

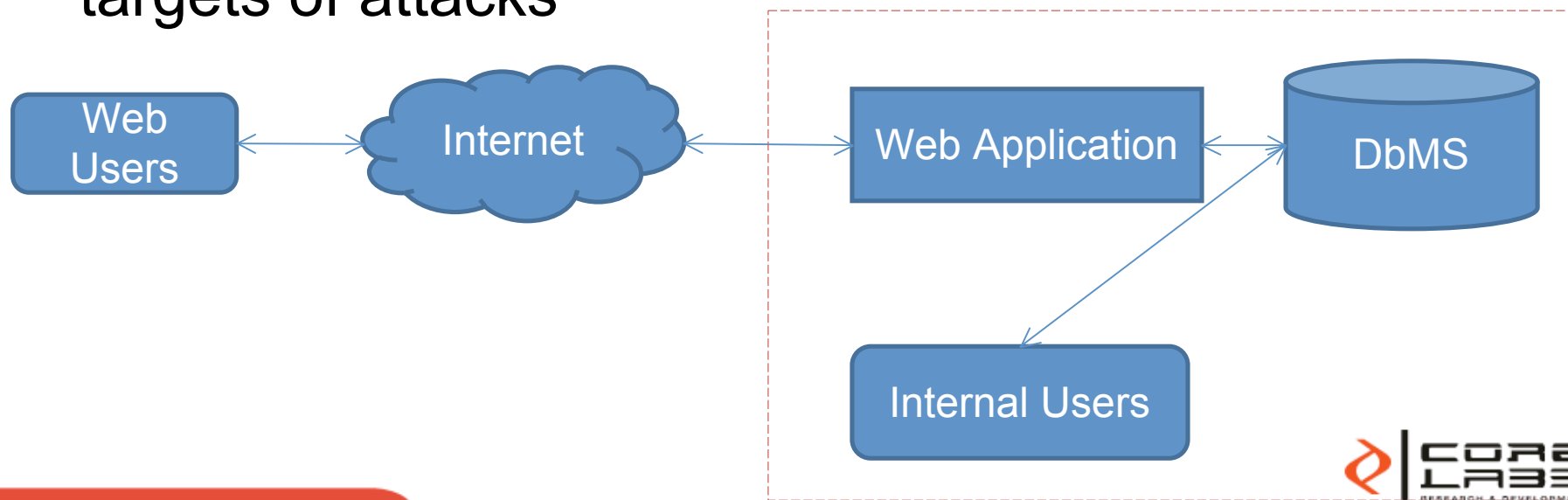
# Timing Attacks for Recovering Private Entries From Database Engines



Damian Saura, Ariel Futoransky  
and Ariel Waissbein

-Core Security Technologies-

- Database management systems are used to store huge amounts of data that need to be searched for and refreshed.
  - E.g., target credit card data, health care info., social security numbers and other personal data, ...
- So DbMSs and the servers that host them are targets of attacks



- An attacker breaks into the web server hosting the DB.
  - Insecure configuration, lack of patching, ...
- An attacker exploits a SQL-injection vulnerability in the web application (front-end of the DB).
  - Insecure development of the webapp
- An attacker leverages lax permissions and privilege levels in the DB.
  - Someone that can connect to the server, but is not a DB user, compromises an insecure authentication protocol.
  - A legitimate user siphons out confidential data.
- An attacker uses a timing side-channel that relies on the ability to make INSERTs with chosen data.

- Consider a populated 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.
- Users can measure the response time of the INSERT transaction.

- 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 “complexity” 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.

- Explicitly,
  - We designed a side-channel attack that relies only on a data structure, B-trees, that is used by most commercial DbMS and the ability to make inserts in the target field and time responses (accurately).
  - We implemented the attack in our lab against a MySQL database and proved it real.
- Further remarks,
  - What does this vulnerability imply?
  - The attack could be improved (complexity).

Indexing table columns, containing sensitive data, is dangerous.

A first example

- Imagine a Content Management System (CMS) that:
  - displays a user/password table (as below) and
  - when a user clicks on Password, the table entries are sorted according to the alphabetical order of the passwords.
- A user that is allowed to add entries to the table can then execute a divide et impera search (Latin for binary search) for any other user's password.

Username	Password
Dick	*****
Harry	*****
Tom	*****
....	



- Imagine a Content Management System (CMS) that:
  - displays a table of the form and
  - when a user clicks on Password, the table is reordered according to the alphabetical order of the passwords.
- A user that is allowed to register can then execute a divide et impera search for any other user's password.

Username	Password
Dick	*****
Harry	*****
Tom	*****
....	



Username	<u>Password</u>
Tom	*****
Dick	*****
Harry	*****
....	

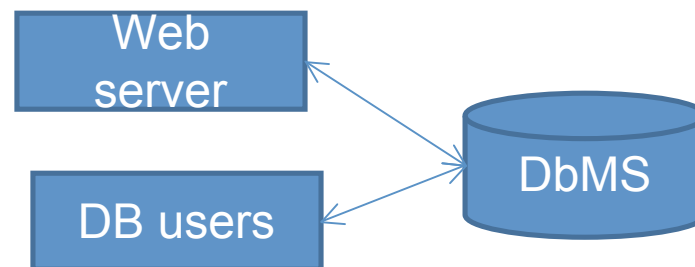
Hence *Tom's password < Dick's password*  
 There is an information leak!

1. Database management systems
2. DbMS leak information
3. An attack that exploits this leak
4. Experiments with MySQL
5. Extensions, countermeasures and discussion

# Database management systems

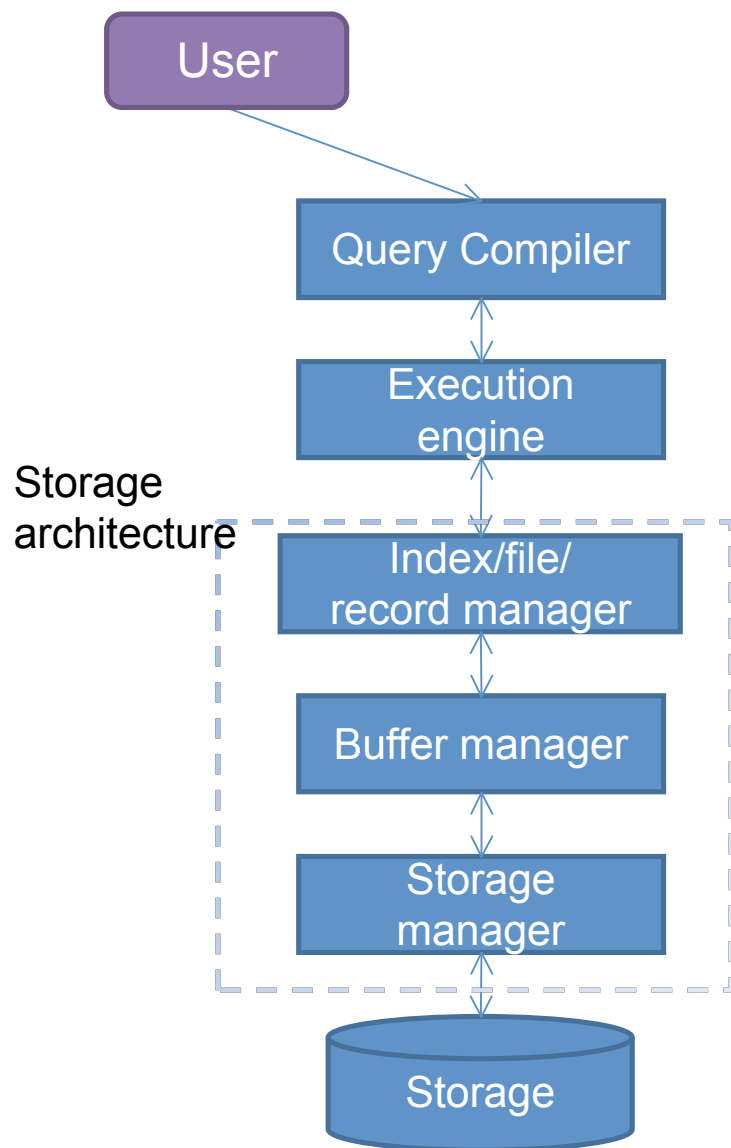
and how is indexing implemented

- Clients connect to access high volumes of data
  - Persistent storage
  - Queries / data manipulation
- Need for efficient searching, writing and deleting data
  - Programming interface.



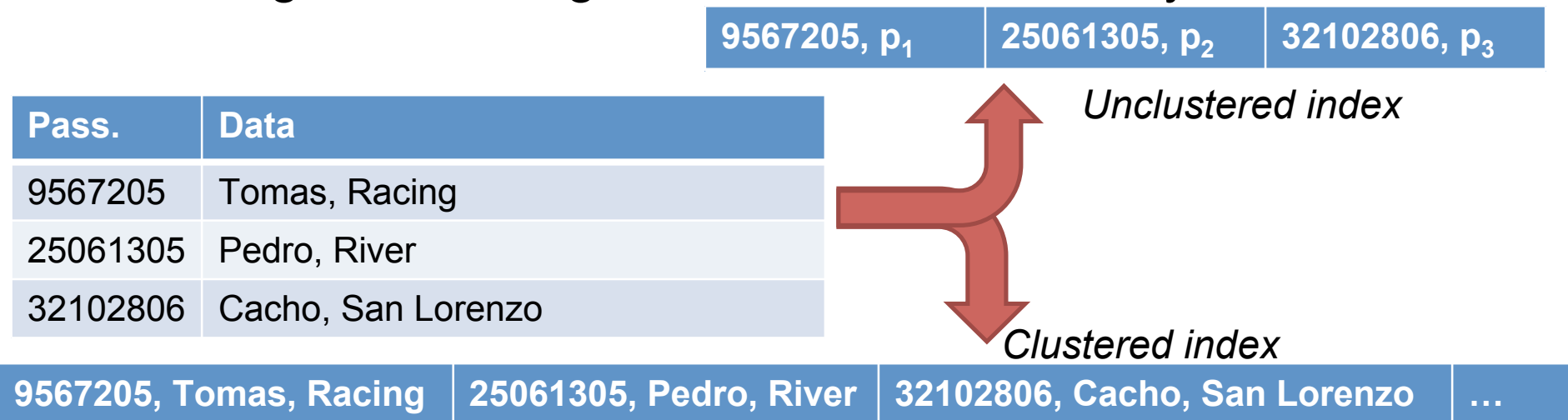
- The relational model & the SQL standard.
- Data is stored in tables: each row contains a record, and the columns represent the record fields.
- If table rows are not sorted by the values in its fields, then each search/insert/delete query (over a field) requires scanning all the column.
  - Thus, TABLES SHOULD BE SORTED!
  - In fact, updating, inserting and deleting must be optimized.
- Can't store everything in RAM. Must use the hard drive and retrieve data to memory in chunks.

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

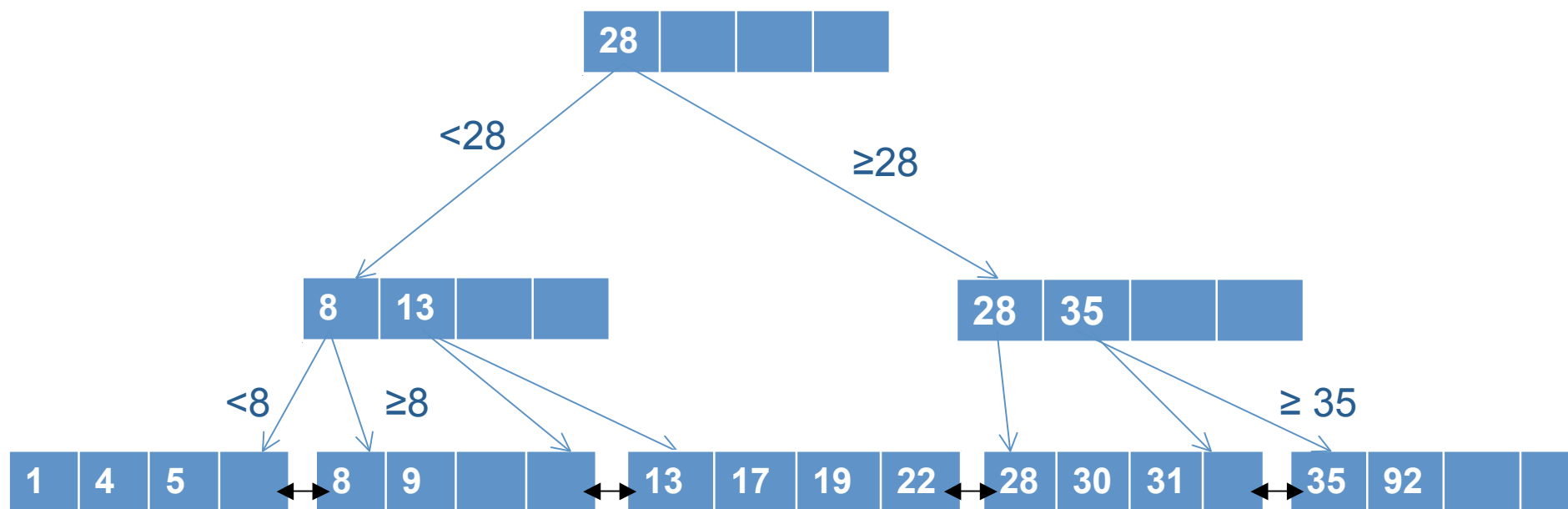


- Data is stored in “sorted chunks” (i.e., pages).
- The querying process:
  - The user makes queries.
  - To answer, the DbMS retrieves only the required pages from *Storage* into memory.
  - The cost of page I/O dominates the cost of typical DB operations.
- To understand more deeply how this *cost* is affected by queries, we must analyze indexes.

- Each DB table requires one primary index
  - It can be generated automatically by the DbMS, or according to a user-selected search key (e.g., a field).
- Each index produces an (internal) table that is stored by the DbMS in an **index data structure** (e.g., B-trees):
  - Storing each search-key together with a pointer to the data (row), or
  - Storing the data together with the search key.

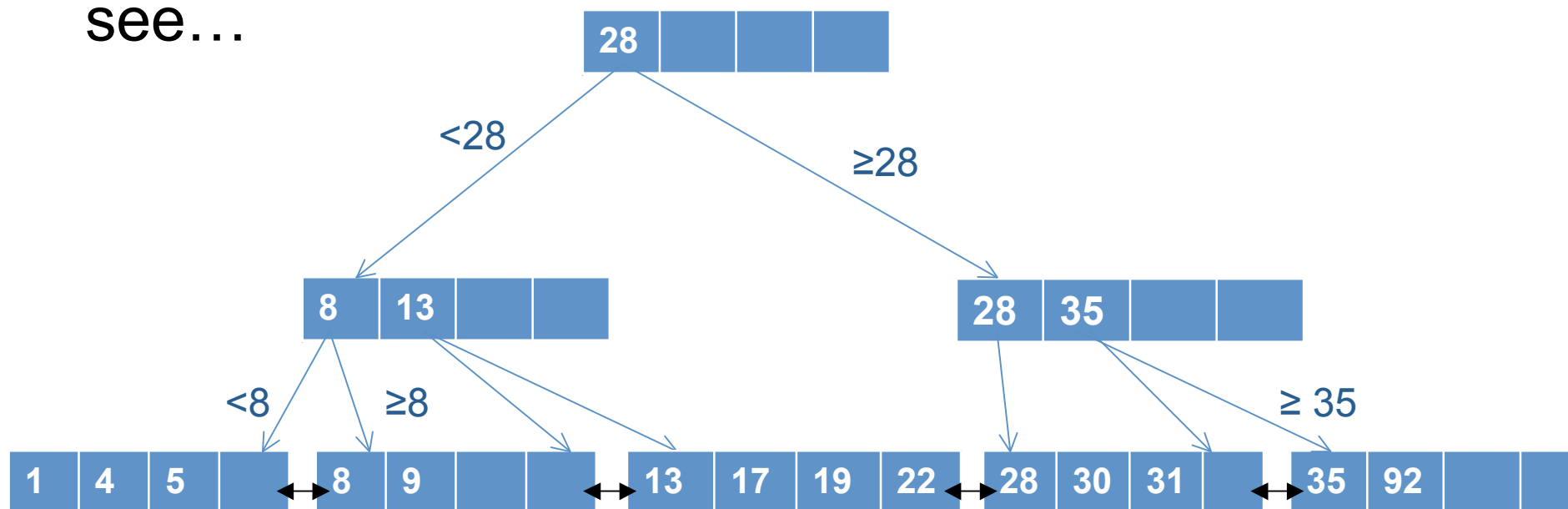


- Each node can store at most a prefixed amount of search keys (and occupies one disk page in Storage).
- Each node must be at least half full.
- Each search key is paired with a pointer or the data.
- Leaf nodes (lower level) are linked in a list (black arrows below).

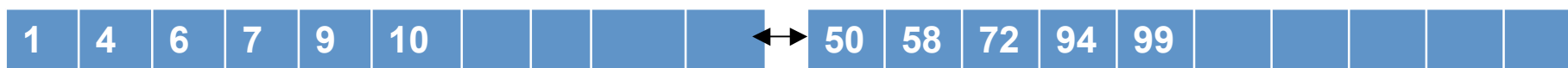




- Looking up a search-key value or range is easy, we start from the root node and move down as in the picture below.
- Inserts to non-full nodes are likewise easy.
- Operations that require adding/deleting nodes: let's see...

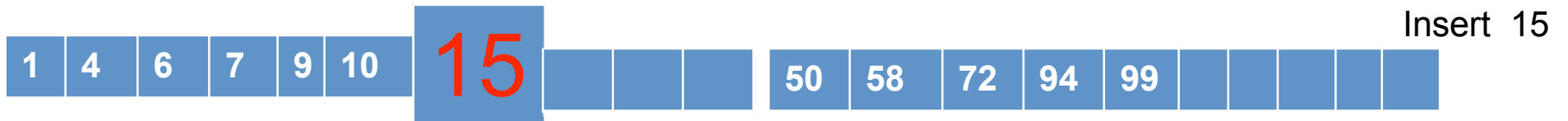


## (TOY EXAMPLES)

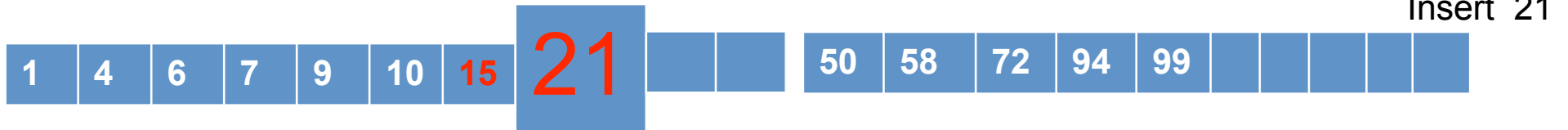
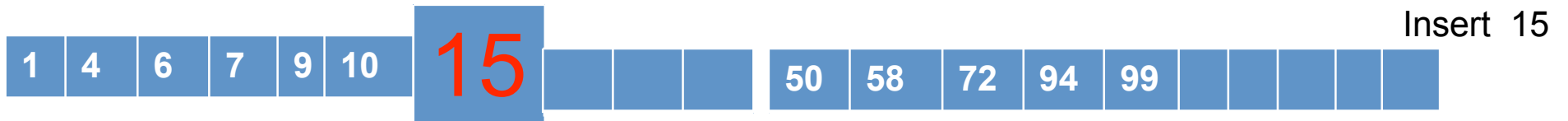


- Let's picture two consecutive leaf nodes.
- We start adding random values until the left leaf is full.

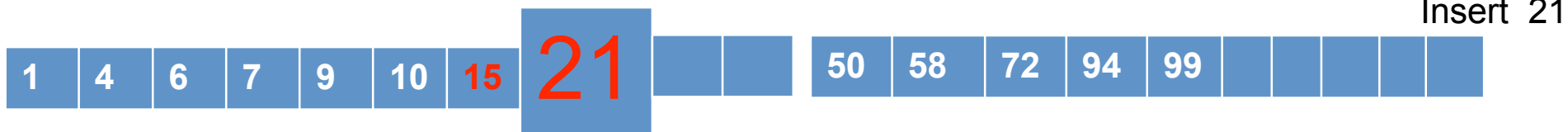
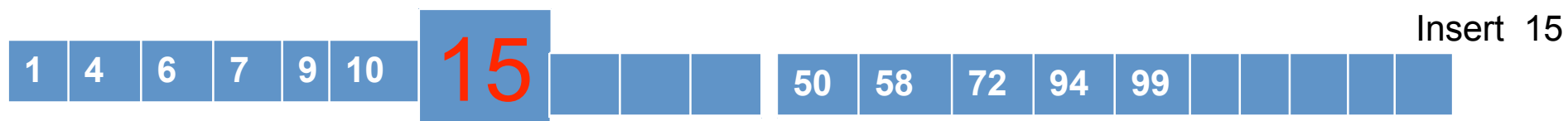
# The effect of inserts (2)



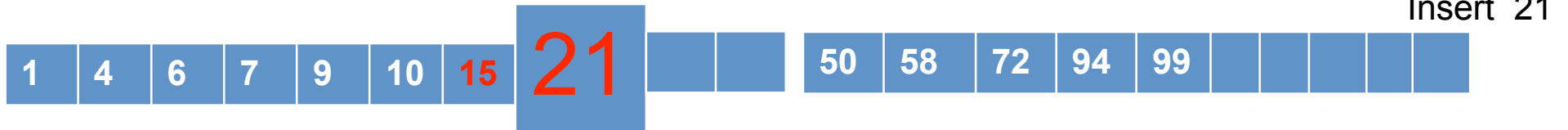
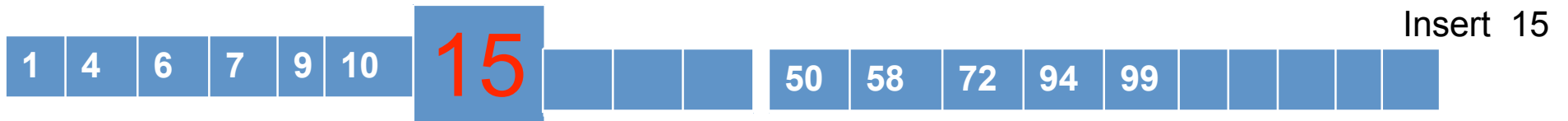
# The effect of inserts (2)



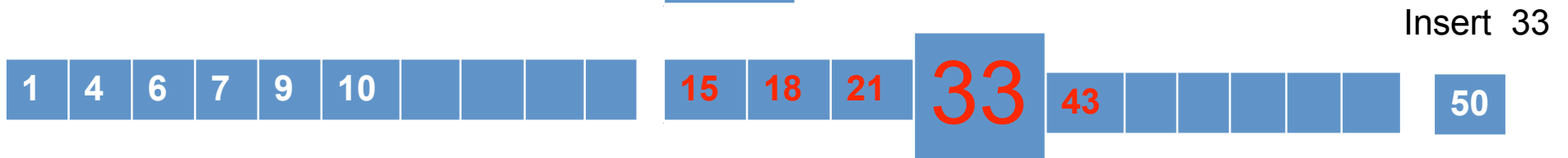
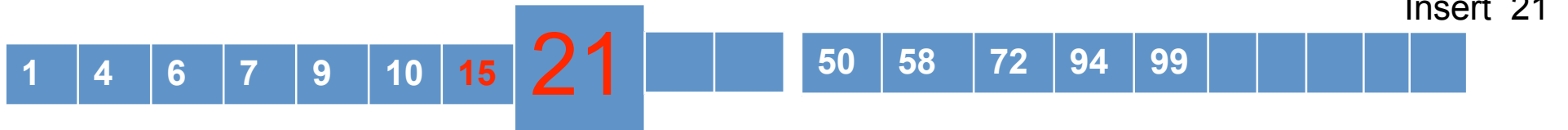
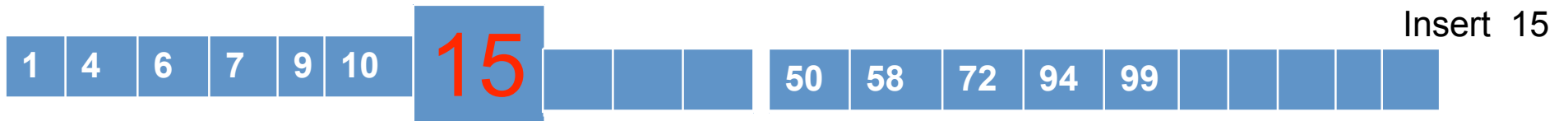
# The effect of inserts (2)



# The effect of inserts (2)



# The effect of inserts (2)



- Once the left node is full, it is *split* in two.
- Remember: each node must be at least half full.
- An insert that produces a split takes more time than other inserts!





# How to turn the information leak into an attack

E.g., can we use split detection to find key values?

- Each line represents a leaf, that can fit 10 search keys.
- Previous inserts are in white, the attacker's inserts in red.
- What happens if a user knows the leaf starts at 3, the next leaf starts at 25 and inserts "11,...,16"?



- Each line represents a leaf, that can fit 10 search keys.
- Previous inserts are in white, the attacker's inserts in red.
- What happens if a user knows the leaf starts at 3, the next leaf starts at 25 and inserts "11,...,16"?



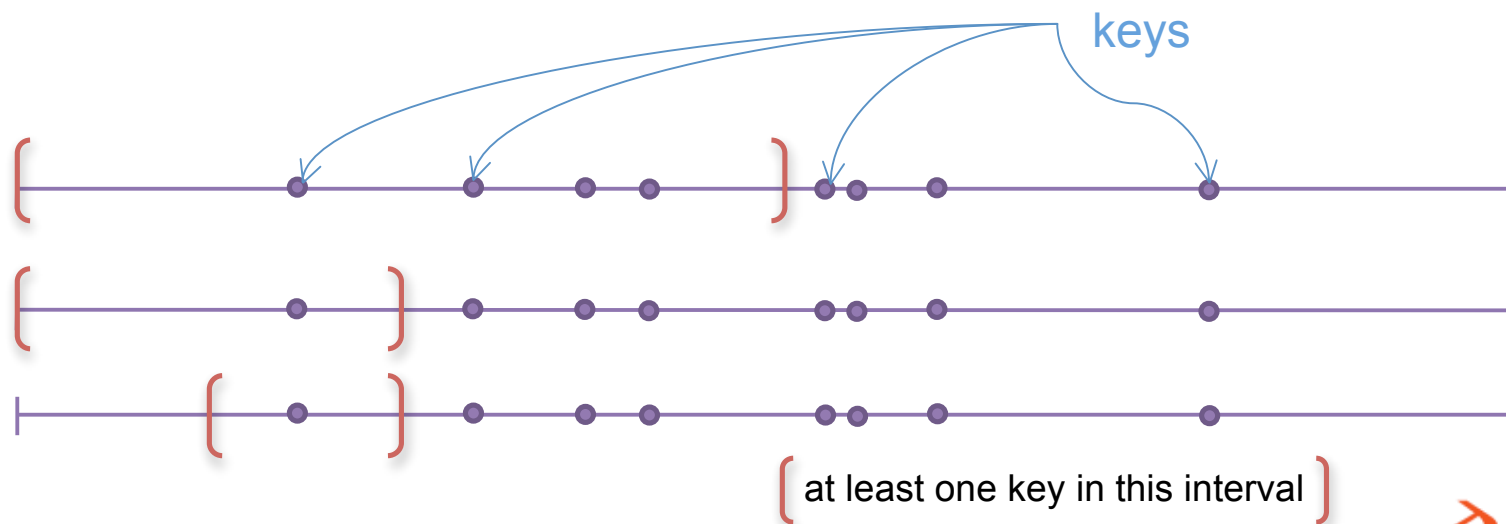


*leaf status before  
inserting 16*

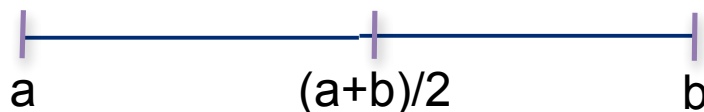
- The user inserts 11-16 and knows nothing about the pre-existent keys (other than 3).
- Assume that he knows that “16” produced a split!
- Then, he knows that there are 4 keys between 3 and 11!
- If the user has more information about the particular B+-tree implementation, he can guess what is the new leaves configuration.
  - This is because, some DbMSs use an optimization of B+-trees and will not split leaves in halves in certain cases.

- We have that:
  - If we have the ability to *make inserts on an indexed field and detect node splits*,
  - Then, given an two search keys  $a, b$  on the same node, we can tell whether there is at least one key between them; plus, learn some info about the new node configuration.
- Why?
  - Assume that  $n$  keys fit in one node and  $n$  is known.
  - Insert the keys  $b+1, \dots$  until there is a node split.
  - If we stopped before inserting  $b+n-1$ , then there must exist keys between  $a$  and  $b$ !
- Also, since *primary keys* are not allowed to repeat:
  - if we attempt to insert a key with an already existing value we will receive an error –and therefore learn the value of this older key!

- At each step, we divide an interval in two halves, if the first half contains one key, we continue with this.
- When the interval is smaller than the page size, we test all its keys.



- In order to design the attack we need to
  - Develop a split detection algorithm
  - Develop 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).



- Let's say we are attacking a credit cards database
  - We start with 0 and  $10^{17}-1$  that includes all the (16 decimal digits, or 56 binary digits) credit cards.
  - Assume that each page disk contains  $n=512=2^9$  keys.



- We need to invoke  $\approx 46=(56-10)$  times the binary-search algorithm, each invocation requiring  $<512$  inserts, plus the search in the last step. This amounts to an upper bound of 11500 inserts.

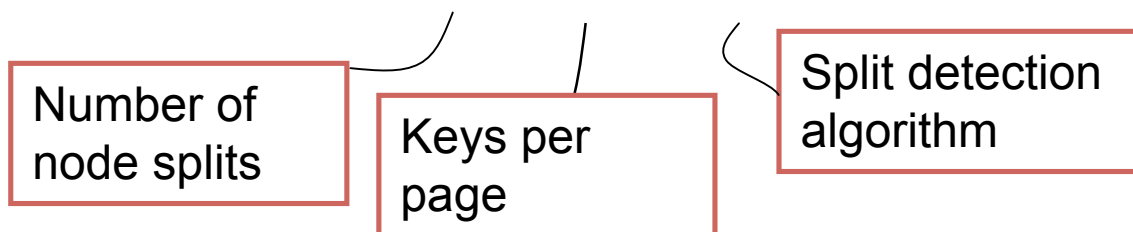


# Attacking MySQL-InnoDB

1. Scenario and Results
2. Attack details
  - a) How splitting works in InnoDB
  - b) The attack algorithm
  - c) Node Split detection algorithm
3. Statistics

- MySQL is an open-source and very popular DbMS.
- InnoDB is one of the storage engines that come with MySQL
  - It requires a clustered index and uses a B+-tree structure for indexes.
- The DbMS
  - Clean install of MySQL-InnoDB
  - Populate the database with different data types and table sizes
  - Connect as a MySQL user through an Intranet (i.e., one switch)
  - Only allowed to make inserts.
- Noise
  - There are other users in the net
  - No other users connecting to MySQL.
  - The web server might run other services.

- We tested our attack
  - against three tables, with one key 113111 plus other uniformly chosen values between 0 and 10M.
  - The (theoretic) estimate for the number of inserts required for the attack is  $6 \times 574 \times 3 = 14100$ .



# 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

## Page splitting in InnoDB

- We need to understand page splits under InnoDB,
  - Indexes are stored in a B+-tree structure, with some *ad hoc* optimizations.
  - The restructuring of the tree after a node addition depends on the last few inserts.
  - When making consecutive inserts it has a special behavior.
  - Else, pages are split in halves when full.

## InnoDB and B+-trees

- We analyze for a non-full node



what is the effect of inserting consecutive values  $i, i+1, \dots$  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.

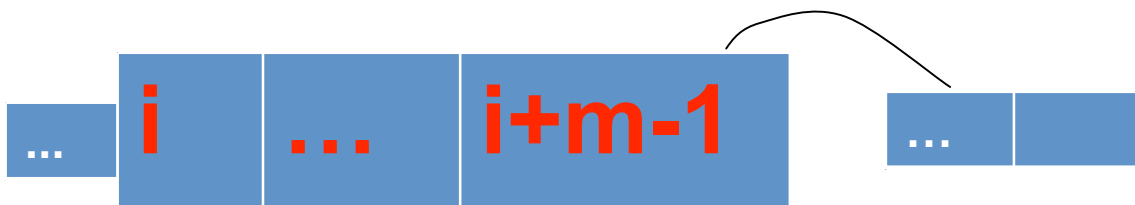


## Case 1

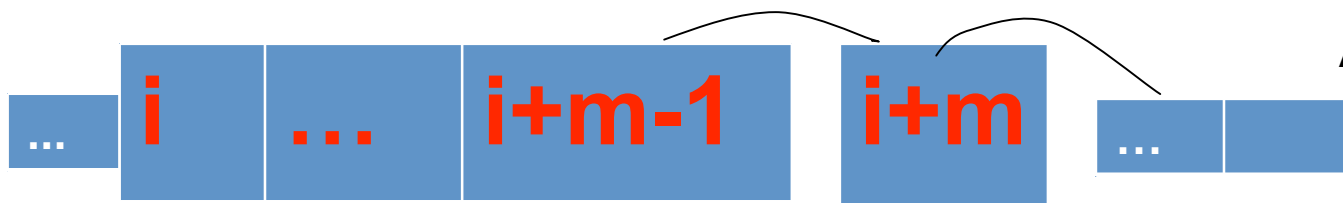
- What is the effect of inserting consecutive values  $i, i+1, \dots$  until there is a split?
  - When  $i$  has *no value* to its right.

...

Initial status



Before the split



After the split

## Case 2

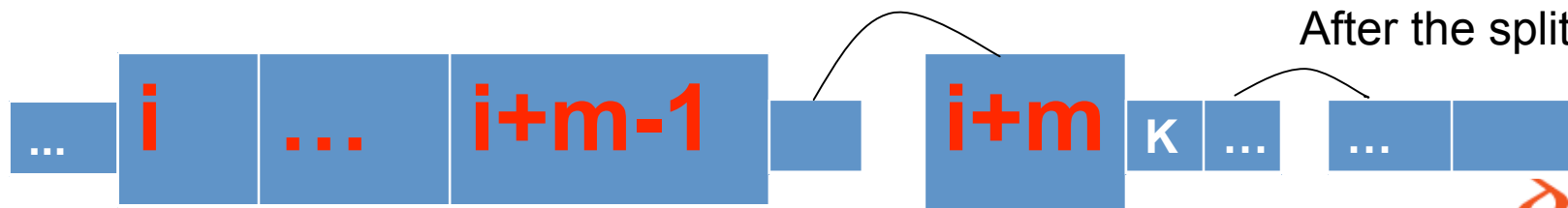
- What is the effect of inserting consecutive values  $i, i+1, \dots$  until there is a split?
  - When  $i$  has one key to its right.



Initial status



Before the split



After the split

## Case 3

- What is the effect of inserting consecutive values  $i, i+1, \dots$  until there is a split?
  - When  $i$  has several keys to its right.





## How to retrieve a secret key $K$

### 1. SETUP

- We insert certain values so that: we get values  $a$  and  $b$  such that  $a < K < b$ , there is no other key between  $a$  and  $b$ , and  $K$  is the first element in its page.



### 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  $a, a+1, a+2, \dots$  until we find  $K$ .

## The binary search algorithm

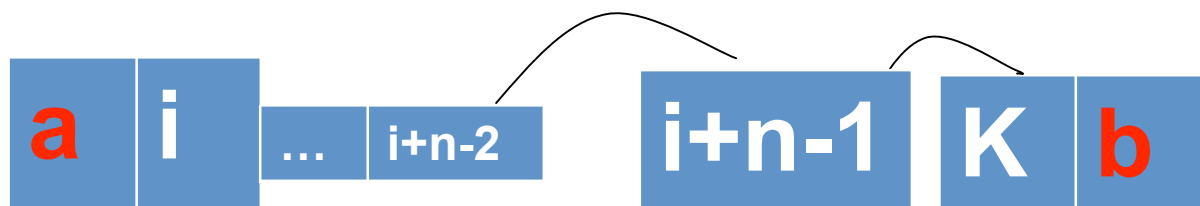
- As input we have values  $a$ ,  $b$  such that
  - $a < K < b$ , where  $a$  and  $b$  are known and  $K$  is unknown.
  - There is no value other than  $K$  between  $a$  and  $b$ .
  - $K$  is the first element on its page



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

## The binary search algorithm

- If all the values inserted are smaller than  $K$  the state of the tree after the split would be

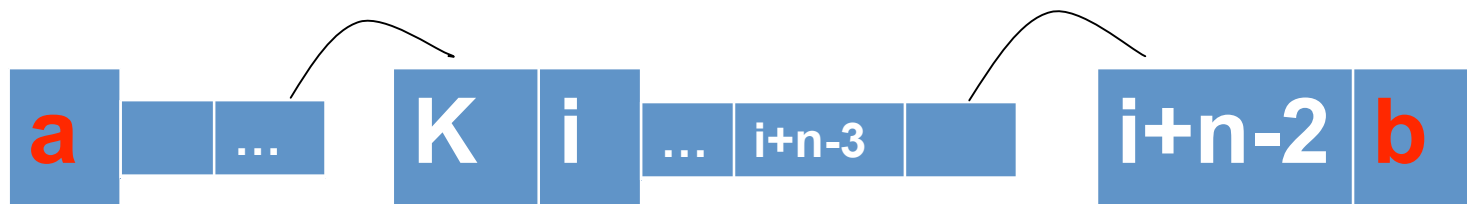


(here  $i = (a+b)/2$ .)

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

## The binary search algorithm

- If all the values inserted are greater than  $K$  the state of the tree after the split would be



Notice that the number of values we inserted is  $n-1$

This assumes that the leaf on the right contained no other values than  $K, b$ . Else the split occurs before the  $(n-1)$ -th insert.

## The binary search algorithm

- By looking at the number of values we insert until there is a split, we know if  $(a+b)/2 < K$  or  $(a+b)/2 > K$ , so we can shorten the original interval  $[a,b]$  in half as follows

if we inserted  $n$  values, we set

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

if we inserted  $n-1$  values

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

- So repeating this procedure we get that the search of  $K$  is done at an exponential speed!



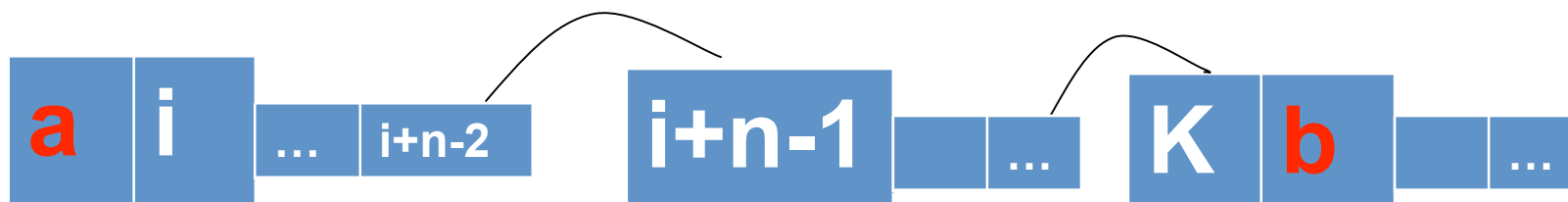
- About noise:
  - In most cases the inserts that do not produce splits take much less time than inserts that produce splits.
  - But, there are many indistinguishable cases.
  - In any case, there is a “time threshold value.”
  - Timing with functions `QueryPerformanceCounter` and `QueryPerformanceFrequency` in `kernel32.dll`
- An experiment
  - we insert consecutive values and time them  $t[1], t[2], \dots$
  - For each  $i$ , such that the values  $t[i], t[i+n], t[i+2n]$  are all bigger than the time threshold, we check whether they correspond to node splits (Case 1).
  - Yes, it is improbable that  $t[i], t[i+n], t[i+2n] > \text{threshold}$  and no split occurred.

- The previous experiment can be translated into a split detection algorithm.
  - We need a table (e.g.,  $(i, i+n, i+2n) \Rightarrow$  Case 1,  $(i, i+n-1, i+2n-1) \Rightarrow$  Case 2, etcetera).
- INPUT: a value  $i$ .
- OUTPUT: left node or right node.
- Remarks:
  - the algorithm is probabilistic.
  - it may need to make more than  $2n$  inserts.
  - This is basic signal processing, and could be improved!



- We need to piece together the split detection and binary search algorithms, and show that this produces the expected result.
- Let's return to the cases  $(a+b)/2 < K$  and  $K < (a+b)/2$

- First, when  $(a+b)/2 < K$

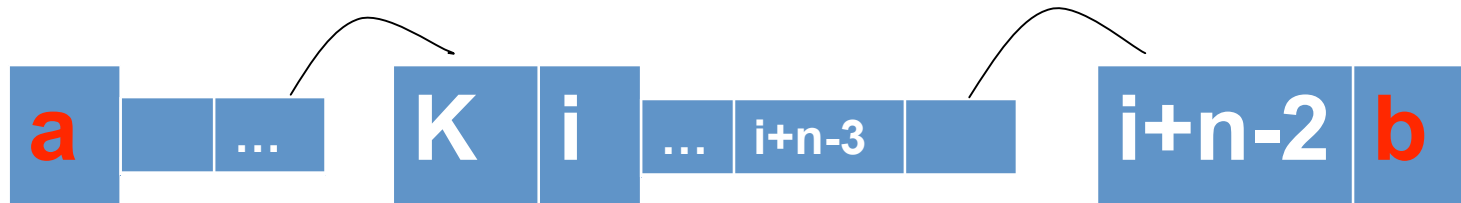


In this case, if we insert  $i, i+1 \dots$  and eventually stop when we detect a split, e.g., at  $(i+n-1, i+2n-1, i+3n-1)$ , then notice that:

- Node splits correspond to cases 1,1 and 1.
- $i+3n-1 < K < b$ , and there is no key between  $i+3n-1$  and  $b$ .
- $K$  remains the first element in a node.

So we take  $a := i+3n-1$  and continue with the binary search.

- Second, when  $K < (a+b)/2$



In this case, if we insert  $i, i+1 \dots$  and eventually stop when we detect a split, e.g., at  $(i+n-2, i+2n-3, i+3n-4)$ , then notice that:

- Node splits correspond to cases 1, 2 and 2.
- $a < K < i$ , and there is no key between  $i+2(n-1)-1$  and  $b$ .
- $K$  remains the first element in a node

So we take  $b:=i$  and continue with the binary search.

- Similarly, the setup procedure can be combined with this split detection algorithm.
- The number of inserts required to execute the attack is multiplied by 3 (we expect!).
  - This is nothing if we consider that the speed of the search is logarithmic (e.g.,  $3 \cdot \log(N) \ll N$ )

# Future work and coutermeasures

- How to improve our attack
  - Can we get outside the lab?
  - Better split detection through signal processing.
  - Require less inserts in order to produce one split.
  - Heuristic optimizations: E.g., if the values are assumed to be uniformly distributed, then we can replace the binary search for a more general divide-and-conquer.
  - Optimize the attack for getting many keys.

- Other DbMSs require a lot of work!
  - Varies depending on DbMS implementation details.
  - 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.

- Don't index privacy searching data: then every query lasts the same amount of time!
- Transaction throttling: Block a user from making more than 10 inserts per day/session.
- Blinding at the DbMS: encode the search-key values.
- Introduce random time delays so that the two types of inserts are indistinguishable from the time they take.
- NIDS: Block certain types of behavior.



Thanks!

Any questions?