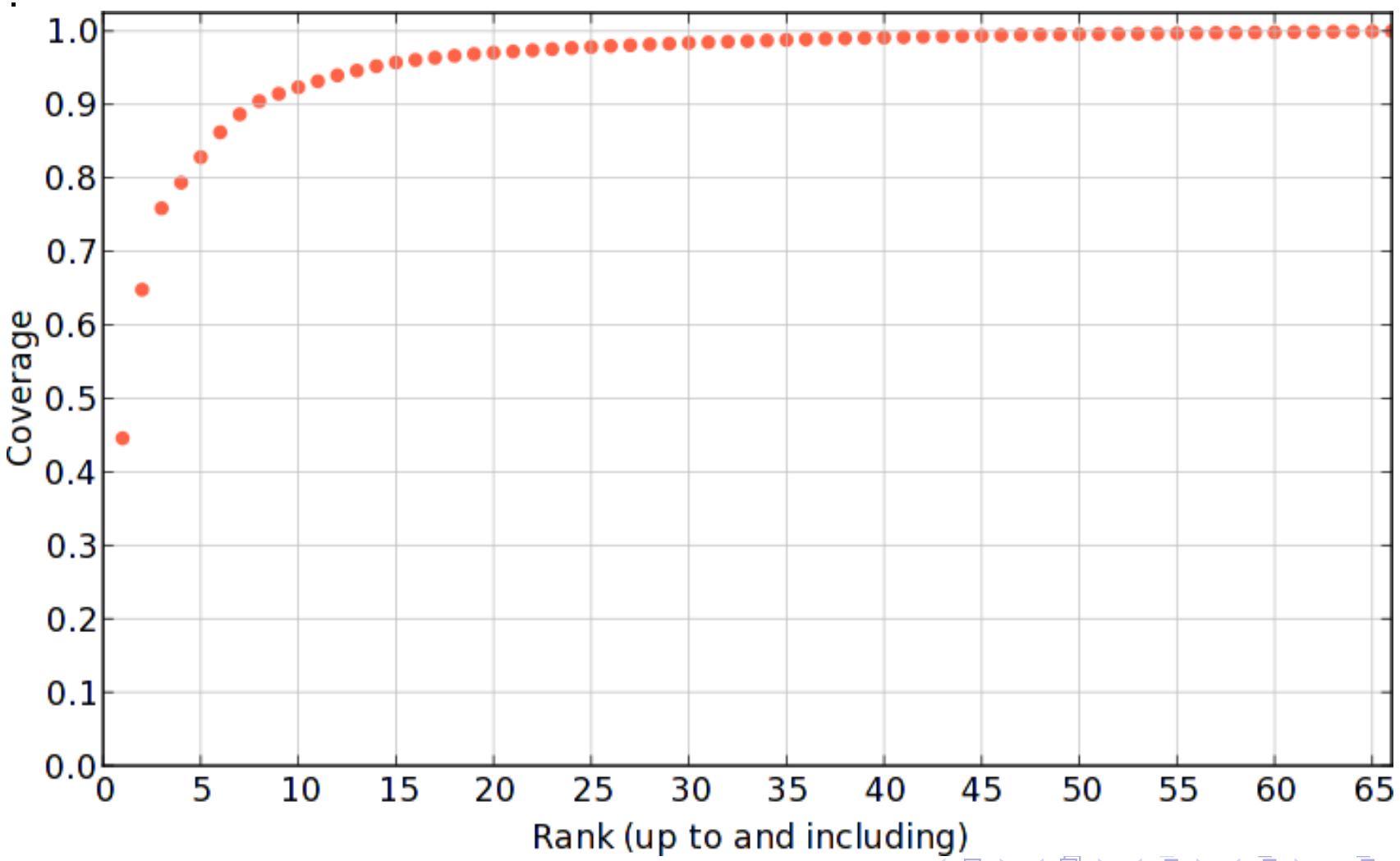
$$\sum_{x=1}^{7} \binom{7}{x-1} = 55581$$

[7] Sum of binomial coefficients

Wildtype timestamps

A skewed, but long-tailed distribution. Example: Cumulative distribution of timestamp fingerprints of EXE files on 1.5 years old Windows 7 system.

.



Conclusions

What forms of NTFS timestamp tampering can be detected by inspecting NTFS structures?
→ *It depends.* When it comes to finding inconsistencies;

- Index records *may* be overlooked by direct-access timestomping tools. However, Windows helpfully repairs resulting inconsistencies.[8]
- Old index records *may* be found in slack space.
- Wildtype timestamp configurations do not follow intuitions. Anomaly detection based on wildtype timestamp configuration frequencies *may* be of some use in the ranking phase.

[8] Next step: Extended consistency checker, for instance, cross-check each of the FN attributes in the multiple namespaces *and* the directory indices