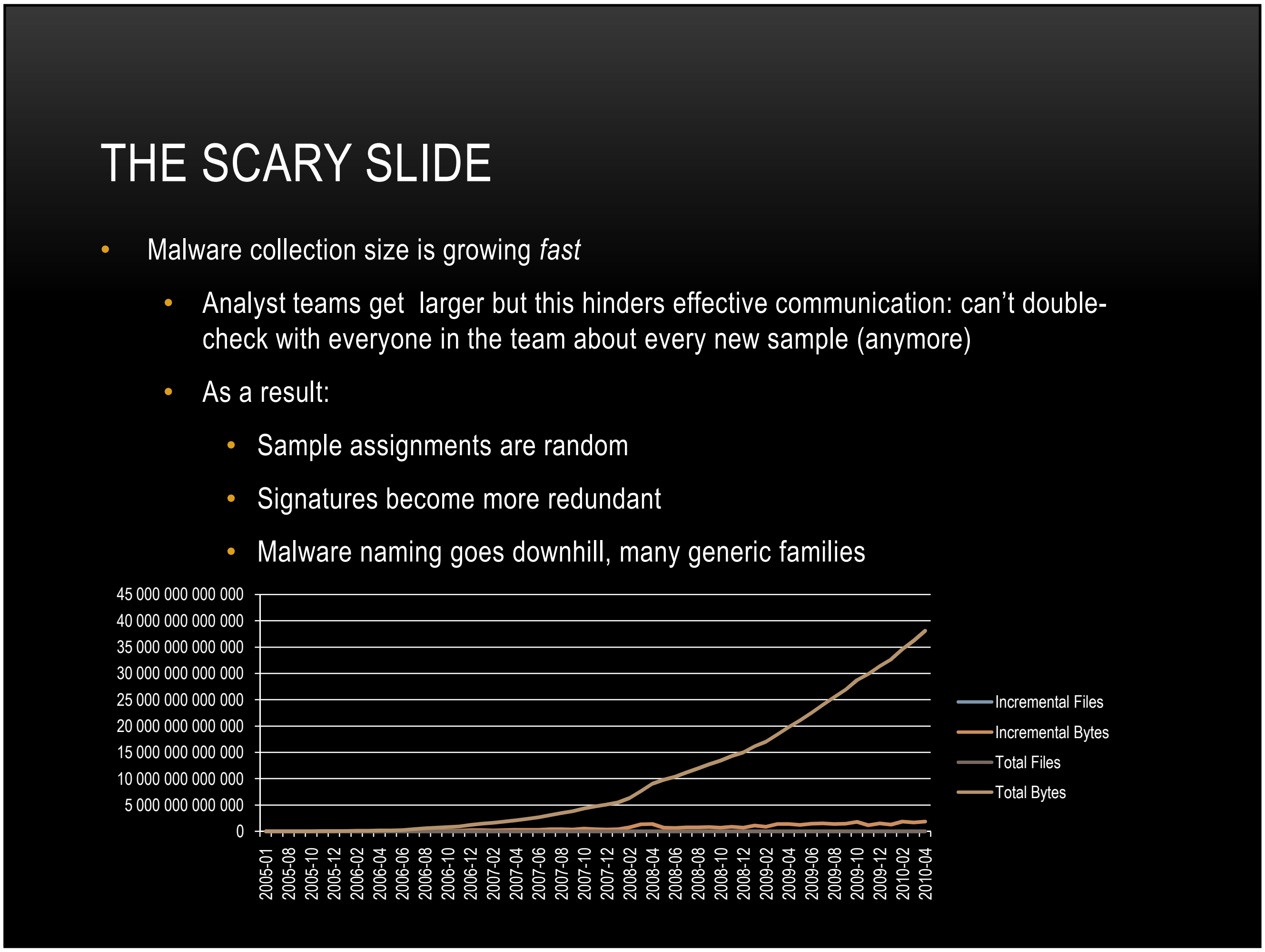
Tim Ebringer

54 69 6d 45 62 72 69 6e 67 65 72

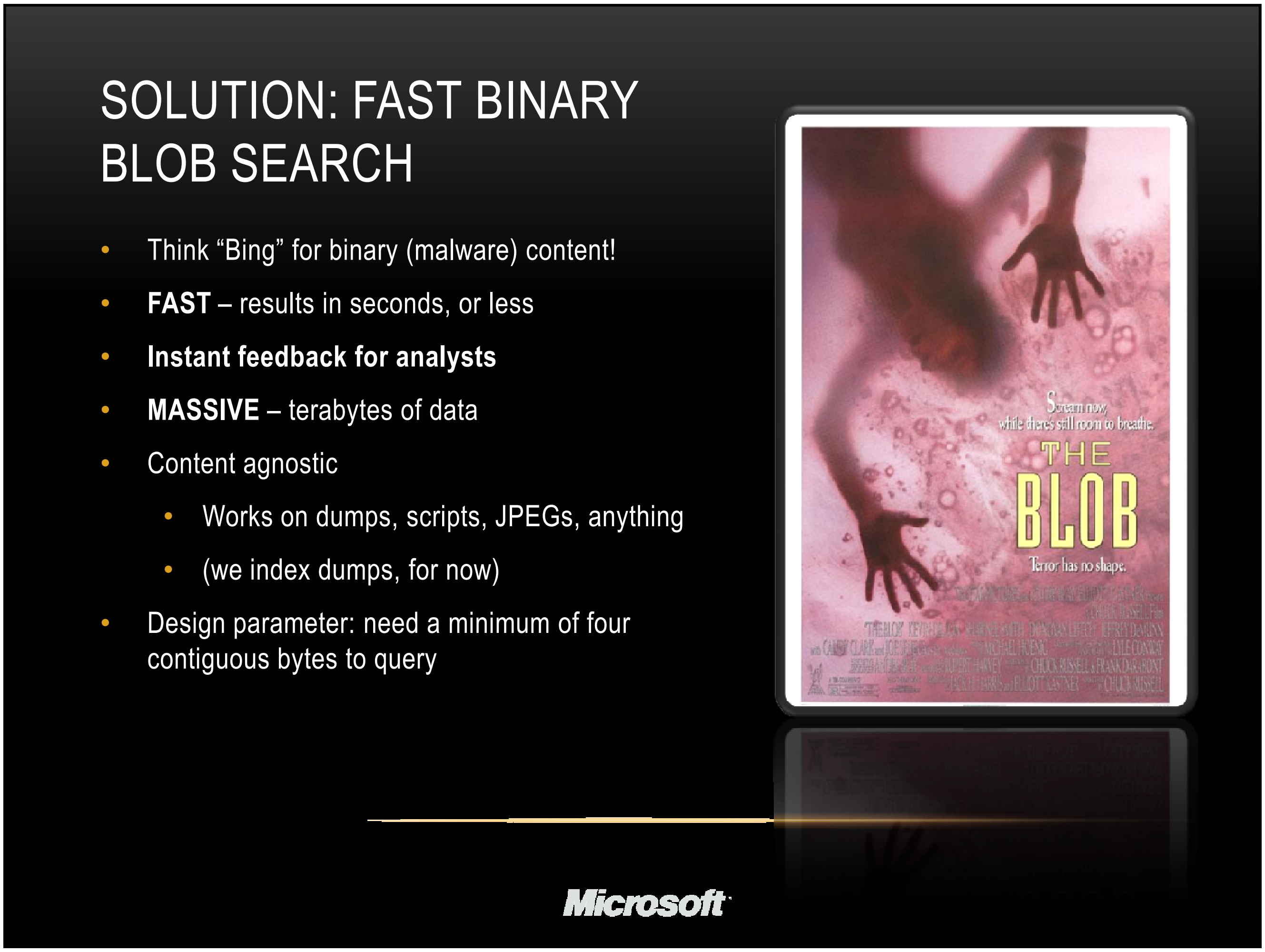
Marius Gheorghescu

4d 61 72 69 75 73 47 68 65 6f 72 67 68 65 73 63 75



THE SCARY SLIDE

- Malware collection size is growing *fast*
  - Analyst teams get larger but this hinders effective communication: can't double-check with everyone in the team about every new sample (anymore)
  - As a result:
    - Sample assignments are random
    - Signatures become more redundant
    - Malware naming goes downhill, many generic families



SOLUTION: FAST BINARY BLOB SEARCH

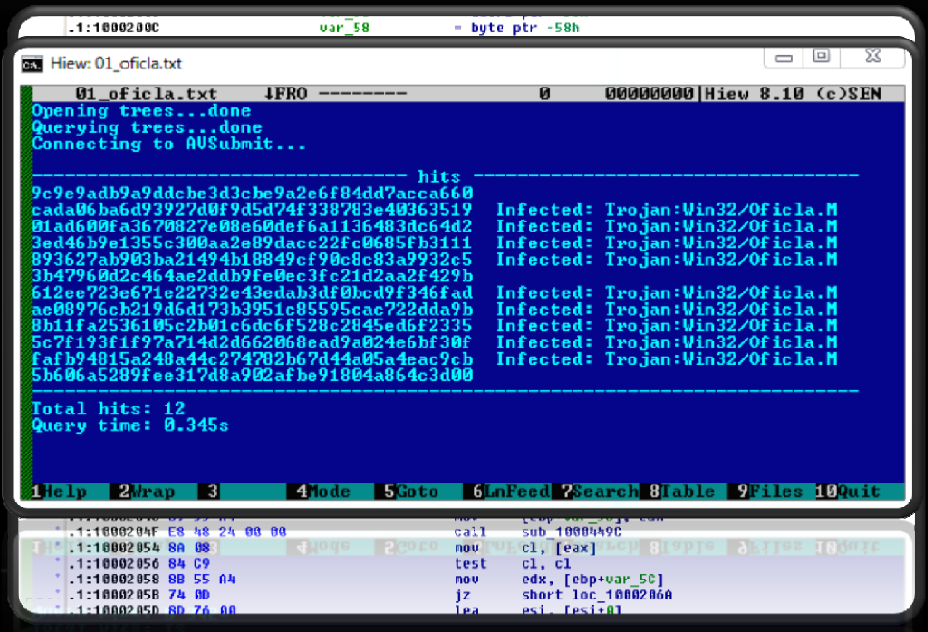
- Think “Bing” for binary (malware) content!
- **FAST** – results in seconds, or less
- **Instant feedback for analysts**
- **MASSIVE** – terabytes of data
- Content agnostic
  - Works on dumps, scripts, JPEGs, anything
  - (we index dumps, for now)
- Design parameter: need a minimum of four contiguous bytes to query

Microsoft

DEMO

NAMING

- Don’t know the name? Just grab interesting bytes from the dump and query



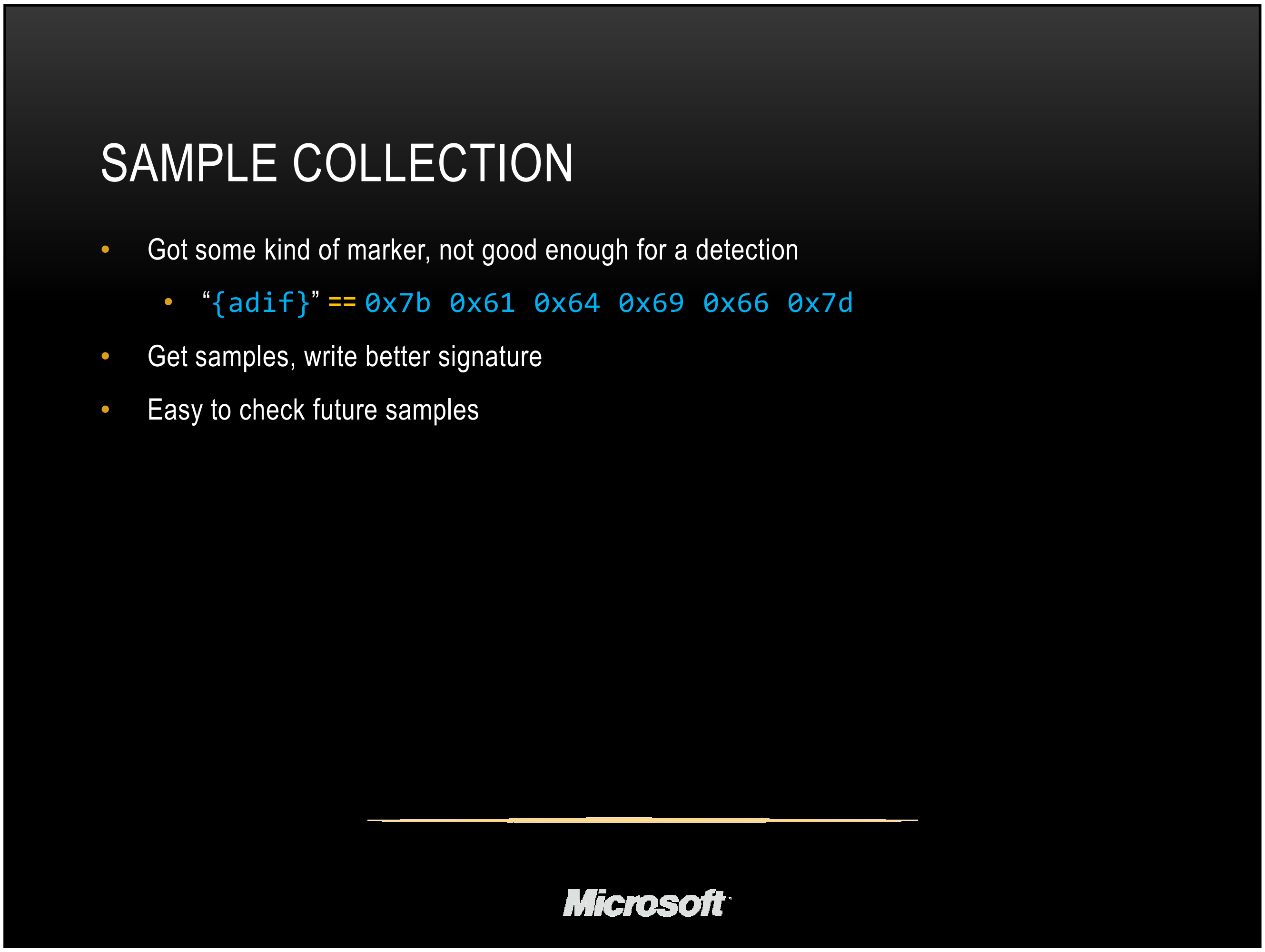
```
.1:1000200C                      var_58          = byte ptr -58h
```

```
   01_oficla.txt     ↓FRO --------              0     00000000|Hiew 8.10 (c)SEN
Opening trees...done
Querying trees...done
Connecting to AVSubmit...

----------------------------------- hits ------------------------------------
9c9e9adb9a9ddcbe3d3cbe9a2e6f84dd7acca660
cada06ba6d93927d0f9d5d74f338783e40363519  Infected: Trojan:Win32/Oficla.M
01ad600fa3670827e08e60def6a1136483dc64d2  Infected: Trojan:Win32/Oficla.M
3ed46b9e1355c300aa2e89dacc22fc0685fb3111  Infected: Trojan:Win32/Oficla.M
893627ab903ba21494b18849cf90c8c83a9932c5  Infected: Trojan:Win32/Oficla.M
3b47960d2c464ae2ddb9fe0ec3fc21d2aa2f429b
612ee723e671e22732e43edab3df0bcd9f346fad  Infected: Trojan:Win32/Oficla.M
ae08976cb219d6d173b3951c85595cac722dda9b  Infected: Trojan:Win32/Oficla.M
8b11fa2536105c2b01c6dc6f528c2845ed6f2335  Infected: Trojan:Win32/Oficla.M
5c7f193f1f97a714d2d662068ead9a024e6bf30f  Infected: Trojan:Win32/Oficla.M
fafb94815a248a44c274782b67d44a05a4eac9cb  Infected: Trojan:Win32/Oficla.M
5b606a5289fee317d8a902afbe91804a864c3d00
-----------------------------------------------------------------------------
Total hits: 12
Query time: 0.345s
```

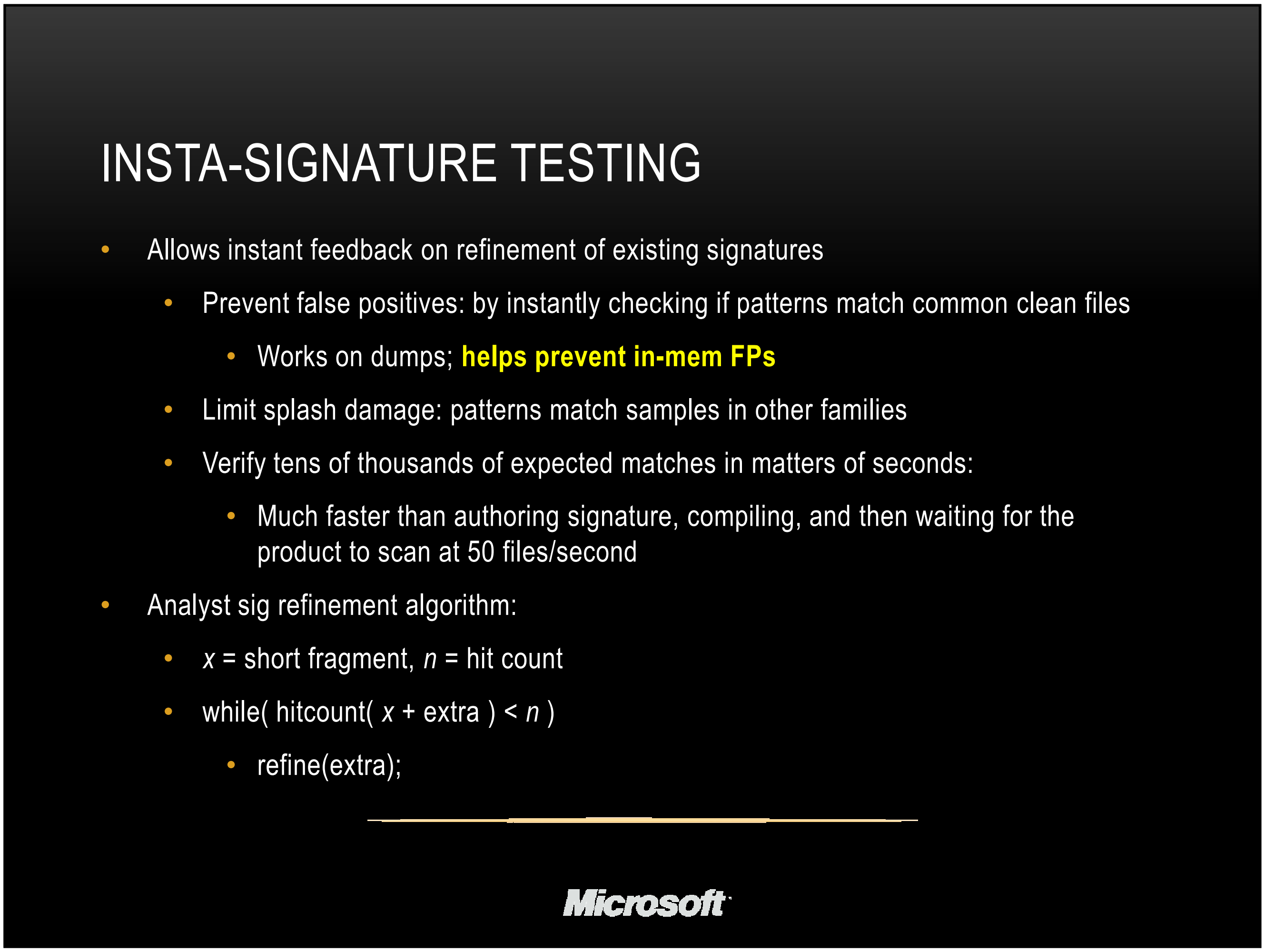
1Help 2Wrap 3 4Mode 5Goto 6LnFeed 7Search 8Table 9Files 10Quit

```
.1:1000204F E8 48 24 00 00          call    sub_1000449C
.1:10002054 8A 08                   mov     cl, [eax]
.1:10002056 84 C9                   test    cl, cl
.1:10002058 8B 55 A4                mov     edx, [ebp+var_5C]
.1:1000205B 74 0D                   jz      short loc_1000206A
.1:1000205D 8D 76 00                lea     esi, [esi+0]
```



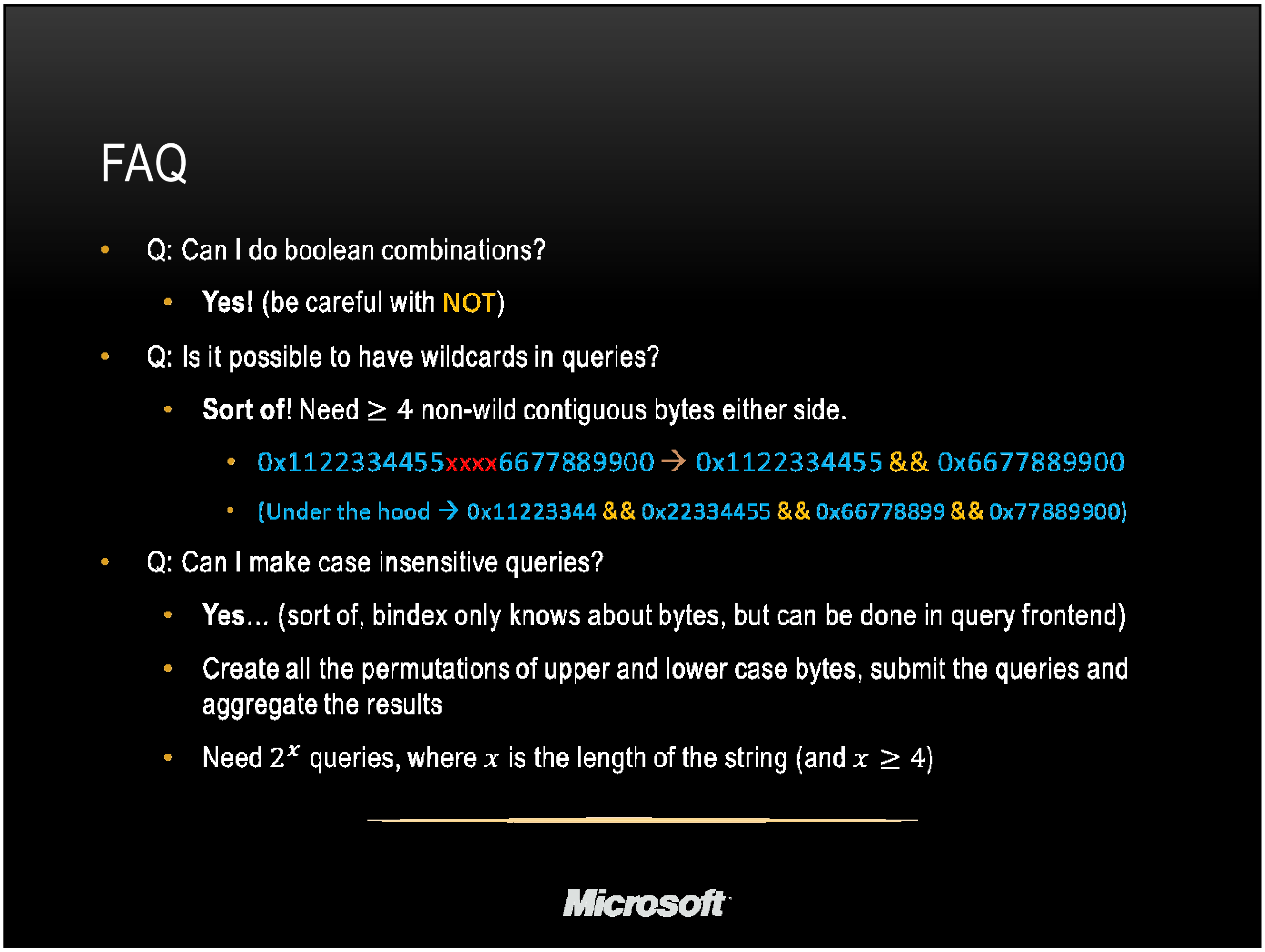
SAMPLE COLLECTION

- Got some kind of marker, not good enough for a detection
  - “`{adif}`” == `0x7b 0x61 0x64 0x69 0x66 0x7d`
- Get samples, write better signature
- Easy to check future samples



INSTA-SIGNATURE TESTING

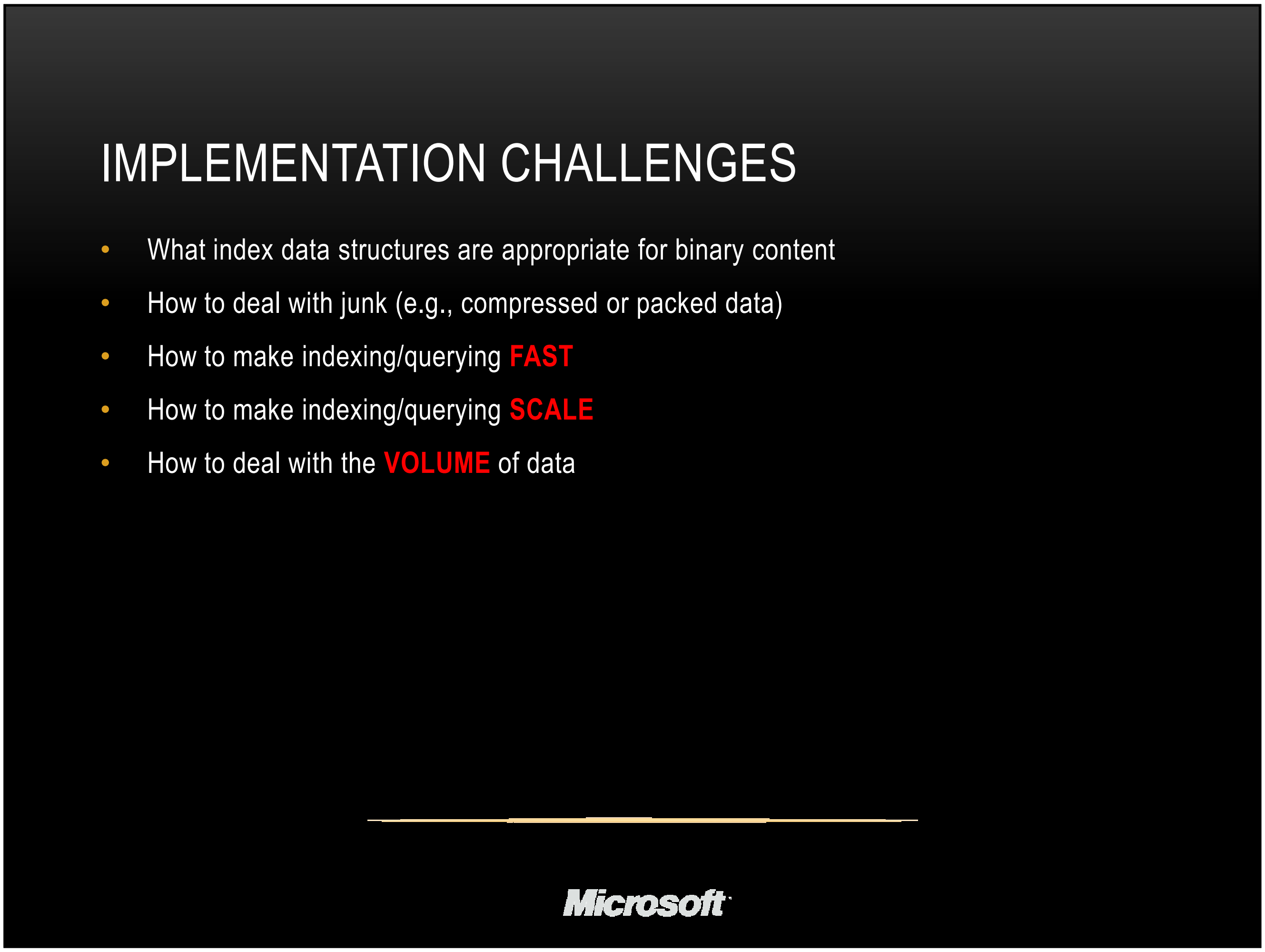
- Allows instant feedback on refinement of existing signatures
  - Prevent false positives: by instantly checking if patterns match common clean files
    - Works on dumps; **helps prevent in-mem FPs**
  - Limit splash damage: patterns match samples in other families
  - Verify tens of thousands of expected matches in matters of seconds:
    - Much faster than authoring signature, compiling, and then waiting for the product to scan at 50 files/second
- Analyst sig refinement algorithm:
  - *x* = short fragment, *n* = hit count
  - while(hitcount(*x* + extra) < *n*)
    - refine(extra);



FAQ

- Q: Can I do boolean combinations?
  - **Yes!** (be careful with NOT)
- Q: Is it possible to have wildcards in queries?
  - **Sort of!** Need ≥ 4 non-wild contiguous bytes either side.
    - 0x1122334455xxxx6677889900 → 0x1122334455 && 0x6677889900
    - (Under the hood → 0x11223344 && 0x22334455 && 0x66778899 && 0x77889900)
- Q: Can I make case insensitive queries?
  - **Yes**… (sort of, bindex only knows about bytes, but can be done in query frontend)
  - Create all the permutations of upper and lower case bytes, submit the queries and aggregate the results
  - Need 2^x queries, where x is the length of the string (and $x \geq 4$)

Microsoft



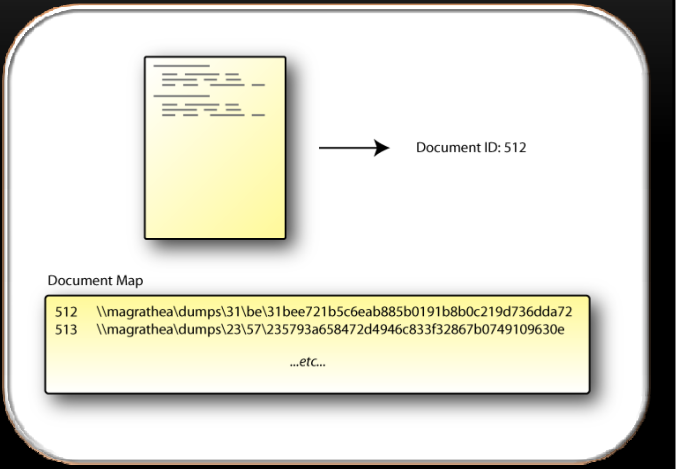
IMPLEMENTATION CHALLENGES

- What index data structures are appropriate for binary content
- How to deal with junk (e.g., compressed or packed data)
- How to make indexing/querying **FAST**
- How to make indexing/querying **SCALE**
- How to deal with the **VOLUME** of data

Microsoft

INDEXING 101

- Each document is assigned a unique Document ID
- This identifier, and the document it refers to, is stored in the Document Map
- Each document is tokenised, and each token is then associated with the document ID in a structure that makes it very efficient to look up tokens. This process is called "**inverting the index**"

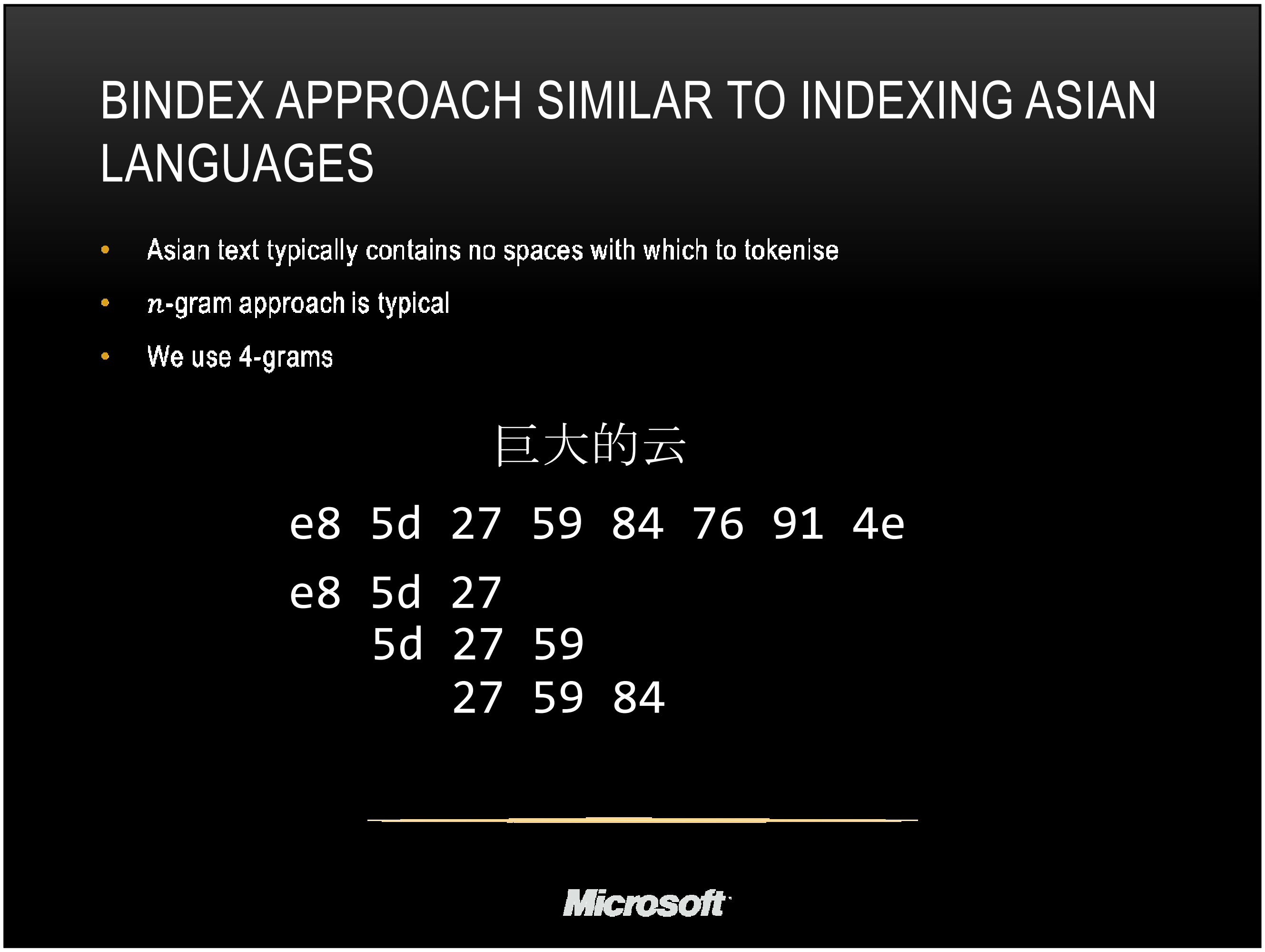


Document Map

512	\\magrathea\dumps\31\be\31bee721b5c6eab885b0191b8b0c219d736dda72
513	\\magrathea\dumps\23\57\235793a658472d4946c833f32867b0749109630e

...etc...

Microsoft



BINDEX APPROACH SIMILAR TO INDEXING ASIAN LANGUAGES

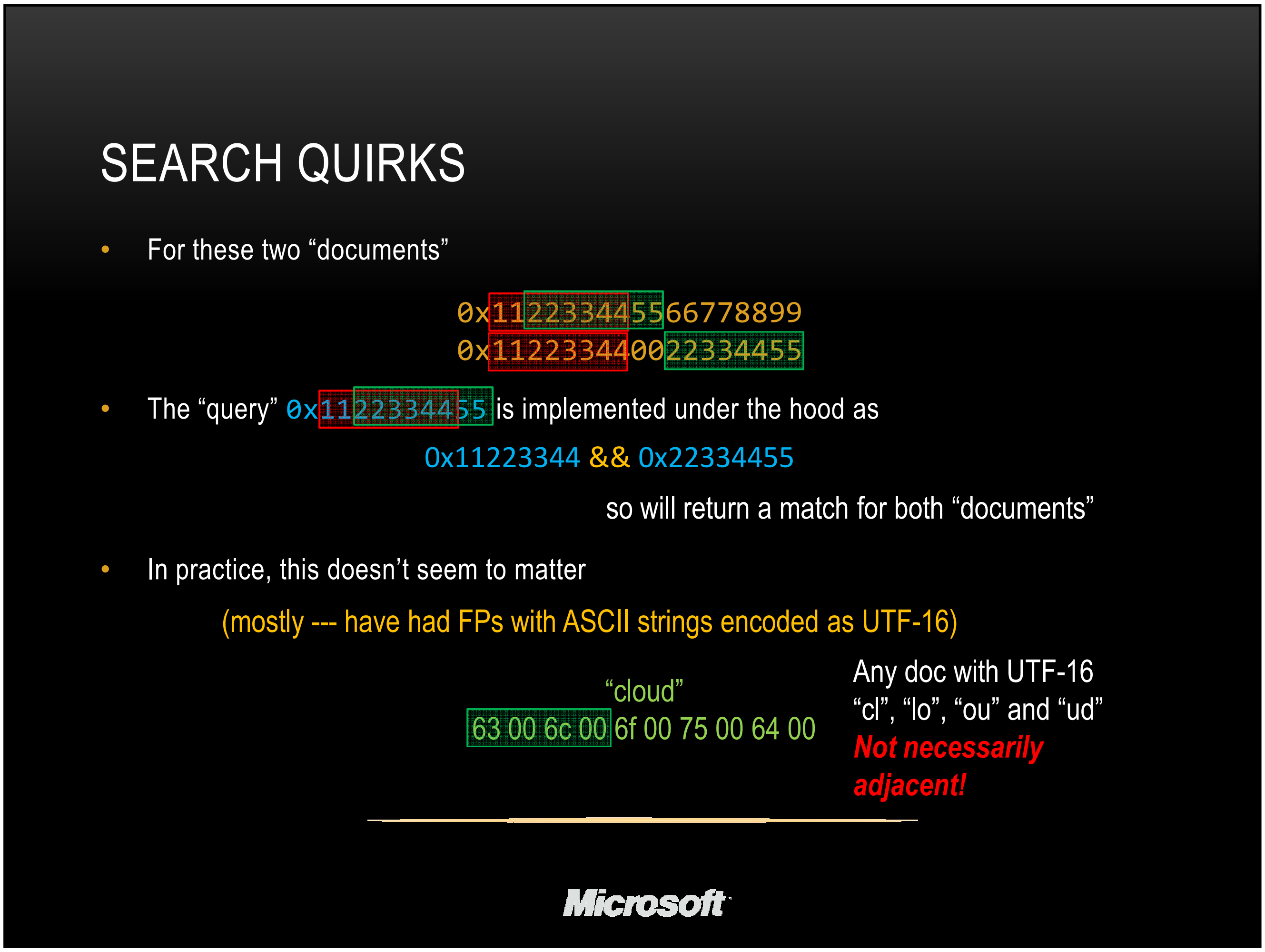
- Asian text typically contains no spaces with which to tokenise
- n-gram approach is typical
- We use 4-grams

巨大的云

```
e8 5d 27 59 84 76 91 4e

e8 5d 27
   5d 27 59
      27 59 84
```

Microsoft



SEARCH QUIRKS

- For these two “documents”

```
0x1122334455566778899
0x112233440022334455
```

- The “query” `0x112233445 5` is implemented under the hood as

  `0x11223344 && 0x22334455`

  so will return a match for both “documents”

- In practice, this doesn’t seem to matter

  (mostly --- have had FPs with ASCII strings encoded as UTF-16)

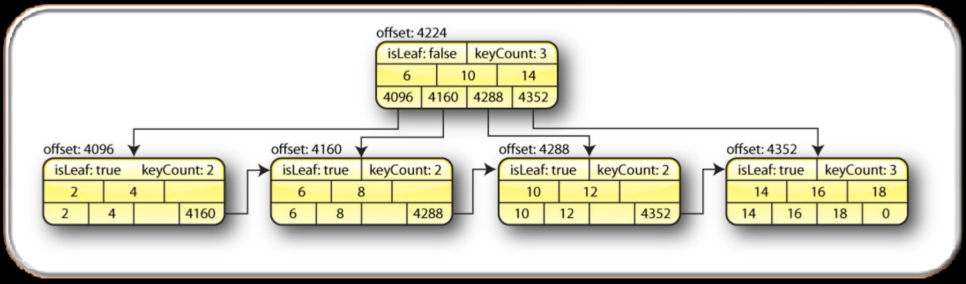
  “cloud”

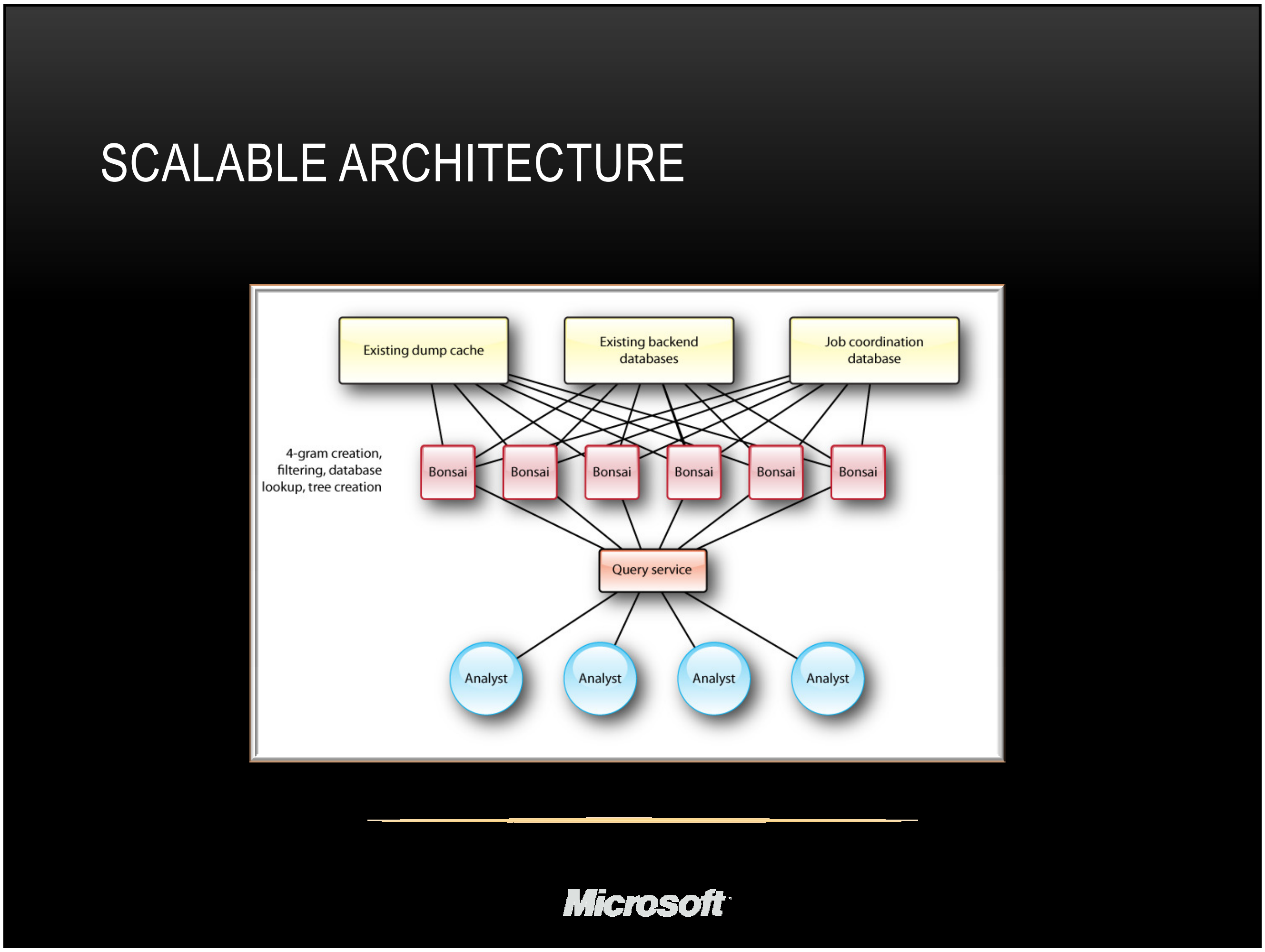
  63 00 6c 00 6f 00 75 00 64 00

  Any doc with UTF-16 “cl”, “lo”, “ou” and “ud” ***Not necessarily adjacent!***

STORING THE N-GRAMS

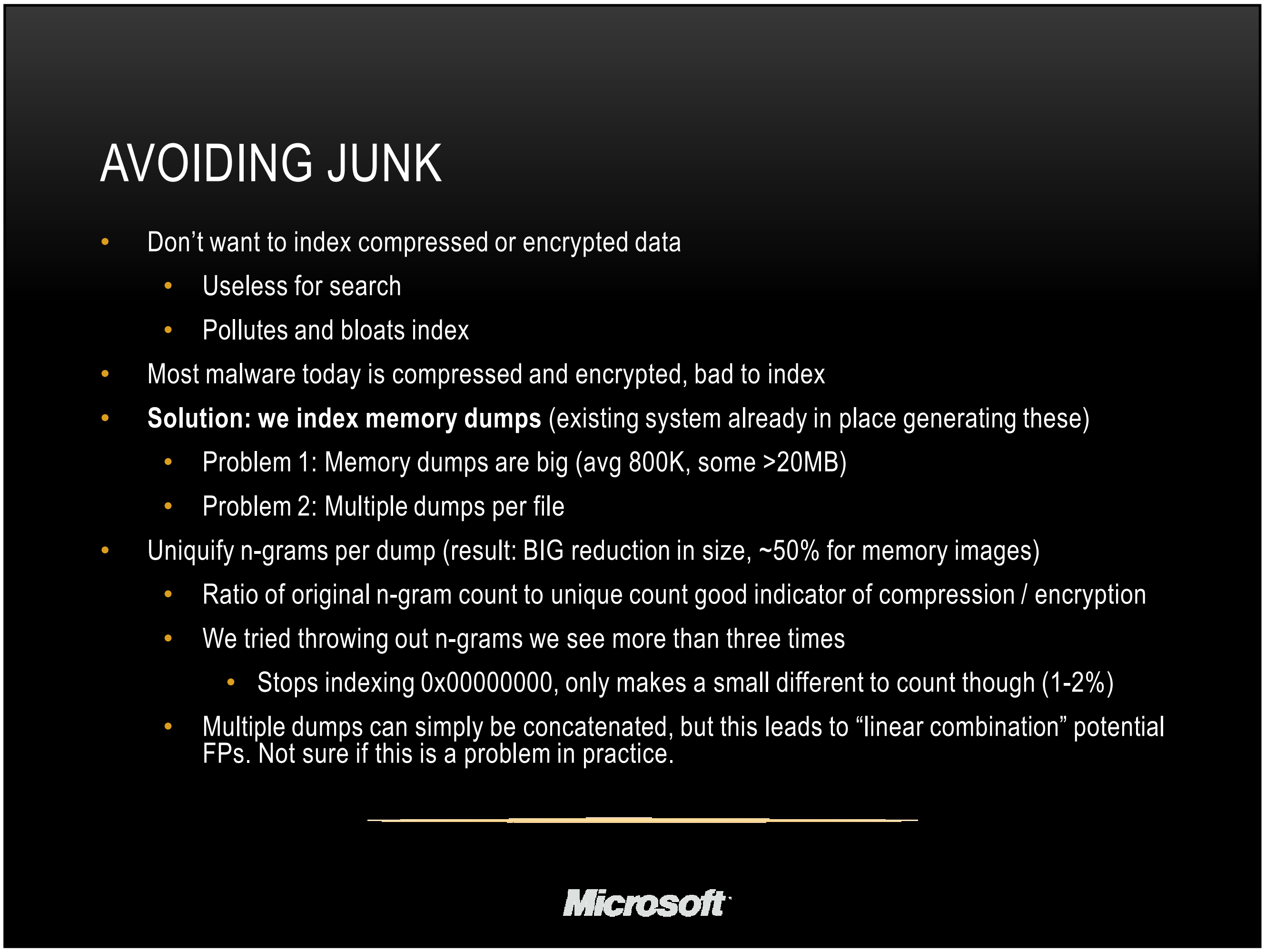
- B+ trees are ideal
  - Designed for external storage use, very high fan-out minimises disk hits, node size aligned with disk block size
  - Leaf traversal capability helps with duplicate key / value pairs
  - If Document ID is ulong / uint, can jam it directly into the leaves
  - Some implementations keep a parent offset as part of the node (convenient, but bad idea)





SCALABLE ARCHITECTURE

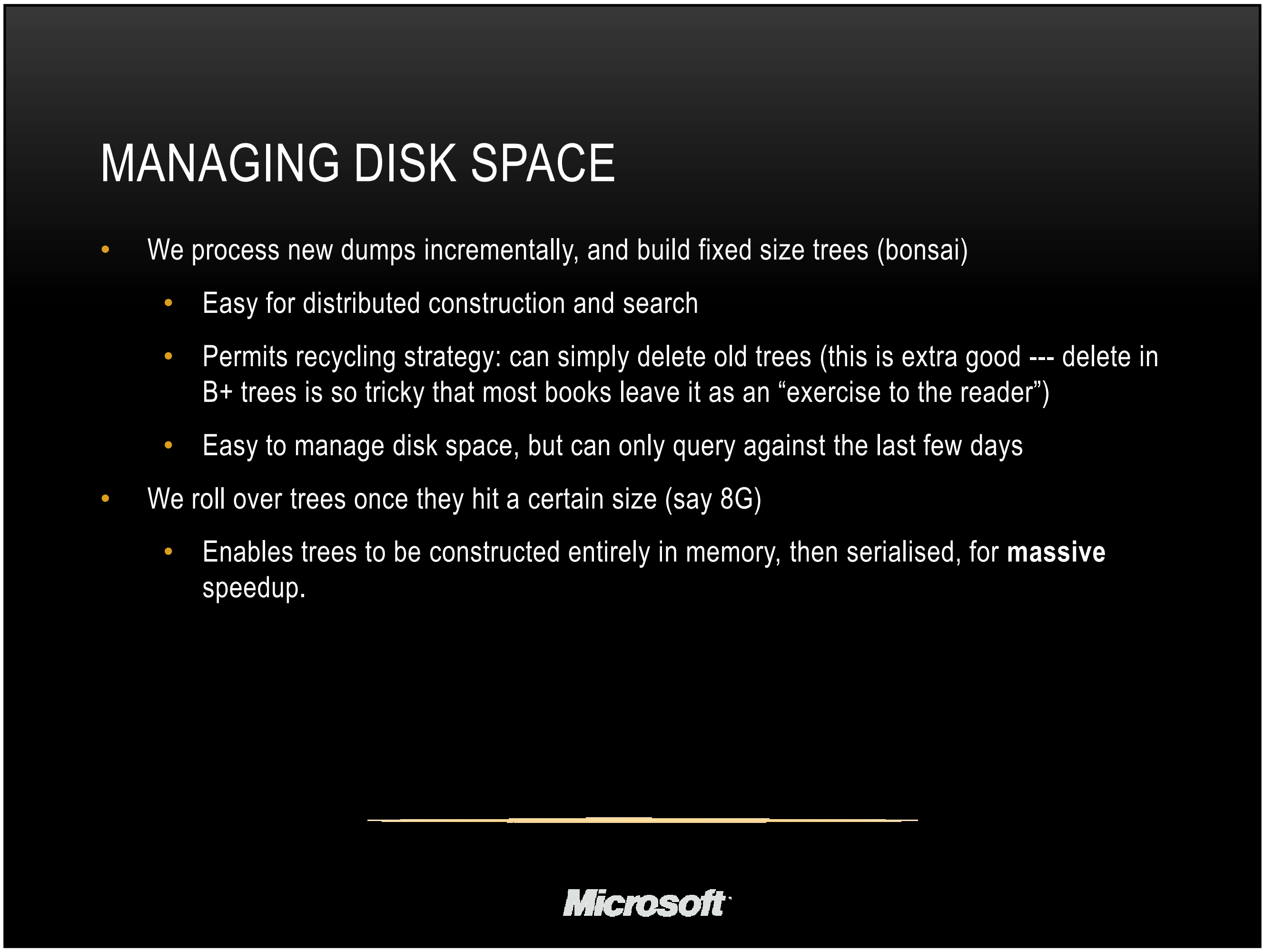
Existing dump cache
Existing backend databases
Job coordination database
4-gram creation, filtering, database lookup, tree creation
Bonsai
Bonsai
Bonsai
Bonsai
Bonsai
Bonsai
Query service
Analyst
Analyst
Analyst
Analyst
Microsoft



AVOIDING JUNK

- Don't want to index compressed or encrypted data
  - Useless for search
  - Pollutes and bloats index
- Most malware today is compressed and encrypted, bad to index
- **Solution: we index memory dumps** (existing system already in place generating these)
  - Problem 1: Memory dumps are big (avg 800K, some >20MB)
  - Problem 2: Multiple dumps per file
- Uniquify n-grams per dump (result: BIG reduction in size, ~50% for memory images)
  - Ratio of original n-gram count to unique count good indicator of compression / encryption
  - We tried throwing out n-grams we see more than three times
    - Stops indexing 0x00000000, only makes a small different to count though (1-2%)
  - Multiple dumps can simply be concatenated, but this leads to "linear combination" potential FPs. Not sure if this is a problem in practice.

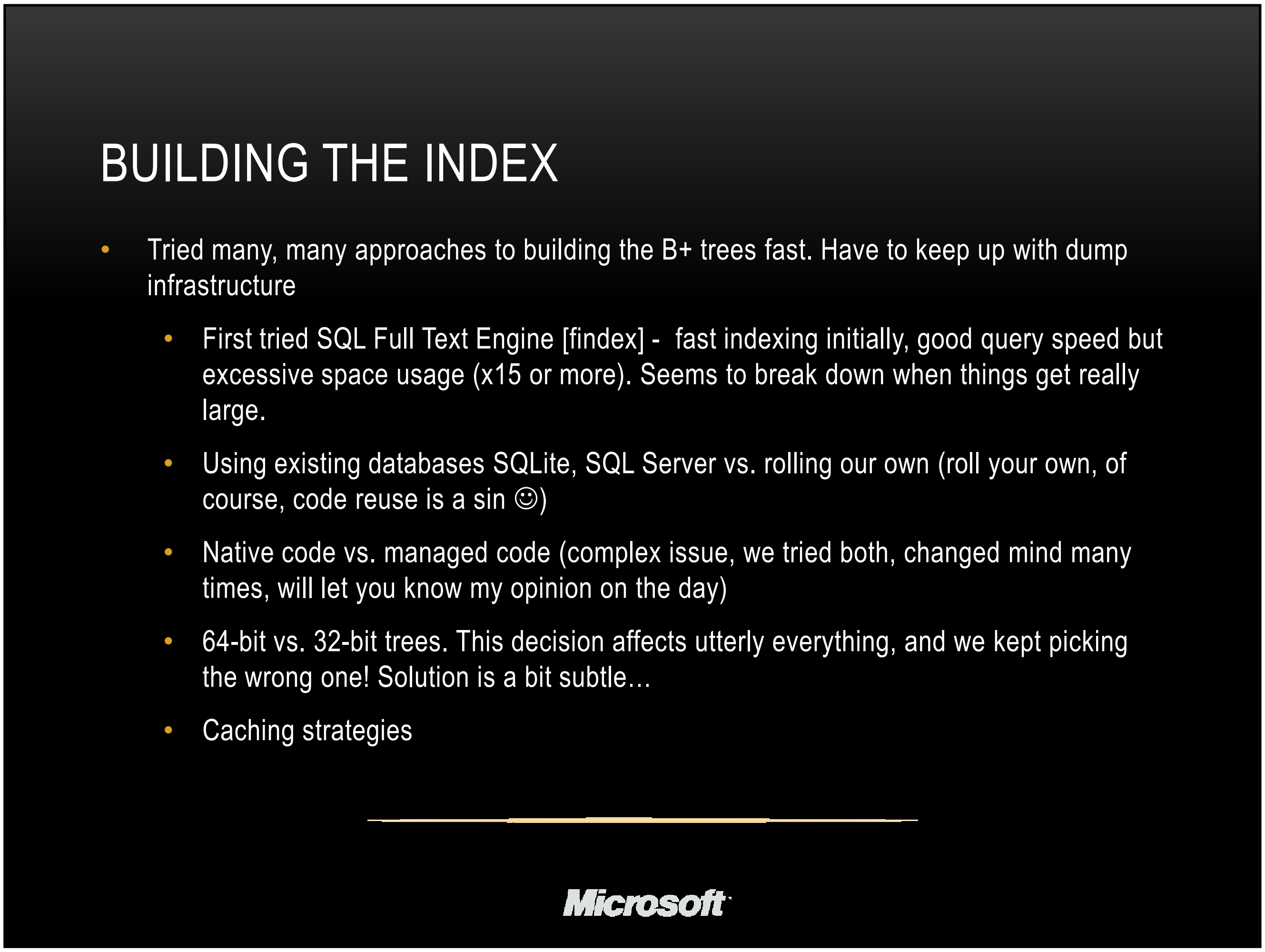
Microsoft



MANAGING DISK SPACE

- We process new dumps incrementally, and build fixed size trees (bonsai)
  - Easy for distributed construction and search
  - Permits recycling strategy: can simply delete old trees (this is extra good --- delete in B+ trees is so tricky that most books leave it as an “exercise to the reader”)
  - Easy to manage disk space, but can only query against the last few days
- We roll over trees once they hit a certain size (say 8G)
  - Enables trees to be constructed entirely in memory, then serialised, for **massive** speedup.

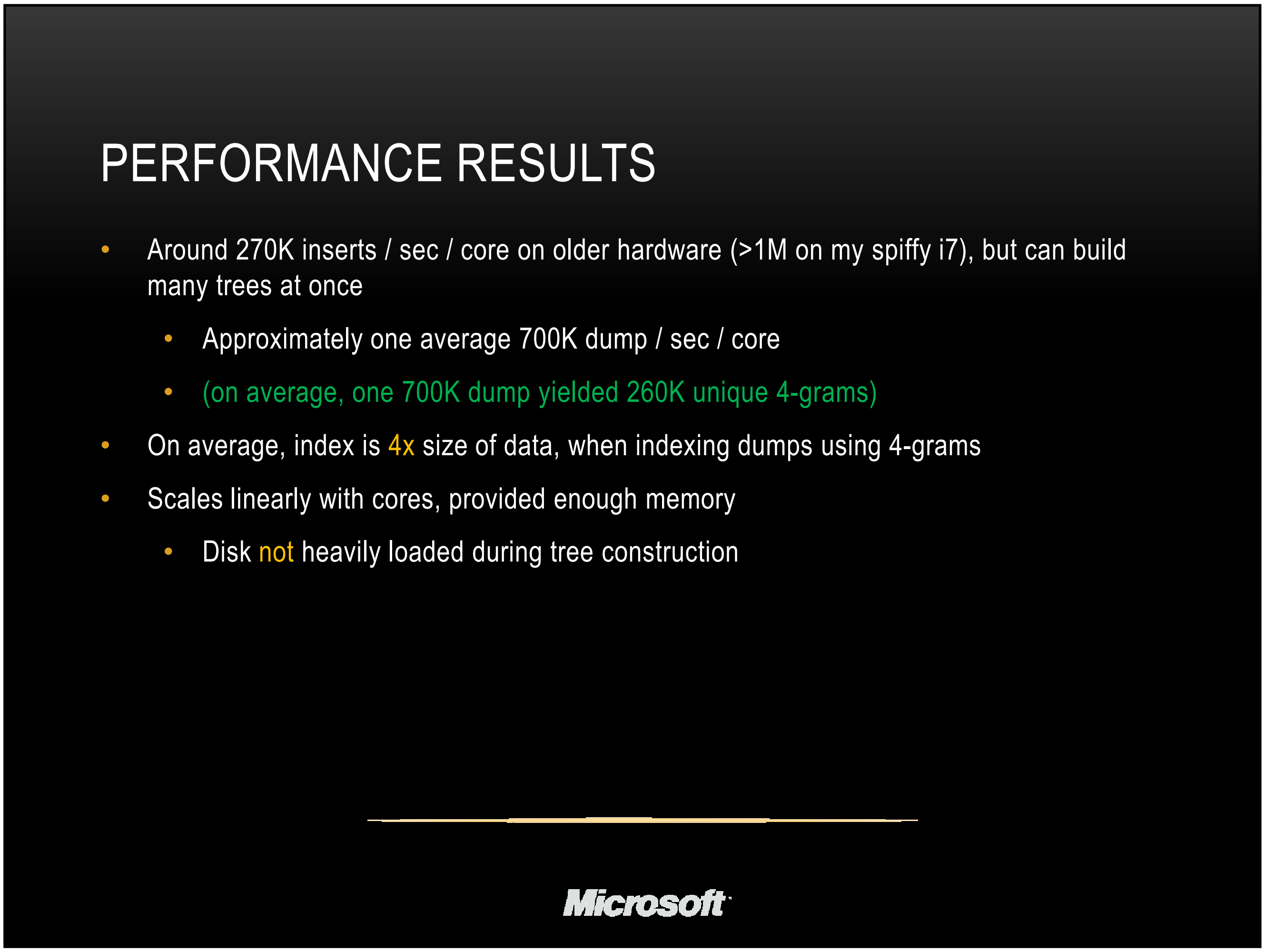
Microsoft



BUILDING THE INDEX

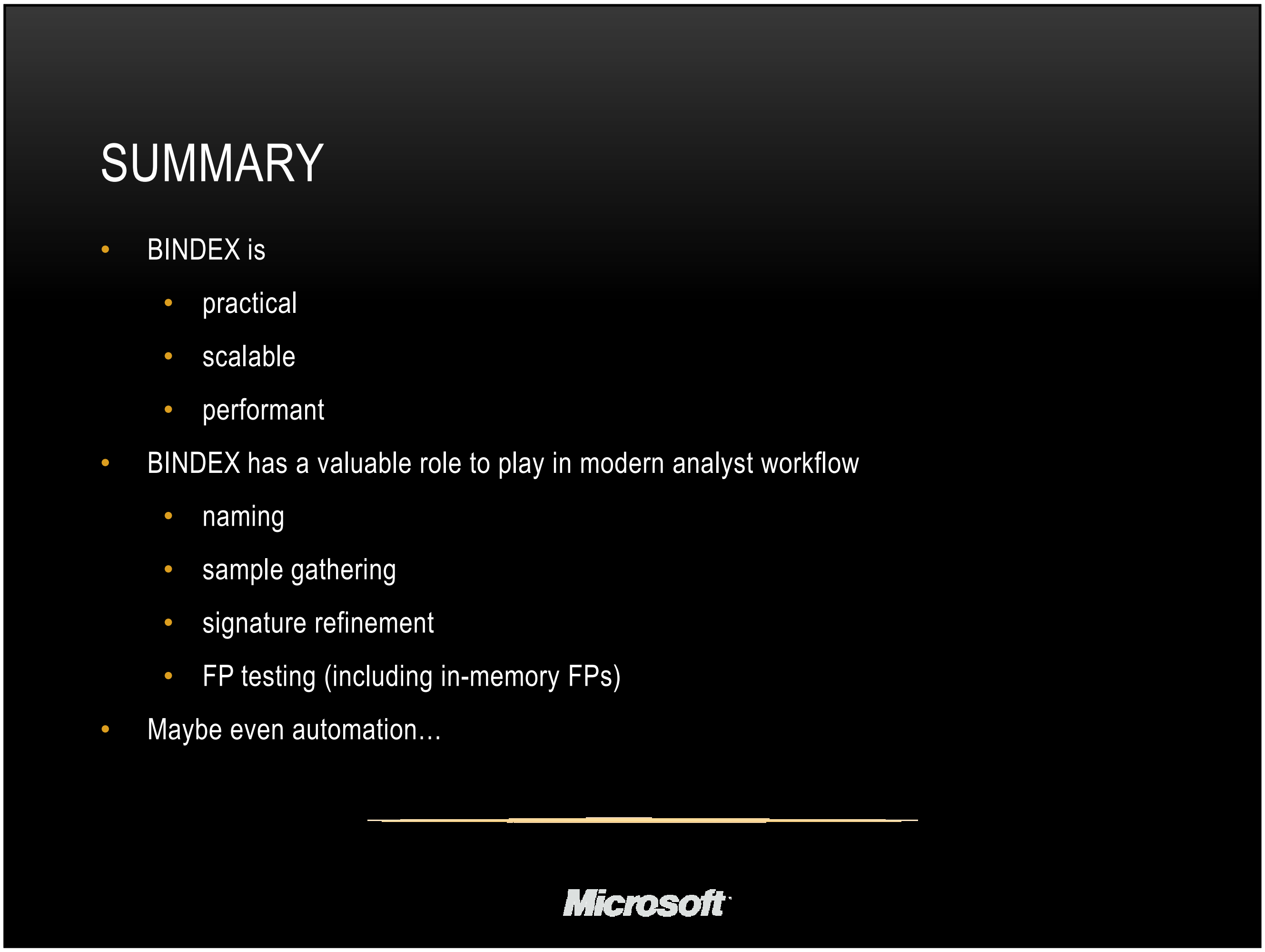
- Tried many, many approaches to building the B+ trees fast. Have to keep up with dump infrastructure
  - First tried SQL Full Text Engine [findex] - fast indexing initially, good query speed but excessive space usage (x15 or more). Seems to break down when things get really large.
  - Using existing databases SQLite, SQL Server vs. rolling our own (roll your own, of course, code reuse is a sin ☺)
  - Native code vs. managed code (complex issue, we tried both, changed mind many times, will let you know my opinion on the day)
  - 64-bit vs. 32-bit trees. This decision affects utterly everything, and we kept picking the wrong one! Solution is a bit subtle…
  - Caching strategies

Microsoft



PERFORMANCE RESULTS

- Around 270K inserts / sec / core on older hardware (>1M on my spiffy i7), but can build many trees at once
  - Approximately one average 700K dump / sec / core
  - (on average, one 700K dump yielded 260K unique 4-grams)
- On average, index is 4x size of data, when indexing dumps using 4-grams
- Scales linearly with cores, provided enough memory
  - Disk not heavily loaded during tree construction



SUMMARY

- BINDEX is
  - practical
  - scalable
  - performant
- BINDEX has a valuable role to play in modern analyst workflow
  - naming
  - sample gathering
  - signature refinement
  - FP testing (including in-memory FPs)
- Maybe even automation…

THE END

Microsoft