Timing Attacks for Recovering Private Entries From Database Engines

Ariel Waissbein

joint work with A. Futoransky, D. Saura and P. Varangot

-Core Security Technologies-

id: HT1-302

Thursday, April 10

Objectives

- Understand the risk associated to DB's data loss.

- Expose new attack vector against DBMSs.
  - Show how B-tree indexing leaks information.
  - Prove that a timing attack works.
  - Describe an exploitation technique and its merits.

- Prove this by attacking MySQL and MS SQL DBMSs, and discuss countermeasures.

Structure

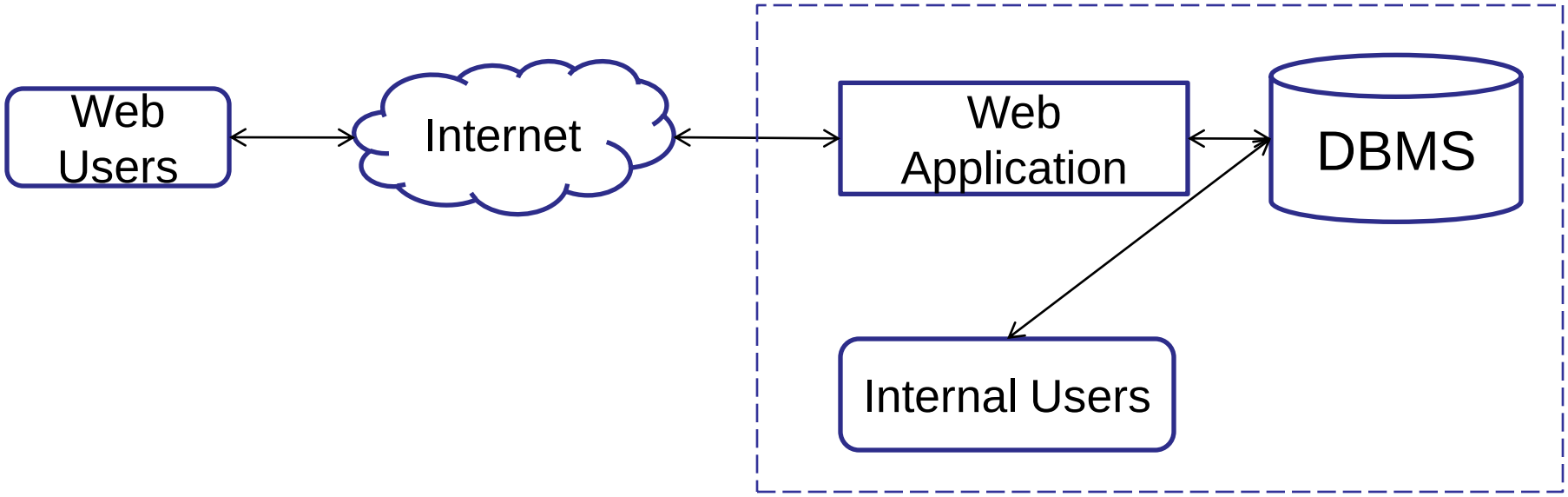
1. Introduce the problem & main result.

2. Theory
   1. Walk through DbMSs' storage inner works.
   2. Describe how does this lead to attacks.
   3. Design an attack.

3. Practice
   1. Exhibit the attacks against MySQL.
   2. Discuss attacks against MS SQL.
   3. Discuss countermeasures and extensions.

Databases store valuable information that must be secured

- Organizations store info from their users / clients, plus their own info.
- Then DBMSs and the servers that host them are interesting for attackers.



Hackers have compromised DBMSs in several ways

1. Insecure web server hosting the DB.
   - Insecure configuration, lack of patching, …
2. A SQL-injection vulnerability in the web application.
   - Insecure development of the webapp.
3. Lax permissions and privilege levels in the DB.
   - An ``outsider'' connects to the server and compromises an insecure authentication protocol.
   - A legitimate user siphons out confidential data.

All these are known, and countermeasures have been studied and deployed.

Main result: scenario

- Consider a table in one deployed database management system (e.g., MySQL, MS SQL, Oracle, …)
- Users cannot retrieve data from one column directly, but can insert values in this “privacy-sensitive” column, which is indexed by a B-tree.
- Users can measure the response time of the INSERT transactions they make.

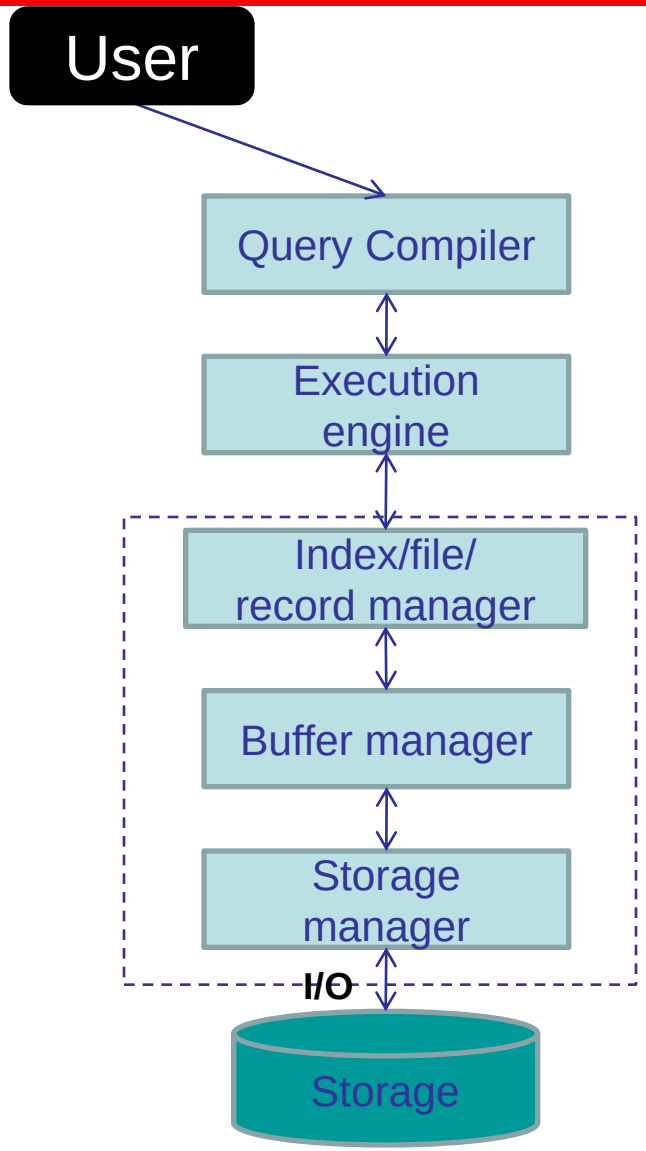
Main result: thesis

- Then an attacker, passing as a user, can retrieve the values of this column.
  - The success of the attack depends on the accuracy to time inserts and other parameters.
  - The “cost” of the attack can be measured by the number of inserts it requires.
  - The number of inserts required is proportional to the size (in bits) of these values, times the number of values retrieved.

Corroborated for MySQL and MS SQL!!!

INSIDE THE DBMS

The cost of I/O dominates the cost of operations



- Data is stored in “sorted chunks,” i.e., pages.
- Queries flow from User to Storage, and back.
- The cost of page I/O dominates the cost of typical DB operations.
- I/O cost depends on how data is sorted in the DB.

Data is stored according to a table representation

- DBMSs use table as for data representation.
- Rows must be sorted according to **some of the columns** for efficient searches and modifications.

Name	Passport	Football team
Cacho	32102806	San Lorenzo
Pedro	25061305	River
Tomas	9567205	Racing

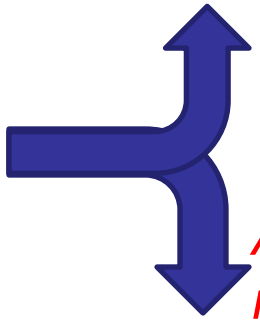
Data is sorted by indexing algorithms into tables

- Each database table must have one *primary* index.
- Data is then stored in Storage (the hard drive) in nodes which come in either form:

A page in an Unclustered index

9567205, p_1	25061305, p_2	32102806, p_3

Pass.	Data
9,567,205	Tomas, Racing
25,061,305	Pedro, River
32,102,806	Cacho, San Lorenzo

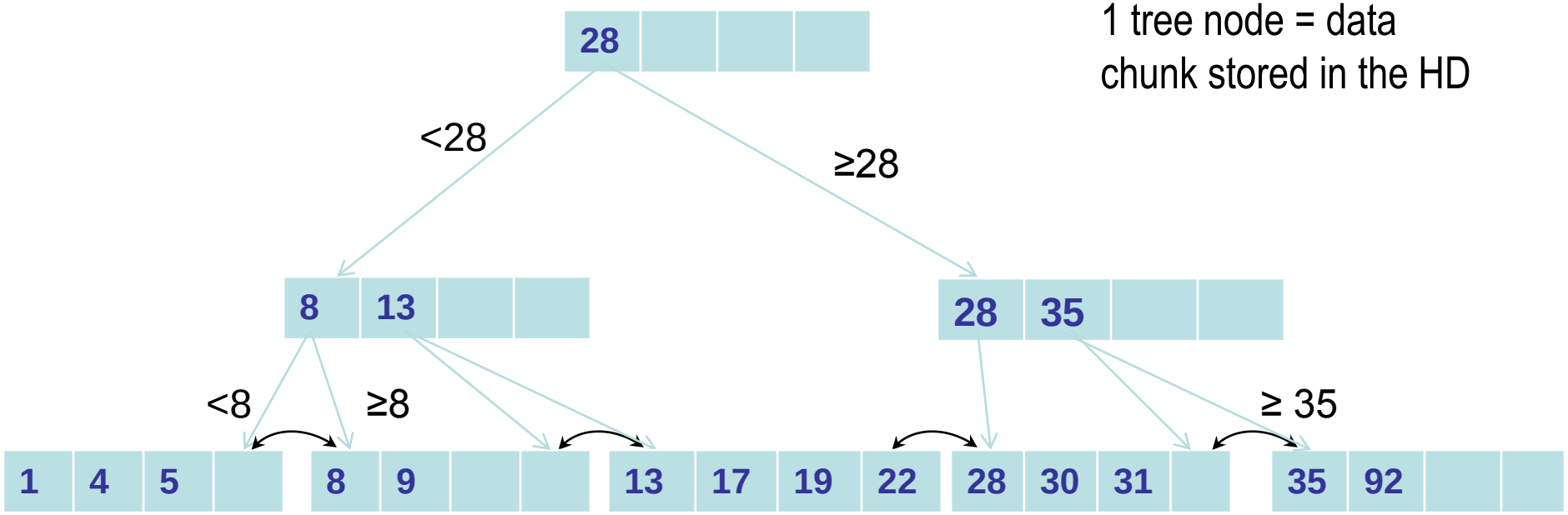


A page in a Clustered index

9567205, Tomas, Racing	25061305, Pedro, River	32102806, Cacho, San Lorenzo	…

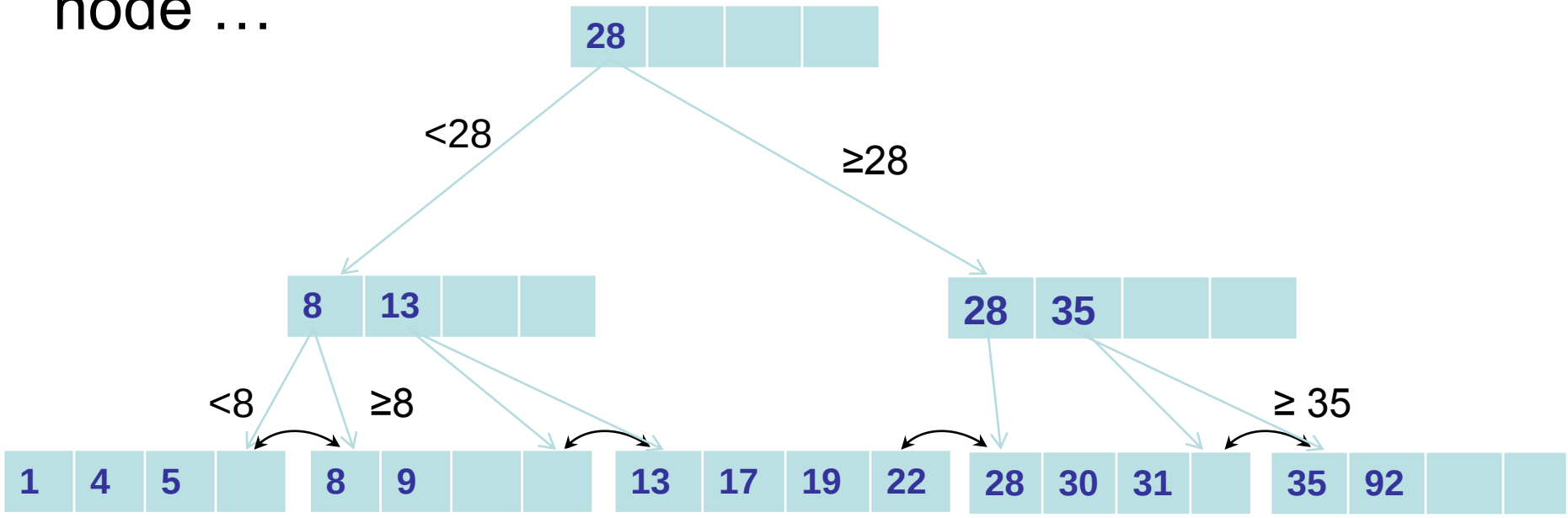
B trees are designed according to a few simple principles

- A B-tree is a tree.
- Each node can store a fixed amount of search keys.
- Search keys are followed by
  - a pointer (unclustered) or ,
  - the data (clustered) as in the past example.
- Each node must be at least half full.



Searching and Inserting keys is easy!

- In a key search, we start from the root node and follow branches.
- Inserts to non-full nodes are likewise easy.
- Keys that fall in a full node require creating a new node …



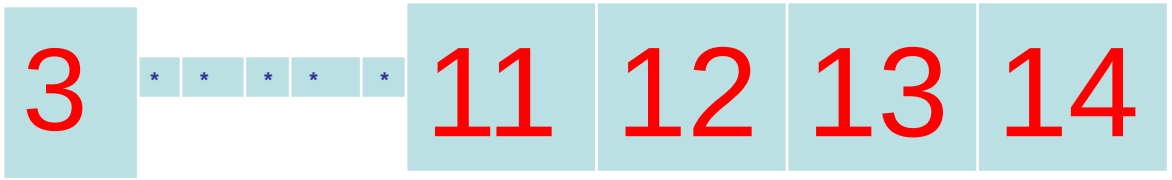
Let's fill a node with keys

- This node holds keys 4,6,7,9 and 10.
- It can fit 10 keys at the most.
- What happens if a user inserts "3,11,…,15"?

The node will fill and a new node will be created

- This node holds keys 4,6,7,9 and 10.
- It can fit 10 keys at the most.
- What happens if a user inserts “3,11,…,15”?

The user *learns* there are 5 keys between 3 and 11



node status before inserting 15

- On inserting the key 15, there is a "**node split"** and a new node is created.
- This insertion will take more time than other insertions (in the average).
- An *inspired* user can deduce that there are 5 keys between 3 and 11!!!!!
- If the user has more information about the particular B-tree implementation, he can guess what is the new nodes configuration.

This means information leaks!

- We have that:
  - The ability to *make* *inserts on an indexed field* and *detect node splits*,
  - Allows an attacker to *learn if the keys interval [a,b] is empty*;
  - Plus, learn some info about the new node configuration.

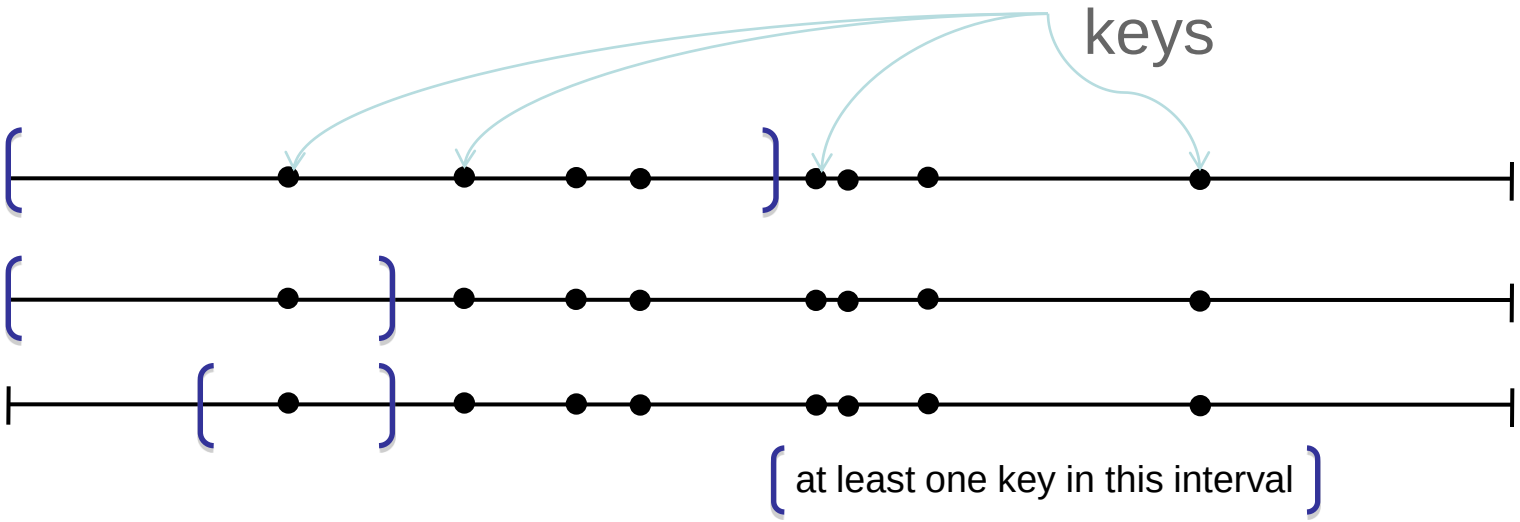
- Why?
  - Assume that *n* keys fit in one node.
  - Insert the keys *a, b, b+1, …* until there is a node split.
  - If we stopped before inserting b+n-1, then there must exist *some keys* between a and b!

- Also, reinserting a primary key produces an error.

CONSTRUCTING AN ATTACK

We use the information leak we discovered

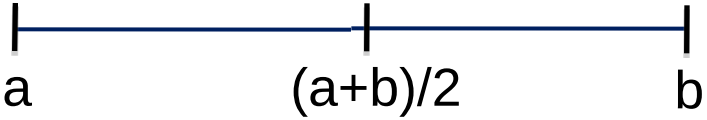
- We make a binary search until the interval size is smaller than the page size.



We can learn whether the first half of an interval is empty

To do this we need:

1. A split detection algorithm.
2. A **binary-search algorithm** that, given an interval [a,b] containing at least a key, determines whether [*a, (a+b)/2*] contains a key (else [*(a+b)/2, b*] contains a key).



Doing this *in practice* turns to be difficult.

We must estimate the cost of an attack to see if it's worth the effort

- The estimated cost is approx. =
  (No. of calls to the binary search algo.) X (No. of inserts per call to this algo.)

- For a credit cards database we have
  16 decimal digits, or 56 binary digits.
  Assume each node contains $n=512=2^9$ keys.

- We need to make 47 X 512 + 512 inserts
  Because 47 = 56 – 9

An upper bound of 24573 inserts is fast enough!

IN PRACTICE: ATTACKING MYSQL-INNODB

We have two jobs:

-detecting node splits, and

-designing the binary search procedure.

Scenario

- Server
  - MySQL was hosted in a VMware in a Pentium 4 1G server running Windows XP.
- The attacker
  - He is connected to the server through a switch.
  - He is only allowed to make inserts, and time them.
- The DbMS
  - Clean install of MySQL-InnoDB, default configuration.
  - Populate the database with different data types and table sizes.
- Noise
  - There are other users in the net, but none connects to MySQL.
  - The web server might run other services.

Our analysis starts with node splits

- In InnoDB, indexes are stored in a B+-tree structure, with some *ad hoc* optimizations.
- After a node is created, keys reordering depends on the last few inserts.
- When making consecutive inserts it has a special behavior (recall the previous example).
- Else, pages are split in halves when full.

Understanding node splits in these 3 cases allows us to construct the attack

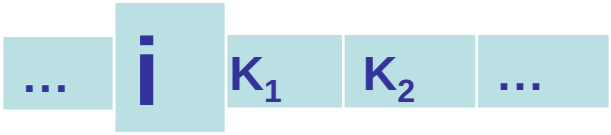
Let's imagine *that we can detect splits* for the sake of analysis.

What is the effect of inserting consecutive values i,i+1,… until there is a split?

- When i has *no value* to its right.

- When i has *one* key to its right.

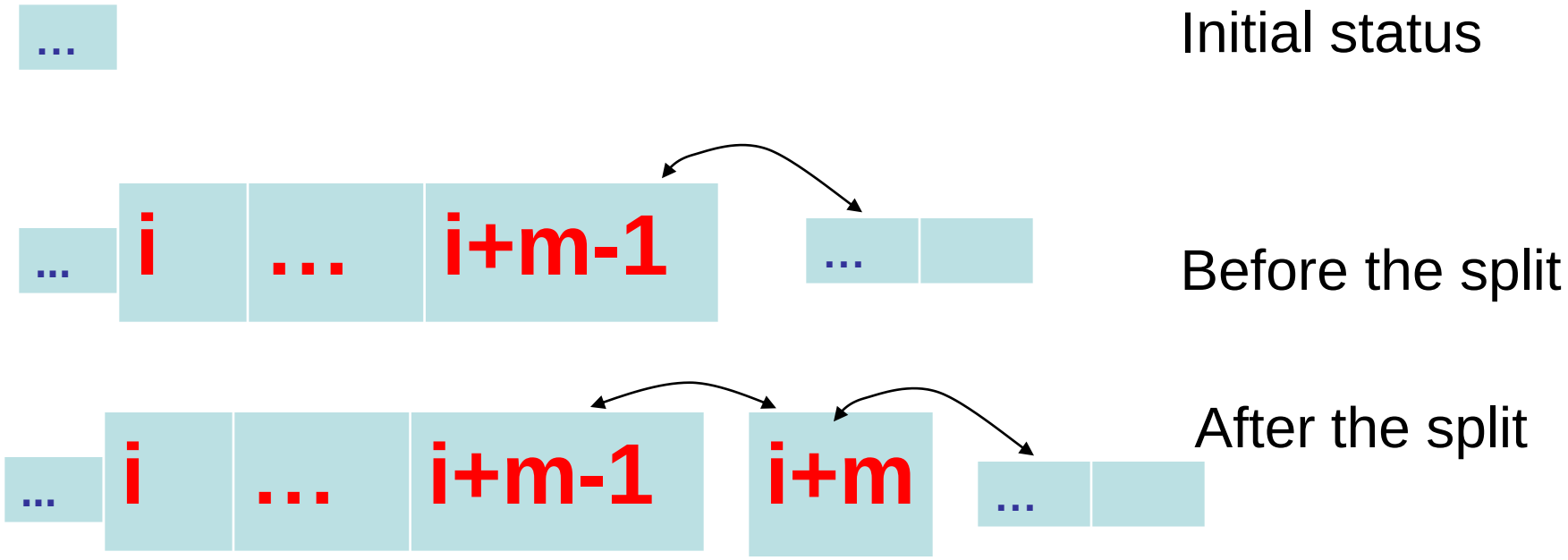
- When i has *several* keys to its right.



Case 1, when there is no value to the right of i

What is the effect of inserting consecutive values i,i+1,… until there is a split?

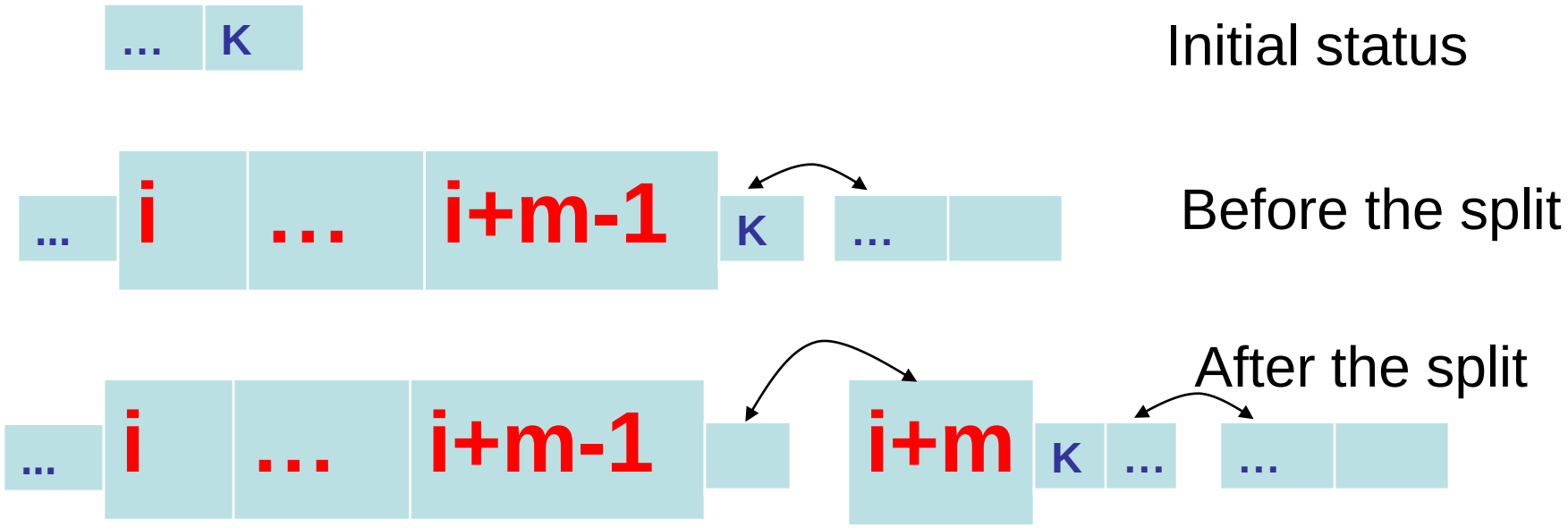
- When i has *no value* to its right.



Case 2, when there is one key to its right

What is the effect of inserting consecutive values i,i+1,… until there is a split?

- When i has *one key* to its right.



Case 3, when there are several keys to the right

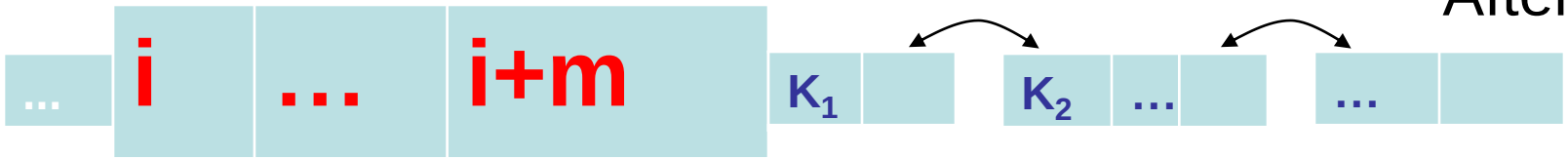
What is the effect of inserting consecutive values i,i+1,… until there is a split?

- When i has several keys to the right.

Initial status

Before the split

After the split



An algorithm outline

1. SETUP
   - Make some tricky inserts in order to produce values *a* and *b* so that a< K <b, there is no other key between *a* and *b,* and K is the first element in its page (using cases 1,2,3!)
2. BINARY SEARCH
   - We iterate over a procedure that, at each step, it halves the interval, it can tell in which half is *K*, and K is still the first element in its page.
3. FINAL STEP:
   - When the size of the interval is smaller than the page size, we check a, a+1, a+2,… until we find K.

The binary search algorithm *always* starts with this setting

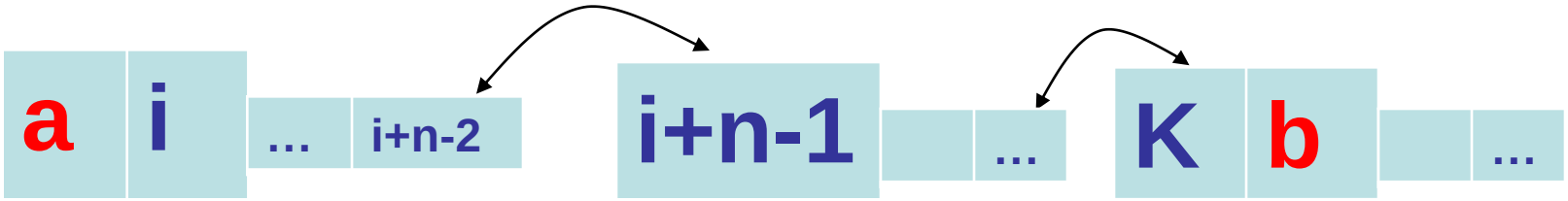
- We know values a, b such that
  - a < K < b.
  - No key exists between a and b, other than K.
  - K is the first element on its node.

- The algorithm answers K < (a+b)/2 or (a+b)/2 < K
  - the “initial situation” is maintained.

Left branch in the binary search, if (a+b)/2 < K

- What's the effect of inserting (a+b)/2, (a+b)/2+1,… until there is a split?

- If the values inserted are smaller than K, after the split the tree looks like this:

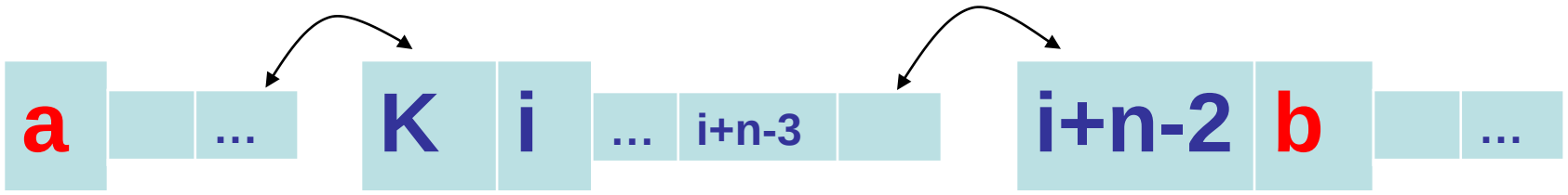


Notice that the number of values we inserted is n = the size of a node!

Right branch in the binary search, if (a+b)/2 > K

- What is the effect of inserting (a+b)/2, (a+b)/2+1,… until there is a split?

- If the values inserted are smaller than K, after the split the tree looks like this:



Notice that the number of values we inserted is n-1 = the size of a page!

This solves the design of the binary search procedure problem

- By counting the number of inserts we make until there is a split, we know if $(a+b)/2 < K$ or $(a+b)/2 > K$.

  If we inserted n values, we set

  $a := (a+b)/2 + n$.

  If we inserted n-1 values

  $b := (a+b)/2 - 1$.

- Still we need to detect splits…

Still need to detect splits. Let's assume nothing.

- On average we detect splits, but there is noise.
  - In most cases the inserts that **do not produce splits** take much **less time than inserts that produce splits**.
  - There are also indistinguishable cases.
  - In any case, there is a "time threshold value."

We define an heuristic from this experiment

- Take any table.
- Insert consecutive keys and time the response t[1], t[2],…
- For each insert, such that the values t[i-2n], t[i-n], t[i] are all bigger than the time threshold, check if they correspond to node splits.
- Yes, they do!

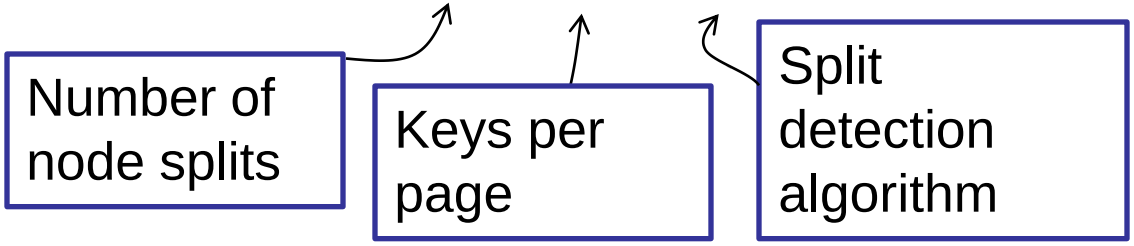
We easily deduce a split detection algorithm

- I/O:
  - INPUT: a value i.
  - OUTPUT: a value m such that there was a split at i+m.

- Remarks:
  - May need more than 3n inserts, if channel is noisy.
  - There is a tradeoff between efficiency and accuracy.
  - This is basic signal processing, and could be improved!

Experimental results corroborate our estimates

- We tested our attack
  - against three tables, with one the key 113111 plus other randomly chosen values.
  - The (theoretical) bound for the number of inserts required for the attack is 6 x 574 x 3 = 14100.



# of keys	Success rate	# of inserts	Time
1	3/3	14100	10:37
101	3/3	13145	10:39
1001	3/5	14371	10:47

Closing (mysql) summary

- We just saw that indexing by B-trees provides a side channel of information.
- We've seen how to exploit this channel, theoretically and practically.
- We went though laboratory experiments that corroborate our claims.

PRELIMINARY RESULTS ON MS SQL

An approach similar to the MySQL attack works fine.
1 slide

MS SQL indexing implements B-trees differently

- We didn't get detailed info. about the *storage engine* internals.
- We made some experiments on a clean install, standard config., with a local attacker.
- We can detect splits with good probability, but guessing new nodes configuration is difficult.
- Thus, we designed a binary search algorithm implementing a (small) "dictionary" that:
  - for consecutive splits after n[1], n[2], n[3], n[4] inserts.
  - tells us if the left half of the interval is empty.

This shows that we can search for keys in exponential speed.

COUNTERMEASURES AND ATTACK EXTENSIONS

Suggestions and discussion

Countermeasures might deteriorate efficiency

- Don’t index privacy searching data
  - E.g., what was the gain in sorting passwords in the example we saw.
- Transaction throttling: Block a user from making more than 10 inserts per day/session.
- Introduce random time delays so that the two types of inserts are indistinguishable from the time they take.
- Block certain types of behavior from your IDS.

We could extend our work by improving the attack

- Get into a more realistic scenario.
- Find better split detection algorithms.
- Research insert strategies to optimize the number of inserts that produce a split.
- Getting many keys is cheaper than getting one times number of keys.

We could extend our work to other DBMSs

- The implementation depends on DbMS!
  - We succeeded with MySQL (open code) MS SQL (closed code).
- Transactional systems, caches and journaling can play for/against the attack.
- To adapt our technique, say to other DbMSs which use B-tree indexing, one needs to:
  - Provide split detection algorithms.
  - Find a method to use the node split information leak to narrow the space for potential keys.

Thanks!

Any questions?

Or contact me at
ariel.waissbein:at:coresecurity:com

References

Futoransky, Saura, Waissbein, “Timing Attacks for Recovering Private Entries From Database Engines.” Black Hat Briefings 2007 USA, Las Vegas.

Futoransky, Saura, Waissbein “The ND2DB attack - Database content extraction using timing attacks on the indexing algorithms.” First Workshop On Offensive Technologies (WOOT). Co-located with Usenix Security 2007.

Garcia-Molina, Ullman and Widom “Database System Implementation” Prentice -Hall. 2000