The ND2DB attack: Database content extraction using timing attacks on the indexing algorithms

Ariel Futoransky   Damián Saura   Ariel Waissbein *

July 31, 2007

Abstract

In this paper we present a new attack technique that allows extraction of selected database content relying merely on the attacker's ability to perform database transactions (INSERTs or UPDATEs) that are usually available to any anonymous database user. Our attack technique uses a side-channel timing attack in the realm of database indexing algorithms and data structures. We prove that by exploiting the inherent characteristics of the most commonly used indexing data structures and algorithms in today's commercial database management systems it is possible to extract privacy-sensitive data from a database. In particular we prove, both in theory and practice that it is feasible to do so if the B-tree data structure is used and the attacker is able to insert records with chosen data that is used as the search key of one of the table's indexes. We present experimental results of a successful attack implementation against MySQL and provide conclusions and ideas for further research.

1 Introduction

Database management systems (DbMS), the data stored in them and the applications used to populate, manage and retrieve it constitute a major concern for security-aware organizations seeking to minimize their exposure to information security risks. In the past years, a number of high-profile security breaches, including incidents in which privacy sensitive information was disclosed, were reported. Consequently, the perils of enforcing effective database security have become more evident and database security awareness has increased. The consensus is that the majority of incidents involving disclosure or abuse of privacy-sensitive data stored in databases is caused by miss-configuration of database security mechanisms, exploitation of software implementation flaws (bugs) in the applications used to insert or retrieve data from the database system —such as SQL injection vulnerabilities ([15])— or security policy violations from trusted database users (cf. [16]).

In this paper we present a new attack technique with which an attacker is capable of extracting privacy-sensitive data stored in a database system —such as Credit Card numbers, social security numbers, authentication credentials or PINs— using only the capabilities generally available to anonymous (untrusted) users of database applications that execute only a minimal set of database operations. Namely, the characterization of our attacker will only require that he is able to perform an arbitrary number of INSERT (or alternatively UPDATE) operations (see [13]) on a table with chosen contents for the field (table column) values of the records (table rows) to be inserted. For example, in a typical attack scenario an attacker with the ability to add records with his own credit card numbers would be able to retrieve valid credit card numbers that belong to the records of other persons in the database.

The attack technique is based on the application

*Core Security Technologies. Humboldt 1967, 2nd floor. Cda. de Buenos Aires, Argentina. The authors wish to express their gratitude to Ivan Arce for his active interest in the publication of this paper and the anonymous referees for many helpful comments.

of timing attacks to the algorithm used to insert new search keys in a B-tree (and variations) —which is the most common data structure used to implement table indexes in current DbMS ([10]). In order to execute our attack against a given table field, we require that this field is indexed. Indexing has been always considered in terms of efficiency, and never before in terms of security. Our result, shows that indexing has security implications.

Although timing attacks were first applied to cryptanalysis (cf. [11], [4], [6] and [7]) today they are also commonly used to exploit a common type implementation flaw of web applications: Blind SQL-injection vulnerabilities ([1], [12]). In this later case, the attacker relies on his ability to amplify a measurable timing difference between successful and unsuccessful exploitation attempts of the bug (SQL-injection vulnerability) when the output of the injected SQL statement is not visible. The attacker must then be able to create and inject a SQL statement that implements a measurable side-channel on the target system ([1]). On the other hand, our attack technique does not rely on the ability to introduce a new side-channel vulnerability to the database system and does not require the existence of a software bug in the database or its client application. Instead, the attack leverages an existent side-channel vulnerability that is inherent to the use of B-trees for storing indexes: There is a measurable timing difference between those search key insertion operations that require B-tree node splitting and tree re-balancing and those that do not. We prove that this leak provides the attacker with enough information to feasibly derive search keys that were already present in the index before the attack started. Specifically, we will describe a general methodology that allows to exploit these information leaks and compute the search keys of a table. Moreover, we demonstrate this attack against a MySQL DbMS used with the InnoDB storage engine and reproduce statistics that confirm our claims. For example, our attack retrieved the first key of a table of 64-bit integers in about 10 minutes after making in the order of 10 thousand inserts.

Two earlier works ([9] and [5]) introduce side-channel timing attacks against different layers of a web application: web servers and web browsers. Briefly speaking, these timing attacks profit from optimizations at the side of the client (mainly caching) to recover web-browsing histories ([9]) and other private data ([5]) and optimizations at the server side to determine certain properties of the private data they host (*op. cit.*), e.g., knowing whether a given username is valid or estimating the size of private data.

The rest of this paper is structured as follows: In the next section we describe our attack technique. We first present databases, tables, indexes, the B-tree data structure and the search key insertion algorithm. Next, in Section 2.2 we describe our attack technique in s generic manner that is applicable to any database that uses the B-tree indexing. Section 3 describes the implementation of the attack against a MySQL database with the InnoDB store engine, where we first give some clues that help in tuning the attack for different scenarios (Section 3.1), we then describe our split detection method (Section 3.2) and finally, in Section 3.4, we present experimental results of performing our attack against the MySQL-InnoDB pair. We end this short article with conclusions and ideas for future work.

2 The attack technique

Our attack recovers search-key values from a given nonempty table. In many cases, such as with MySQL-InnoDB, the search-key value and data value are the same. In any case, (e.g., from the cryptographic standpoint) the search-key value leaks information about the data itself.

Given a value x_0 in the range of the search-key values under attack, our attack technique allows you to recover the smallest search-key value y that is bigger than x_0 and requires making $O(n \log(y - x_0))$ inserts, where n is the block length of this index structure (described below), e.g., in MySQL we have $n \leq 600$. If all the values are less than k bits long, then this quantity can be estimated by $O(nk)$. In order to recover a second key y_2 from the table, we can launch a first attack, call this $y_1 := y$, and then start a new attack with $x_0 := y + 1$. This only reuses the attack as is, although optimizations are surely possible. More generally, we can estimate the effort required to re-

cover s keys from the table by $O(nsk)$.

Central to our attack is the fact that some private columns of a table are indexed (e.g., with B-trees), an external user which is able to insert new search-key values and detect node splits, can therefore estimate the values of the search keys. In order to understand this, we follow to explore database internals and proceed describing a generic attack technique for any B-tree indexing algorithms.

2.1 The data structure and functioning of DbMS

Database Management Systems (DbMS), such as the Oracle, Microsoft SQL Server, MySQL, Postgres, DB2, Microsoft Access, and FileMaker implementations, assist with high-volume and heterogeneous data storage, retrieval and organization.

DbMSs store collections of tables. A table has bidimensional structure, given by a predefined number of columns and an arbitrary number of rows. Each record is stored in a row and the record fields are divided in columns, e.g., “name,” “age,” etcetera.

A naive search over a table, say for all the records with a given field value, would require scanning all the table rows —which is inadmissible in real-sized applications. In order to make data retrieval, insertion and deletion efficient, DbMSs are configured to sort some of the columns of a table using a data structure called indexes. That is, each record receives a unique identifier and for each index they build a two-columns table, e.g., a sorted version of the indexed table column, where each of these values is paired with a pointer to the record to which it belongs.

B-trees, B+-tree and other variants (see [2], [3], [10]) are the most popular choice for indexing. In B-trees, data is organized in blocks and these blocks in a balanced tree. The tree is said to be balanced because each path from the root of the tree to any leaf has the same length. Each node contains at most n search-key values, for a fixed integer n that is called the block length and whose value is determined by the DbMS (*op. cit.* and [14, Section 14.2.13]). At the root, except in a border case, there are at least two pointers: one pointing to each block below.

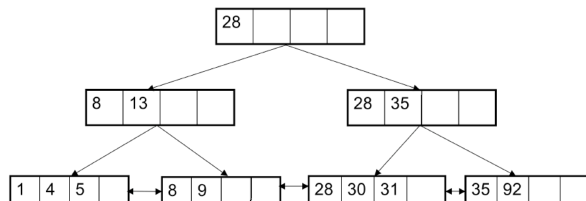

Figure 1: A B+-tree

In internal nodes, the $n+1$ pointers point to blocks in the next level and at least $\lceil (n+1)/2 \rceil$ of these should be used. Pointers are ordered increasingly and they represent consecutive nodes in the level immediately after. Explicitly, for $j+1$ pointers used there exist j keys, $K_1, \dots, K_j$ such that all the keys in the first node in the node level below are smaller than K_1, all the keys in the second node are between K_1 and K_2, and so forth.

Each leaf can contain at most n and no less than $\lfloor (n+1)/2 \rfloor$ search keys. Consecutive nodes are linked by pointers. Each search key is accompanied by a placeholder for the data. Depending on the implementation, it will contain a complete copy of the record (e.g., for clustered indexes) or a pointer to the actual record. For B+-tree we additionally require that all the search keys appear in the leaf nodes.

B-tree designs and implementations vary, and with these the different insertion and sorting algorithms. When a leaf is full (with search keys $K_1, \dots, K_n$) and a new key is inserted in the table, whose value is between K_1 and K_n, a *node split* or *split* occurs: a new leaf is created and the $n+1$ keys are divided between the original and the new leaf. Design principles dictate that the values should be split in two halves, one half for each leaf. However in some cases, such as with MySQL-InnoDB, the index is optimized and the search-key values may not be divided in halves.

In DbMS implementations of B/B+-trees each node is stored in permanent memory in units called page disks (typically of 8KB, 16KB) and these page disks are retrieved to RAM only when they are required. In brief, DbMSs process each data manip-

ulation command which is optimized and forwarded to a "storage engine" which will efficiently search the indexes and retrieve to RAM the required data. For example, this implies that when looking up for a given search-key value, the DbMS will not require to fetch to memory and inspect all the search keys.

It should be noticed that in DbMSs it is the storage input/output operations that dominate the cost of typical data manipulations. And since indexes save I/O operations, they save considerable time. On the other hand, a side effect of indexes is that the same data manipulation operation when performed with different values, will require different amounts of time to be completed. For example, inserts that produce node splits should take more time than inserts than do not. The next remark demonstrates that B-trees leak information.

Remark 1 *Consider a nonempty table and fix a field which is indexed by B+-trees. Assume that x_1 is the smallest search-key value in a leaf and that the search key with value x_1+B falls in this same leaf, for some positive integer B. Insert the values $(x_1+B)+i$, for $i=0,1,2,\ldots$ until there is a split. Then one of the following is true:*

- *There is a node split after making $k<n$ inserts. Then, there are at least $n-k-1$ search keys whose values are between x_1 and x_1+B.*
- *There is a split when $x_1+B+n-1$ is added, then the tree contains no values between x_1 and x_1+B.*
- *We inserted a duplicate key.*

This remark not only proves that B+-tree insertion leaks information to users if they can detect node splits, but it also gives a hint into how to exploit these leaks to design an attack. To apply our attack technique we need more information on how the B-tree is implemented in order to exploit this data leak. Below we describe 3 different cases of node splits with InnoDB which cover all the splits that take place during our attack.

For the first case, assume there is a node with a single value, $i+1$. We write this as $[i+1]$ —ignoring other leaves in the tree. We continue to add values $i+2, i+3, \ldots$. The insertion of $i+n+1$ produces a split and a new leaf is created. The two nodes will then look like $[i+1,\ldots,i+n][i+n+1]$; that is, after making n inserts (we are not counting $i+1$) we end with two nodes, the left node has n values and the right node only 1.

For the second case, assume there is a node $[i+1,K]$ that contains two values i,K with $i+1<K$, then adding the values $i+2,i+3,\ldots$ produces a split at the insertion of $i+n$ and a new leaf is created. The two nodes will then look like $[i+1,\ldots,i+n-1][i+n,K]$; that is, after making $n-1$ inserts (we are not counting $i+1$) we end with two nodes, the left node has $n-1$ values and the right node 2.

For the third case, assume there is a node $[i+1,K_1,\ldots,K_s]$ that contains several values $i+1<K_1<\cdots<K_s$, with $1<s<n$. We continue to add values $i+2, i+3, \ldots$. The insertion of $i+n-s$ produces a split and a new leaf is created. The two nodes will then look like $[i+1,\ldots,i+n-s,K_1][K_2,\ldots,K_s]$; that is, after making $n-s$ inserts (we are not counting $i+1$) we end with two nodes, the left node has $n-s$ values and the right node $s+1$.

In fact, these cases can be generalized (e.g., only the last few inserts need to hold consecutive values). Finally, we mention that InnoDB behaves symmetrically, so that the mirror images of the above cases hold as well.

2.2 Algorithm and results

Let us fix a nonempty table and a search key indexed using B+-trees. For simplicity, let us assume that search keys are primary (i.e., if we attempt to insert a repeated value, we receive an error notification) and they hold integers. Although the primary requirement is natural in many scenarios, our technique could be adapted to handle repeated keys. Let n denote the page size, i.e., the number of search keys that fit in a node.

Let us assume that we can connect to the DbMS (e.g., as a DB user), insert values to the tree and know whether this operation produced a split. Although this last requirement might seem excessive, we will prove later on that it can be replaced by a

more realistic assumption. In any case, split detection depends on the DbMS under attack and several parameters, including network latency, disk caching, etcetera. See Section 3.2 for more details on split detection.

The attack algorithm depends on the following input parameters: a value x_0 in the range of the fixed column and integers b and $B := b^r$, for some integer r, standing for a *base* and an *initial step size*. The attack succeeds after computing the minimum value y in the table that is bigger than x_0. Although the technique is —a priori— general and should be applicable to other database engines, we will describe the instantiation against MySQL configured to work with the InnoDB storage engine in its default installation. This is mainly because our attack technique relies not only on split detection, but in the changes that splits produce in the tree, and the fact that B+-tree implementations differ in each DbMS. In summary, designing and executing this attack against other DbMS is a difficult task and is out of the scope of this paper. The reader will understand what of the details we give are particular to the attack, and which are general and can be applied to other scenarios.

The attack algorithm relies on a procedure that receives an interval containing the key y, splits it in b parts of the same size, and detects in which of these lies y. Due to the particularities of InnoDB our attack is not straight forward and requires a brief setup. It can divided in three steps:

- **Setup:** We prepare the tree for applying the divide and conquer procedure. Once finished, y is the smallest value of its leaf and we obtain an interval of size b^r containing y (i.e., we obtain integers l, u such that $l < y < u = l + b^r$).

- **Divide and conquer:** We set $k := r$ and recursively apply a procedure that sets $k := k - 1$, takes as input an interval of size b^{k+1} containing y and so that y is the smallest value in a leaf, and returns an interval of size b^k containing y, also y is the smallest value in a leaf.

- **Last step:** Once we have $b^k < n$ we look for y by an exhaustive search on this interval, that is, we make an insert for each of the values in the interval until we receive a "repeated insert" error —which will mean that we have found y.

Roughly speaking, the setup procedure and each application of the recursive procedure require less than nb inserts. The following lemma summarizes the complexity of our algorithm.

Lemma 2 *Let notions and notations be as before. Then, there exists an algorithm that given:*

- *A starting point x_0.*

- *A step value b and a step exponent r.*

computes the smallest value in the tree that is bigger than x_0 and requires at the most $O(n \sum_{0 \leq j \leq r} i_j) = O(nb \log_b y)$ inserts, where the i_j are the b-ary expression for y; that is, $y = i_r + i_{r-1}b + \cdots + i_0 b^r$, $0 \leq i_j < b$ for $j = 0, \ldots, r$ and $i_0 \neq 0$.

This lemma implies that if we set $x_0 = 0, b = 10$ and $r = 6$ and $y = 3\,020\,581$ and assume that $n = 500$, then our attack requires in the order of $n(3 + 2 + 5 + 1 + 1) = 6000$ inserts. We do not prove this lemma due to space restrictions. A proof would require describing the complete algorithm and going through all the possible branches of this algorithm. As an example, we give the following pseudo-code snippet, which executes the procedure used in step 2 (divide-and-conquer) of our algorithm, a sketch of the proof on how it succeeds and estimate its complexity.

```
m:=0;
k:=k-1;
Repeat
 {

  Set m':=m;
  Insert keys l,l+1,... until a split is
    detected;
  Set m the number of inserts made;
  l := l + b^k;
 } Until m != n;
 l := l - b^k + m';
```

The input for each run are integers l, k such that $l < y < u := l + b^k$, y is the smallest value in

a leaf and there is no other key between y and u. More explicitly, let us assume that there are two consecutive leaves $[l][y, u, \ldots]$, where the search key l is alone in one leave, and the following leave contains y, u (with $y < u$) plus possibly other values. This procedure computes a new value l such that $l < y < u := l + b^{k-1}$ and thence two consecutive leaves in the tree look like $[l][y, u, \ldots]$. To prove this, we go through the procedure. The procedure first sets $k := k - 1$ and checks from first to last, which of the intervals $[l + hb^k, l + (h + 1)b^k]$ contains y for $h = 0, 1, \ldots, b - 1$.

For the first of these intervals, it inserts the values $l + b^k, l + b^k + 1, \ldots$ until there is a split. Then, there are two possibilities to consider: either $l + b^k < y$ or $l+b^k > y$ (if $l+b^k = y$ then we have found the key!) If $l+b^k < y$ then the search keys $l+b^k, l+b^k+1, \ldots$ are inserted in the first of the two leaves, that is $[l]$, and this results in Case 1 that we described in Section 2.1. It can be identified by the attacker because in this case the split will occur precisely after he has made the n-th insert. Hence, a new leave will be created and the three affected leaves will look like

$$[l, l+b^k, l+b^k+1 \ldots, l+b^k+n-2][l+b^k+n-1][y, u, \ldots]$$

If $l + b^k > y$ then the search keys $l + b^k, l + b^k + 1, \ldots$ are inserted in the second of the two leaves, that is $[y, u, \ldots]$, and this results in Cases 2 or 3 that we described in Section 2.1. It can be identified by the attacker because in this case the split will occur before he has made the n-th insert. In this case, a new leave will be created and the three affected leaves will look like

$$[l][y, l + b^k, \ldots][\ldots]$$

Notice, that in this case, the attacker is able to infer that $l < y < l + b^k + m = l + b^k + n - 1$. Finally, we must re-set $l := l - b^k + m'$ not to repeat insertions during the next run of this procedure.

Each execution of the above procedure takes at most hn inserts, and since $h < b$ we deduce that the claimed estimates hold.

The first step of the algorithm goes similarly but is more complicated to describe and is left out of this short paper.

3 Experiments

To determine the feasibility of our attack technique we customized it to attack MySQL-InnoDB configurations running in Windows XP. We launched our attack against tables holding different numbers of records, ranging from 1 to several thousands. A clustered index with integer search keys was the target of the attack and the index values were selected uniformly, during our first test in the set of 64-bits integers and at other tests as strings of 8 characters and 32 characters. The attacker connected to MySQL as a user running in the same server where the DB was running. Time measurements were taken using kernel32.dll functions `QueryPerformanceCounter` and `QueryPerformanceFrequency`. We also tested this from another computer a switch away in the same network as the web server.

The first byproduct of our research was a trick that allowed us to concentrate on the data leaks produced by the B+-tree implementation of InnoDB, without caring about spit detection, and next face the split detection problem. That is, at a first stage we produced an instrumented version of MySQL where splits were very easy to detect and designed the attack to work against it. Once done, we used this knowledge to implement the attack against the (uninstrumented) MySQL engine. The next sections describe these steps.

This same attack design method can be repeated for customizing our attack technique against other DbMSs configurations —provided one has the ability to instrument the detection of splits and read tree topologies.

3.1 A research and design framework

We produced an instrumented version of the MySQL engine that behaved exactly like the original one except that: inserts took 1 millisecond to add a key when no leaf was split and took 100 milliseconds when a split occurred. Additionally, the instrumented MySQL took snapshots of the topology of the tree before and after a split. With this functionality available, we were able to experiment and design the attack as described in the above algorithm.

3.2 Detecting a split

Once the attack was working for this instrumented MySQL, we undertook the problem with the MySQL-InnoDB engines *as is* and tested it against a sample of tables of different sizes. Adapting the attack was not trivial, and it was necessary to develop a special split detection algorithm that we follow to describe.

In designing our attack, we did some experimental measurements in a controlled scenario. All measurements are dependent on the computer where the engine runs, the table under attack and the computer running the attack. However, our algorithm adapts to variations of these parameters (for example, we tested the attack from another computer a switch away and it worked without modifying the code, and we tested the attack running MySQL in another computer and it worked fine as well). The MySQL engine takes longer time to respond from an insert that produces a split than from an insert that does not. MySQL optimizations, hard-drive specifics, caching, CPU usage and other factors act as noise and, regrettably, noise might render an insert which produces a split indistinguishable from one that does not when the only information used to decide this is the time taken by MySQL to make the insert (taking into account our measurement limitations).

For our split detection algorithm to work we require that there is a threshold value t_* so that most of the inserts that produce splits take more time than t_* and most of the inserts that do not produce splits take less time than t_*. In order to overcome the noise interference, we devised a statistical test that guesses when splits occur. This test profits from the fact that most of the inserts that do no produce splits take little time to respond (smaller than t_*), and only a handful of these take as much time as inserts that split; e.g., during our experiments the mean time of the inserts that generated splits doubled the mean time of inserts that did not (the means were 73ms and 32ms respectively). The following remark is an immediate conclusion of the existence of this threshold. Notice, however, that if no threshold can be computed, then there might be other means for assessing these leaks[1].

[1] When attempting to estimate this threshold value in other computer systems we found evidence of trade-offs that help or damage the detection probability, e.g., faster hard-drives make it more difficult to detect splits, but larger search-key values make it easier. Understanding exactly what are all the parameters that affect the attack and how they interplay is outside the scope of this paper.

Remark 3 *Let k be a positive integer. Consider consecutive inserts which took time $t_1, t_2 \dots$ respectively. Let i be such that $t_i, t_{i+n}, \dots, t_{i+kn}$ are all bigger than t_*. Then, the probability that these timings do not correspond to splits goes to 0 as k grows, and the probability that they do belong to splits goes to 1 as k grows.*

Our application of this remark required balancing efficiency with accuracy, and this was in an *ad hoc* fashion. Explicitly, to decide whether a split occurred, we made consecutive inserts and recorded the number of inserts that took more than t_*, we then matched this with a table that contained all the possible values and the associated consequences. For example, if there were three of these that were each a distance n apart, then we assumed that they corresponded to the *Case 1* described in Section 2.1. In the next section we show how to combine this with the attack algorithm described before. In testing this method, we discovered that it has a very low rate of false positives (i.e., when our method claims it detected a split, it is because there was a split), but a less appealing rate of false negative alarms (e.g., several node splits were not detected). Luckily, this method was sufficiently accurate for executing the attack.

3.3 Combining both algorithms

The attack algorithm described in Section 2.2 requires that the attacker is able to detect all the splits that he generates precisely after he makes the "generating" insert. However, our split detection method might fail and this should be taken into account. On the other hand, as we mentioned in the previous section, split detection can fail either detecting a spit when no split occurs or ignoring a split. In the former case, our attack algorithm will fail to compute a key (although some error recognition and rewinding could be implemented, it is out of the scope of this paper). On the other hand, in the latter case,

the algorithm can be adapted to compute the key successfully.

For example, if $l < l + b^k < y$ and we insert values $l + b^k, l + b^k + 1, \ldots$, then there will be node splits at $l + b^k + (n-1) + hn$ for $h = 0, 1, \ldots$. These are the node splits covered in Case 1 of Section 2.1, hence if we detect three of these splits in a row, we deduce that $l + b^k < y$. If we have that $y < l + b^k$ and the leaf containing y is $[y, u]$, then inserting $l + b^k, l + b^k + 1, \ldots$ will result in node splits at $l + b^k + (n-2) + hn$ for $h = 0, 1, \ldots$. These are the node splits covered by Case 2. Hence, if we detect three of these node splits in a row, we deduce that $y < l + b^k$. The situation where there are more keys in the leaf with y, u can be tackled similarly.

At any rate, we can precompute a table that describes all the possible cases, and combine our attack algorithm with the split detection method we just described to produce a *complete* algorithm that attacks standard (un-instrumented) MySQL engines.

3.4 Statistics

We executed the attack against different scenarios. In each case, the MySQL engine (as downloaded from `http://www.mysql.org`) ran in a fixed computer under Windows XP.

We executed our attack against a table with a single column of 64 bit integers with 1, 101 and 1001 keys. Below find a table with the attacks we executed, where columns include the number of keys in the table, whether the attack was successful or not, the number of inserts made, and the time in minutes and seconds spent by each attack.

# of keys	Result	# of inserts	time elapsed
1	Success	14291	09:48
1	Success	14864	11:13
1	Success	13145	10:52
101	Success	13145	10:54
101	Success	13145	10:53
101	Success	13145	10:11
1001	Success	12858	09:56
1001	Failed	10590	08:34
1001	Failed	20094	15:47
1001	Success	12592	08:33
1001	Success	15723	11:09

4 Final remarks

We have devised a general technique that allows us to retrieve keys from a table in a database engine, only by requiring that we are allowed to make inserts and compute the time the database engine takes to answer.

We mention some open questions. First, it remains to understand under what the conditions does the attack technique work. That is, can we apply a procedure that will tell us a priori if our attack can be successful against a given setting? We already have some means that help to answer this question, and that is our split detection algorithm. If the split detection algorithm detects splits with good probability, this means that we can detect information leaks and (might probably) be able to execute the attack. However, we believe that this algorithm could be replaced with a split detection method which is more efficient and has a better success probability. This is left for future work.

Another question that one can make is what countermeasures could one take to block this attack methodology, or if not to block, to detect attacks which are ongoing or have occurred. Again, we cannot answer this question but give partial solutions. For example, implementing some sort of transaction throttling (e.g., limiting the number of inserts per database user or IP address); using anomaly detection techniques in the connection to the DbMS to statistically detect known forms of this attack (e.g., a large number of splits or consecutive inserts); one could also apply a *blinding* operation to each search-key value (see, e.g., [8], [11], [4]); or altering the B-tree algorithms to thwart information leaks. Doing these analysis from security logs might be easier, so after-the-fact detection will probably be more accurate.

It further remains to analyze how to tune up our attack for the different scenarios with a fixed DbMS (e.g., with MySQL and InnoDB), how to extrapolate the attack to other DbMSs, and analyze whether more efficient attacks can be designed (e.g., can we optimize the attack to compute all the keys in the table).

References

[1] Chris Anley. Advanced SQL injection in SQL server applications. NGSSoftware Insight Security Research (NISR) Publication, 2002.

[2] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix)*. ACM, 1971.

[3] Rudolf Bayer and Karl Unterauer. Prefix B-trees. *ACM Trans. Database Syst.*, 2(1):11–26, 1977.

[4] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *LNCS*, pages 1–12. Springer, 1998.

[5] Andrew Bortz, Dan Boneh, and Palash Nandy. Exposing provate information by timing web applications. In *World Wide Web Conference (WWW 2007), Track: Security, Privacy, Reliability and Ethics. May 8–12, 2007. Banff, Alberta, Canada*, 2007.

[6] David Brumley and Dan Boneh. Remote timing attacks are practical. In *12th Usenix Security Symposium, Washington DC, August 4–8, 2003, proceedings of*, 2003.

[7] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password interception in a ssl/tls channel. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, Lecture Notes in Computer Science, 2003.

[8] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Proceedings of CRYPTO '82. Plemum, New York, 1983*, pages 199–203, 1982.

[9] Edward W. Felten and Michael A. Schneider. Timimg attacks on web privacy. In *Proceedings of the 7th ACM Conference on Computer and Communications Security. November 1-4, 2000, Athens, Greece. ACM, 2000*, 2000.

[10] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Perntice Hall, 2000.

[11] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *LNCS*. Springer, 1996.

[12] Mateo Meucci (editor). The OWASP testing guide v2. `http://www.owasp.org/index.php/Image:OWASP_Testing_Guide_v2_pdf.zip`, 2007.

[13] Anthony Molinaro. *SQL Cookbook*. O'Reilly, 2005.

[14] MySQL. *MySQL 5.0 Reference Manual*, 2007.

[15] Rain Forest Puppy. NT web technology vulnerabilities. *Phrack Magazine*, 8(54), 1998.

[16] Andrew van der Stock, Jeff Williams, and Dave Wichers. The ten most critical web-application security vulnerabities (2007 update). OWASP technical report. URL: `http://www.owasp.org/index.php/Top_10`, 2007.