Search-Based Security Testing of Web Applications

Julian Thomé
Saarland University
Saarbrücken, Germany
s9jnthom@stud.uni-saarland.de

Alessandra Gorla
Saarland University
Saarbrücken, Germany
gorla@cs.uni-saarland.de

Andreas Zeller
Saarland University
Saarbrücken, Germany
zeller@cs.uni-saarland.de

ABSTRACT

SQL injections are still the most exploited web application vulnerabilities. We present a technique to automatically detect such vulnerabilities through targeted test generation. Our approach uses search-based testing to systematically evolve inputs to maximize their potential to expose vulnerabilities. Starting from an entry URL, our BIOFUZZ prototype systematically crawls a web application and generates inputs whose effects on the SQL interaction are assessed at the interface between Web server and database. By evolving those inputs whose resulting SQL interactions show best potential, BIOFUZZ exposes vulnerabilities on real-world Web applications within minutes. As a black-box approach, BIOFUZZ requires neither analysis nor instrumentation of server code; however, it even outperforms state-of-the-art white-box vulnerability scanners.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Security

Keywords

Security testing, SQL injections, Search-based testing

1. INTRODUCTION

Web applications are easy to access, and easy to deploy. Unfortunately, they also are easy to attack. SQL injections (SQLI) form one of the most common class of attacks. An SQLI attack inserts malicious SQL statements into an entry field for execution—to extract data, to bypass authentication, or to execute remote commands.

As an example for a SQLI attack, consider WebChess, a Web application for online chess playing [29]. When logging into *WebChess* (Figure 1), the user has to provide

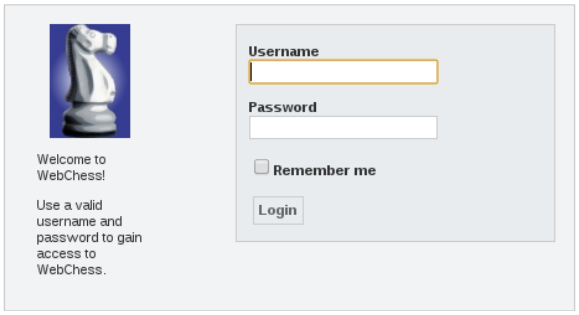

Username
Password
Remember me
Login
Welcome to WebChess!
Use a valid username and password to gain access to WebChess.

Figure 1: *WebChess* login page. To bypass authentication, enter ' OR 1=1 # as user name, and an arbitrary password.

```
<?php
  ...
  $query="SELECT * FROM players WHERE nick='".
      $_POST['username']."' AND password='".
      $_POST['pwd']."' LIMIT 1";
  $login_check=$db->fetch($query);
  if (!$login_check) {
    //login failed
    exit;
  } else {
    //successful login
  }
?>
```

Figure 2: *WebChess* login code, omitting sanitization.

his user name and password. The PHP code in Figure 2 would then issue a SQL query to check whether the user is authenticated—namely by checking whether user name and password appear in the "players" table:
SELECT * FROM players WHERE nick='max' AND password='1234'

If a malicious user, however, enters a "user name" in the form ' OR 1=1 #, the query becomes:
SELECT * FROM players WHERE nick = ' ' OR 1=1 #' AND password = '1234'

In this statement, the SQL expression 1=1 always evaluates to true, and everything behind the hash sign becomes an SQL comment. This query always returns a non-empty result and thus allows the user to bypass authentication.

Injecting ' OR 1=1 # is an instance of a so-called *tautology attack,* which is just one of many patterns of SQL

SBST'14, June 2 – June 3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2852-4/14/06...$15.00
http://dx.doi.org/10.1145/2593833.2593835

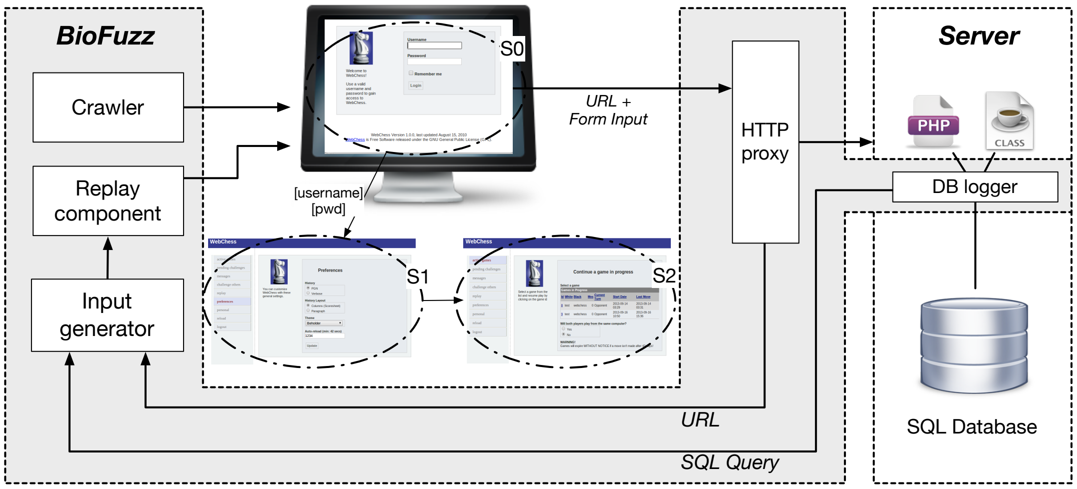


Figure 3: How BIOFUZZ works. Using a Web crawler, BIOFUZZ builds a finite state machine model of the web application, and by means of a HTTP proxy it identifies the parameters exchanged between clients and servers. The input generator then sends form inputs to the server, and evolves the inputs by checking how close the resulting SQL queries are to an attack. The result is a set of SQLI attacks exposing vulnerabilities in the application.

injections. Due to the risk of SQLI attacks, a number of techniques has been developed to prevent and detect them. *Sanitizing* escapes quotes in third-party input, as well as other fragments with semantics. In a large Web application, however, one single missed sanitization still exposes full vulnerabilities. *White-box* techniques thus instrument or analyze server code to track information flow through the code; however, they require that the code be available for analysis or instrumentation, which is difficult in systems composed of several third-party components. *Black-box* techniques test Web applications against given attack patterns. They do not require code access, but are limited in the set of patterns. Due to these deficiencies, SQLI are still the most exploited Web vulnerabilities.[1] [2]

In this paper, we present and explore BIOFUZZ, a security tester for Web applications that uses *evolutionary black-box testing* [3] to detect vulnerabilities, specifically SQL injections. As sketched in Figure 3, BIOFUZZ generates inputs at the Web application level and intercepts SQL statements at the interface between server code and database. In contrast to white-box techniques, BIOFUZZ does not require code access and thus is applicable in a far wider range of situations. In contrast to the state of the art in black-box techniques, BIOFUZZ is not restricted to a set of given patterns, but instead systematically *evolves inputs* to detect vulnerabilities.

To evolve inputs, BIOFUZZ leverages *search-based testing*, a family of *test generation* techniques that use evolutionary algorithms to systematically evolve a population of inputs towards a given *target:* the higher the *fitness* of an input, the greater its chance to be further evolved. The typical target of search-based testing is to maximize coverage; consequently, the closer an input gets to yet uncovered behavior, the higher its fitness. In our case, however, the target is to find SQL vulnerabilities. As a fitness function, BIOFUZZ thus specifically evolves inputs whose characteristics are *close to triggering an attack:*

a) *Exhibit SQL grammar fragments.* This favors a malicious input, such as the tautology attack ' OR 1=1 #, over regular inputs.

b) *Pass as many SQL tokens as possible into the query.* This favors complex SQL inputs, such as the *piggybacked query attack* '; DROP TABLE players #, which deletes the table from the database.

c) *Minimize the difference between the provided input values and the values that appear in the SQL query.* This bypasses common sanitization techniques: For instance, if the well known attack ' OR 1=1 # were sanitized, this rule may prefer the syntactically different but semantically equivalent ' OR COS(0) = SIN(PI()/2) #.

d) *Alter the visible output in the web application.* This favors attacks that produce information leaks, such as the *union query attack* ' UNION ALL SELECT * FROM players #, which retrieves a full list of players and passwords.

BIOFUZZ synthesizes all these types of attacks automatically from scratch, using only the fitness function as described above; it thus can detect a large number of vulnerabilities in Web applications even without any prior knowledge on successful attack vectors. The combination of black-box testing and systematic evolution makes BIOFUZZ both applicable and effective: In our evaluation, BIOFUZZ detected 37 exploitable vulnerabilities in six mid-sized Web applications, performing better than ARDILLA, a state-of-the-art white-box technique and *sqlmap*, a state-of-the-art black-box technique.

[1]`https://www.whitehatsec.com/resource/stats.html`
[2]`http://www.veracode.com/blog/2013/07/the-real-cost-of-a-data-breach-infographic`
[3]BIOFUZZ is a black-box technique in the sense that it does not need access to the server side code. Still, it requires access to the server, because it needs to log the interactions to the database.

The remainder of the paper is structured as follows: Section 2 explains how BIOFUZZ works, and provides details on all its major components. Section 3 presents the evaluation results, while Section 4 compares BIOFUZZ with the related work, and Section 5 presents the conclusions and future works.

2. THE BIOFUZZ APPROACH

Figure 3 shows the main components of BIOFUZZ. BIOFUZZ works in two phases. In the first phase, BIOFUZZ uses a crawler to navigate through the Web application pages, and keeps track of all the inputs and events that lead from the entry URL to other web pages. A HTTP proxy is used to collect all the GET and POST parameters that client and server exchange. In the *WebChess* login page, for instance, these are the POST parameters *username* and *pwd*. By means of a database logger component, BIOFUZZ identifies which input parameters are likely to be used in SQL query statements, and as a consequence it considers these input parameters as the target of possible SQLI attacks.

During the second phase, BIOFUZZ tries to generate SQLI attacks for each of the target input parameters that were identified in the first phase. In order to do that, it replays all the events and provides all the inputs that were needed to get from the entry URL to the Web page containing the target input parameters. Finally, using its input generator component, it generates input values that are likely to expose SQLI vulnerabilities. Section 2.1 explains the first phase, and Section 2.2 the second one.

2.1 Identifying Target Input Parameters

The first essential step to identify SQLI vulnerabilities is to identify which input parameter values are likely to flow into SQL query statements, since such input parameters are the ones that might be vulnerable to SQLI attacks. White-box techniques can precisely identify which input parameters are vulnerable thanks to taint analysis [13, 8]. Given a set of sources, i.e., the input parameters, taint analysis identifies which SQL query statements in the server side code are potentially influenced by such inputs, and marks them as potentially dangerous. Black-box techniques, such as BIOFUZZ, do not have access to the server side code, and as a consequence it is harder to identify which input parameters may be vulnerable to SQLI attacks.

A well known black-box test to identify vulnerable parameters is to inject a single quote character (') as an input to all the text fields of a Web page. If such an input leads to error messages from the DBMS, then it means that the input field is exposed to SQLI. This simple trick may work for several applications, but it does not work for the *WebChess* login example presented in the introduction. The reason is that even if the (') character raises a DBMS error, the server side code masks this error, thus preventing it to propagate to any visible element in the web page. This test may fail also in presence of partially sanitized inputs. A web developer, in fact, may realize that some input parameters are exposed to SQLI attacks, and as a consequence may wisely decide to escape the single quote character (') when reading the `username` and `pwd` values. Most black-box techniques would think that an input field is not vulnerable, since it ignores the (') character. However, alternative SQLI attacks could still be generated. A web application might, for instance, use the `addslashes()` function to sanitize the input. Whenever a quote character `0x27` appears in the input, this function escapes it by putting a backslash character `0x5C` before it, such that `0x27` becomes `0x5C27`. However, when the database uses a Chinese charset, some characters are encoded in single bytes and some in double bytes. The character `0xBF5C` is considered as one single character with a length of two bytes. This fact can be exploited by injecting the username `0xBF27 OR 1 = 1 #`, such that the result of `addslashes($username)` would be `0xBF5C27 OR 1 = 1 #`. The database will then treat `0xBF5C` and the following `0x27` as separate characters, and consequently the input would be transformed to ' OR 1=1 #, and would thus successfully generate the following tautology attack:

SELECT * FROM players WHERE nick=' ' OR 1=1 # AND password=''

To identify target input fields, BIOFUZZ relies on a crawler, a HTTP proxy, and a database logger. The crawler allows BIOFUZZ to explore the Web application, and to this end, we adopted *Crawljax,* an open source crawler for Web 2.0 applications [20]. While crawling a Web application, BIOFUZZ builds a finite state model similar to the one represented in Figure 3. DOM trees of visited Web pages represent states (e.g., S0 represents the *WebChess* login page, S1 represents the first page after a successful login, and S2 represents the web page when a user starts a new game), and transitions represent the actions (e.g. click on links, form submissions) required to go from one state to another. Transitions are annotated with GET and POST parameters that the HTTP proxy observed during the HTTP requests. Thus, for instance, when crawling the *WebChess* application, the transition between the login page (S0), and the main page after a successful login (S1) would be annotated with the post parameters *username* and *pwd*. At the end of the crawling phase, we identify which transitions have input parameters, and thus should be further explored with BIOFUZZ. Notice that this step can only identify which input parameters a web application has, but what BIOFUZZ needs is a list of input parameters whose values might flow into query statements. To identify which inputs may have an effect on the query statements, BIOFUZZ relies on a *database logger*, which logs all the executed SQL queries before executing them. We used an instrumented version of the `php_mysql` module to intercept calls to the database together with their return codes.[4]

By providing random values as input, and by checking if such values appear anywhere in the log file produced by the database logger, BIOFUZZ has a high confidence that the provided input flows into query statements in the server side code. Such input parameters then, become targets for the second phase, i.e. generation of SQLI attacks.

2.2 Generating SQLI Attacks

In the second phase, BIOFUZZ analyzes the finite state model produced in the first phase, and looks for all the transitions that contain at least one likely exploitable input parameter. In the *WebChess* example of Figure 3, the transition between the login page (S0) and the page after the login (S1) would be the only one. By means of the replay component, BIOFUZZ replays all the events and provides all the inputs to take the application from the entry URL to the

[4]The implementation of BIOFUZZ is currently bound to PHP applications using MySQL as DBMS, but the technique itself does not have these limitations.

state before the considered transition. In this example no replay is required, since the state preceding the transition (S0) is the entry point.

2.2.1 Instance Generation

BIOFUZZ generates a set of random values (one for each parameter contained in the transition), and generates a new HTTP request providing these values as inputs. It then retrieves from the database log the queries that were executed containing these random values. In the *WebChess* login page example, for instance, by providing '0000' as a random input value for the `username` parameter, the database logger would report the following query:

SELECT * FROM players WHERE nick = '0000' AND password = ''

Using a context-free grammar representing SQL queries, BIOFUZZ parses the SQL statement up to the point where the random values appear. This point is what we call the "*prefix barrier*":

SELECT * FROM players WHERE nick = '

BIOFUZZ then generates a set of individuals, which are represented as *parse trees* of the SQL query after the prefix barrier, and randomly appends new tokens by following the grammar. Notice that this process does not necessarily produce complete statements, because only one token is added at every iteration. The reason for this decision is that by adding a token per time, and by checking the effects caused by the newly added token, we can be more accurate in guiding the input generation. BIOFUZZ, then, takes whatever has been generated after the prefix barrier, and provides it as a new input via the replay component.

By parsing new queries, BIOFUZZ gathers more information about the database schema (i.e, column and table names). BIOFUZZ extends the initial grammar with this new information, thus allowing to generate context-sensitive SQLI attacks. The grammar represents the search space of possible solutions, i.e., SQLI attacks, and by extending the grammar with the learned database schema, the search space grows even more. Given that it is time consuming to evaluate a generated input, since BIOFUZZ has to replay all the events, check the HTTP proxy and the database logger, BIOFUZZ implements a search-based approach to guide the input generation, and get to a valid solution (i.e. a valid SQLI attack) faster.

2.2.2 Fitness Function

In a search-based approach, the fitness function measures the quality of an individual I within a population P, and is essential to guide the evolution of individuals towards the desired solution. In our context, the individuals are the parameter inputs, and the fitness value of the individuals represents how close these inputs are from generating a successful SQLI attack. The fitness function that BIOFUZZ implements considers the following components:

1. **Node Count (NC)** The node count is the number of all newly generated terminals of an individual divided by the sum of generated terminal within one population. If the node count of an individual is high, it means that BIOFUZZ was able to inject a lot of tokens as inputs. NC is important to guide the evolution towards more complex SQL injections (e.g., piggybacked queries).

$$NC(I) = \frac{\#GeneratedTerminals(I)}{\#GeneratedTerminals(P)}$$

2. **Checkpoint Count (CC)** Since we want to guide the token generation towards complete SQL statements that can be executed on the database without errors, we count the number of "*checkpoints*" that an individual has. We call checkpoints those tokens that mark the end of valid queries (e.g ';'). We favor individuals that have a high number of checkpoints because they represent longer statements, and are complete (i.e., they do not lead to errors due to malformed queries):

$$CC(I) = \frac{\#Checkpoints(I)}{\#Checkpoints(P)}$$

3. **Input Proximity (IP)** This value measures how close the string that has been injected as an input $S(I)$ is from the string that appears in the database s_{log}. When this value is less than 1, it means that the input has been sanitized, and as a consequence, alternative strings should be generated. We use the Levenshtein distance $ldis(s,t)$ to measure the distance of the two strings s and t. We favor those individuals that have a high IP (i.e. low distance):

$$IP(I) = \frac{1}{1 + ldis(S(I), s_{log})}$$

4. **DOM Distance (DD)** It is sometimes the case that a successful SQL attack leads to visible differences in the Web page. We favor individuals that can cause bigger differences in the new DOM tree (dom_{new}) compared to the one observed during the crawling phase ($DOM(I)$), since they indicate that the input caused significant changes to the behavior of the Web application. To measure the distance, we use once again the Levenshtein distance:

$$DD(I) = \frac{ldis(DOM(I), dom_{new})}{len(longest(DOM(I), dom_{new}))}$$

To weight the influence of each component on the fitness value of an individual, we use four different weights (s_1, s_2, s_3, and s_4) whose sum must be 1. The overall fitness value $F(I)$ of an individual I is therefore computed as

$$F(I) = s_1 \times NC(I) + s_2 \times CC(I) + s_3 \times IP(I) + s_4 \times DD(I)$$

The s_i weights can be configured by the user. According to our experience, the Checkpoint Count and the DOM Distance are the most important factors, and should therefore have higher values. Section 3.1 has details on the values used in our evaluation setting.

2.2.3 Mutation and Crossover

The Genetic Algorithm implemented by BIOFUZZ relies on mutation and crossover operations to evolve the inputs across different populations. Mutation means to apply small changes to an individual, and crossover means to combine two individuals in an attempt to generate better solutions. In our context, we defined the following mutation operations:

- **Case**: Changes the case of a string to upper, lower or mixed case.
- **Encoding**: Changes the representation of single characters in a string. This can cause, for instance, the character (') to be mutated into CHAR(39).

Both mutation operations can be effective to circumvent some sanitization functions. Crossover operations, instead, are useful to propagate parts of successful solutions to other individuals. In this context we implemented crossover by allowing two individuals to exchange part of their sub-trees.

The process described in this section takes place for every transition in the Web application model that contains at least one parameter. In the next section, which presents the evaluation results, we refer to the process of generating SQLI attacks for a parameter associated with a transition as a "BIOFUZZ instance".

3. EVALUATION

To evaluate BIOFUZZ, we focused on two main research questions (RQ):

- **RQ1:** *Is BIOFUZZ effective in detecting and exploiting SQLI vulnerabilities?*

  With this RQ, we aim to assess whether BIOFUZZ is useful in exposing SQLI vulnerabilities and in generating successful SQLI attacks that exploit vulnerabilities in real applications. To this end, we took six real mid-sized applications that have known SQLI vulnerabilities, and we checked whether BIOFUZZ could detect them, and generate valid SQL attacks.

  The effectiveness evaluation also aimed to assess whether the solutions produced by BIOFUZZ were *diverse* (i.e., do not produce the same attack patterns), as an effect of the search-based nature of BIOFUZZ. Diverse solutions would show that BIOFUZZ is more general and more powerful than other similar techniques that generate SQLI attacks on the basis of a set of predefined patterns.

- **RQ2:** *How does BIOFUZZ compare with other similar white-box and black-box techniques?*

  Here, we compare the effectiveness of BIOFUZZ with other similar techniques; specifically ARDILLA, an effective white-box technique, and *sqlmap*, an advanced black-box tool. To ease the comparison, we evaluated BIOFUZZ on the same subjects that ARDILLA was evaluated on. These are the first four web applications listed in Table 1, i.e., *WebChess*, *Schoolmate*, *FaqForge* and *geccBBlite*.

The next sections present the evaluation setup and the results of our evaluation.

3.1 Evaluation Subjects and Setup

Table 1 reports the six Web applications that we considered in our evaluation. The table reports the name of the application, the version we considered ("Version"), the number of lines of code ("LOC"), and the number of states ("#States") as well as the number of GET and POST parameters ("#Pars") that we observed during the application crawling.

Table 1: Evaluation Subjects

	Version	LOC	#States	#Pars
WebChess	0.9.0	3,376	10	13
Schoolmate	1.5.4	6,923	18	27
FaqForge	1.3.2	512	4	2
geccBBlite	0.2	323	11	5
phpMyAddressbook	8.2.5	47,481	6	9
Elemata	RC3.0	3,212	18	6

Our selection of subjects is motivated as follows: The first four subjects stem from the ARDILLA paper [13], since we wanted to compare the results of BIOFUZZ with ARDILLA:[5] *WebChess* is an application that allows multiple players to play online chess games against each other and to keep track of their results. *FaqForge* is a simple application to manage FAQs and *geccBBlite* is an online forum. *Schoolmate* is meant for course-management and it is targeted towards schools and teachers.

Since these four web applications are rather old, and some of them are not maintained anymore, we included two additional Web applications for which some SQLI vulnerabilities were recently reported via the Common Vulnerabilities and Exposures (CVE) Website[6]: *phpMyAddressbook* is a Web application that allows users to manage contacts, and *Elemata* is a simple Content Management System (CMS).

For these six applications, we provided a URL as an entry point, and we let BIOFUZZ automatically crawl the application, identify the vulnerable input fields, and generate an SQLI attack. In some cases the entry point was the login page (*WebChess*, *Schoolmate*, *phpMyAddressbook* and *Elemata*), and in order to let the crawler work, we manually provided the login information. During our evaluation we ran into some limitations of the crawler, reported in Section 3.4. We ran our evaluation on a desktop PC with an Intel Core i5 650 and 4 GB of RAM, and we used the following configuration for the evolutionary algorithm:

- **Iterations: 20**. We gave BIOFUZZ a maximum number of iterations as a stopping criterion. An iteration denotes the attempt of improving each individual using the generation of new tokens/mutation or crossover. Applications without any input sanitization need few iterations to generate successful SQLI attacks, but applications with partial sanitization may require more. 20 as number of iterations is a good compromise between required time and quality of the SQLI attacks generated by BIOFUZZ.
- **Crossover Cycle: 2**. This parameter defines the crossover frequency. Given a value of 2, BIOFUZZ attempts a crossover operation after every second iteration. A number greater than one gives newly created individuals the chance to reintegrate and gain a better fitness value before performing a crossover operation.
- **Crossover Offset: 5**. The crossover offset defines the number of iterations after which BIOFUZZ should

[5]ARDILLA has been evaluated on another web application, *eve*, but unfortunately this application seems to be no longer available, so we could not include it in our evaluation subjects.

[6]http://cve.mitre.org

Table 2: Effectiveness Results

	#Vulnerable Parameters	#Vulnerabilities	Avg. Fitness	#Instances	#Individuals	Run time (s)
WebChess	4/13	13	0.25932	8(3)	304	596
Schoolmate	4/27	6	0.25561	21(5)	534	1,687
FaqForge	1/2	1	0.27520	2(1)	20	32
geccBBlite	3/5	4	0.25988	8(6)	300	656
phpMyAddressbook	4/9	10	0.18531	17(9)	510	2,672
Elemata	1/6	3	0.34008	13(6)	270	191

start to perform crossover. At the beginning, an individual I does not have any tokens, and the crossover operation does not work in absence of tokens. As a consequence, it is necessary to wait few iterations before allowing good individuals to propagate subtrees thanks to crossover.

As mentioned when introducing the fitness function in Section 2.2.3, we weighted the fitness elements as follows: Node count (NC): $s_1 = 0.125$, Checkpoint Count (CC): $s_2 = 0.375$, Input Proximity (IP): $s_3 = 0.25$, and DOM Distance (DD): $s_4 = 0.25$.

3.2 RQ1: Effectiveness

The first and most important research question is about the effectiveness of the BIOFUZZ approach. Given that BIOFUZZ aims to identify vulnerable input parameters, and aims to generate SQLI attacks for such vulnerabilities, we were interested in measuring how many of the known vulnerabilities BIOFUZZ could be able to exploit.

3.2.1 Vulnerabilities Found

Table 2 reports the results of this experiment. For each application, we report the number of vulnerable parameters ("# Vulnerable Parameters") out of the total number of parameters. A parameter is considered *vulnerable* if it is possible to inject SQL statements that may lead to successful SQLI attacks.

The next column ("#Vulnerabilities") reports the number of vulnerabilities that BIOFUZZ could exploit for each web application. The number of vulnerabilities represents the unique pairs of input parameters and query statements that BIOFUZZ could successfully exploit with SQLI attacks. Thus, for instance, if a single parameter (e.g. `username`) can flow into two different SQL statements (e.g. SELECT * FROM players WHERE nick='*username*', and UPDATE players SET pwd='*newpwd*' WHERE nick='*username*'), and BIOFUZZ can successfully inject SQL via the `username` parameter, and successfully generate attacks using both SQL statements, then we consider these as two vulnerabilities.

BIOFUZZ could successfully find and exploit all the vulnerabilities that we were aware of except one. By manually checking the source code of the applications, we later tried to identify additional exploitable vulnerabilities, but we could not identify more.

In the evaluation subjects, BIOFUZZ could successfully identify 37 out of 38 known SQLI vulnerabilities.

3.2.2 Evolutionary Progress

Table 2 also reports some data about the evolutionary algorithm. The column "#Individuals" reports the number of individuals that were generated during the search, and the column "#Instances" reports the number of BIOFUZZ instances that were created. The number of BIOFUZZ instances represents the number of transitions that contain at least one parameter that is likely to be vulnerable. The number in brackets is the number of instances that could generate at least one successful SQLI attack. The number of individuals is also correlated with the number of parameters. The more parameters there are, the more individuals BIOFUZZ generates in total.

Column "Avg. Fitness" reports the average fitness value computed over all the population. The lower this number is, the worse the quality of solutions were in average according to our fitness function. In the case of *Elemata*, for instance, the average fitness is pretty high because the vulnerabilities were quite easy to exploit from the first few iterations. As a consequence, the individuals in the first few iterations had already a good fitness value, and the following generations could improve on already good ancestors. Another reason for the higher fitness value in *Elemata* is that successful attacks lead to completely different DOM trees in the HTTP response, and as a consequence the DOM distance value of the fitness function has a high impact.

Such high fitness values do not occur in other cases. For instance, *phpMyAddressbook* shows a warning message from the database, but the rest of the Web page stays the same, and as a consequence, the DOM distance is minimal. The main reason for the low fitness value of *phpMyAddressbook*, though, is that some individuals tried to generate attacks for a sanitized parameter without success, and thus had final low fitness values. The PHP code reads the GET parameter, and, by means of the `isNumeric()` function, it checks whether the value of the parameter is a number. In the case it is not, it discards the value, thus preventing SQL injection. Since BIOFUZZ realized during the first phase that some random numeric inputs could flow into some query statements, it considered this a likely vulnerable parameter, and thus it kept generating, mutating and crossing over individuals in an attempt to workaround the sanitization. Unfortunately, without success. When we later manually analyzed this case, we could not come up with a valid attack ourself.

Sanitized parameters have negative impact on the fitness value, and the search in this cases has no guidance.

3.2.3 Efficiency

Table 2 reports also the run time in seconds that BIOFUZZ needed to analyze the whole application and exploit all the vulnerabilities. Overall, we can say that BIOFUZZ can efficiently analyze entire Web applications to expose SQLI vulnerabilities, as in the worst case (i.e., *phpMyAddressbook*) it needed less than 45 minutes to expose 10 vulnerabilities.

BIOFUZZ converges quickly on vulnerabilities.

3.2.4 *Diversity of Solutions*

An effective technique that aims to expose SQLI vulnerabilities should be able to generate solutions with diverse SQLI attacks. Diverse solutions show the generality of the technique in identifying different types of attacks, and show that a technique is likely to be more effective in reporting vulnerabilities that can be exploited with non trivial attacks, in contrast to techniques that use a set of predefined attack patterns. To evaluate the diversity of the solutions that BIOFUZZ produces, we manually analyzed the top three individuals in each population according to their fitness value, and among these we could find examples of tautologies, union query attacks, and attacks to retrieve sensitive information.

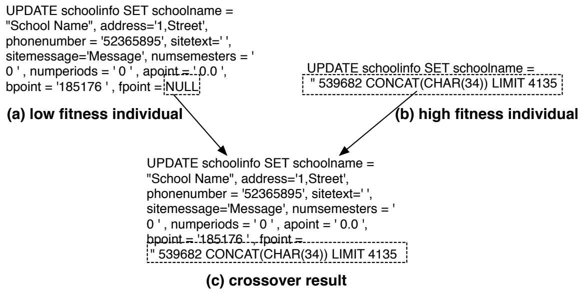
BIOFUZZ could also generate several attacks involving piggybacked queries. However, none of these have high fitness values in our evaluation. The reason is that the PHP code of our evaluation subjects always use the `mysql_query()` function, which considers a unique query and simply ignores any appended query. To inject piggybacked queries, the PHP code should have used the `mysql_multi_query()` function, instead, which can execute multiple queries in sequence. Since all the appended queries were ignored, the checkpoint count value of these individuals was low, and were consequently rarely propagated to following generations.

Beside the diversity of solutions, which can be considered a by-product of the search-based nature of BIOFUZZ, we manually looked at the generated individuals to estimate the impact of mutation and crossover. In *geccBBlite*, we manually included a partial sanitization that allowed to ignore HTML and SQL special characters (including AND and OR in uppercase characters). Thanks to mutation operations, BIOFUZZ could generate the input 383549 or 2 in (NOW()), which results in a successful SQLI attack.

SELECT id, rispostadel FROM forum WHERE id = 383549 or 2 in (NOW())

In general, mutation has proved to be essential in presence of partially sanitized parameters. Crossover, instead, proved to improve individuals in several conditions. One situation we frequently observed in our evaluation, and clearly shows the importance of crossover, is when two individuals used different terminals for column names or values. Crossover allows to generate individuals that have more knowledge of the database structure, and as a consequence they have higher chances to successfully generate context-dependent attacks.

Similarly, crossover proves to be useful when one individual identifies tokens that can be successfully injected, and as a consequence to share this solution with worse individuals. Figure 4 illustrates the crossover operation on two individuals *a* and *b* in *Schoolmate*. The individual *b* was able to successfully inject a database function. By crossing over elements with the worse individual *a*, BIOFUZZ would allow the generation of a new individual *c*, combining features of both parents.



```
UPDATE schoolinfo SET schoolname =
"School Name", address='1,Street',
phonenumber = '52365895', sitetext=' ',
sitemessage='Message', numsemesters = '
0 ' , numperiods = ' 0 ' , apoint = ' 0.0 ',
bpoint = '185176 ' , fpoint = NULL
```

```
UPDATE schoolinfo SET schoolname =
" 539682 CONCAT(CHAR(34)) LIMIT 4135
```

```
UPDATE schoolinfo SET schoolname =
"School Name", address='1,Street',
phonenumber = '52365895', sitetext=' ',
sitemessage='Message', numsemesters = '
0 ' , numperiods = ' 0 ' , apoint = ' 0.0 ',
bpoint = '185176 ' , fpoint =
" 539682 CONCAT(CHAR(34)) LIMIT 4135
```

Figure 4: Crossover operation on SQL statements in *Schoolmate*

Table 3: Vulnerabilities found by different tools

	BIOFUZZ		ARDILLA		*sqlmap*
	#Par	#Vul	#Par	#Vul	#Par
WebChess	4	13	4	12	4
Schoolmate	4	6	4	6	2
FaqForge	1	1	1	1	1
geccBBlite	3	4	3	2	3
phpMyAddressbook	4	10	–		1
Elemata	1	3	–		1

Mutation can be helpful in circumventing partially sanitized inputs, and crossover helps sharing partially successful attacks with other individuals.

3.3 RQ2: Comparison against Other Tools

We compared BIOFUZZ with two other state of the art tools that also detect SQLI vulnerabilities: ARDILLA, a white-box approach, and *sqlmap*, a black-box approach.

3.3.1 *White-box Approach—ARDILLA*

ARDILLA is a white-box approach that relies on *taint analysis* to identify likely vulnerable parameters, and generates SQLI attacks based on a set of predefined attack patterns. Since ARDILLA is not publicly available, to compare it with BIOFUZZ we had to consider the same case studies, for which the authors publicly released the results. This, however, leads us to have incomplete results, since we could not evaluate ARDILLA on the two new case studies.

Table 3 shows the results of our comparison. As in Table 2, we list the number of vulnerable POST and GET parameters ("#Par") and different vulnerabilities ("#Vul") detected.

BIOFUZZ could identify all the vulnerabilities that ARDILLA could identify, and in the case of *geccBBlite* and *WebChess* it reported additional ones. These are additional cases of data manipulation (INSERT and UPDATE) vulnerabilities that were not reported by ARDILLA. Although it is not clearly explained in the paper, it might be that ARDILLA does not report injections that alter the database by adding or changing some values unless they can be exploited later on in the code. In other words, it might be that one can inject a quote character into an insert statement that leads to a malformed query, but as long as it is not possible to change sensitive data, this is not considered as a SQLI by ARDILLA. However, in our opinion it is reasonable to consider these cases as SQLI vulnerabilities, mainly because they show that it is likely that the application is exposed to second-order SQLI attacks. Moreover, even if these were not exploitable vulnerabilities, they are for sure a deviation from the intended behavior of the Web application, and should therefore be reported.

BIOFUZZ detected all the vulnerabilities reported by ARDILLA, a white-box technique, and could report additional ones.

3.3.2 Black-box Approach—SQLMAP

sqlmap[7] is the most advanced black-box tool we are aware of. Despite using a set of predefined attack patterns, and a set of heuristics to identify vulnerable parameters, it is quite effective.

We ran *sqlmap* on all the case studies that we analyzed. Similarly to BIOFUZZ, this tool has crawling capabilities, and can automatically detect parameters to exploit with SQLI attacks. *sqlmap*, though, has major issues with the crawling component: For almost all the vulnerabilities we had to manually provide the URL. Moreover, while *sqlmap* was usually quite good at identifying GET parameters automatically, we had to manually specify POST parameters most of the times.

Apart from the crawling problems, *sqlmap* performed quite well. As Table 3 shows[8], *sqlmap* could identify and exploit all the vulnerable parameters that BIOFUZZ could exploit in all the evaluation subjects, except for two of them. In *Schoolmate* it misses two vulnerable input parameters, i.e., `username` and `schoolname`. *sqlmap* missed a vulnerable parameter in *phpMyAddressbook*, as well. Overall, BIOFUZZ outperformed *sqlmap*, as it found more vulnerable parameters; this result is the most remarkable since, unlike *sqlmap*, BIOFUZZ does not come with a fixed set of known attack patterns. However, *sqlmap* also found one vulnerability that BIOFUZZ did not detect: In *phpMyAddressbook*, *sqlmap* generated a successful attack by exploiting the HTTP-Header field `User-Agent`, a feature not present in BIOFUZZ.

*BIOFUZZ also outperformed **sqlmap**, a black-box testing technique using predefined patterns.*

3.4 Limitations

During our evaluation, we ran into some known limitations of Crawljax that could consequently limit the effectiveness of BIOFUZZ. Some states of a Web application can only be reached if the crawler provides specific inputs, and if the database contains some data. This is the case, for instance, of all the web applications that have a login page as the only entry point. Without valid inputs, the crawler would not be able to explore pages beyond the login page. Moreover, some applications such as *Schoolmate* provide different roles (e.g. teacher, student, admin), and in order to explore some states it is required that these roles interact. All these are known problems of crawlers, and while some of these problems could have been avoided by using alternative crawlers that support multiple roles [24], others can be circumvented with little manual efforts.

BIOFUZZ was able to find all vulnerabilities for *FaqForge*, *Elemata* and *geccBBlite* without any manual work. For *WebChess*, *Schoolmate* and *Elemata*, we had to manually create some elements (i.e. new games, new courses, and new posts), such that Crawljax could detect some clickable elements that would have not been visible otherwise. For *phpMyAddressbook*, we had to create the test cases manually, since Crawljax was not able to detect the main menu and therefore could not detect the main states of the application. For this evaluation subject we defined 14 test-cases, one for each of the main functionality *phpMyAddressbook* provides. Since Crawljax does not support different roles, we had to define 4 additional test-cases (login as student, pick a course as student, update course information as teacher, update school information as admin) for *Schoolmate* as well. For *WebChess*, we had to define 1 additional test-case (continue a chess game for two players on the same PC), since Crawljax could not detect it automatically.

In essence, the ability of BIOFUZZ to automatically find all the vulnerabilities in a Web application highly depends on the ability of the crawler to explore the Web application. In absence of a good crawler, BIOFUZZ requires a good set of test cases, which can be easily created with Selenium[9] or similar frameworks, to exercise as many interactions between client and server as possible.

The effectiveness of BIOFUZZ depends on the quality of the model produced by the crawler or by a test suite.

3.5 Threats to Validity

As any empirical study, our evaluation is subject to threats to validity. The most important is *threats to external validity:* As every tester knows, results from a finite sample do not necessarily generalize towards the entire range of possibilities. Hence, the results we obtained from our six Web applications do not necessarily generalize towards all Web applications. Specifically, if a SQLI attack leveraged very specific knowledge about the application to be attacked, a white-box approach such as ARDILLA has better chances to detect the attack; and if the attack is already known, an approach using predefined attack patterns such as *sqlmap* also has an advantage. Furthermore, if SQLI testing tools are widely deployed, the remaining vulnerabilities will be hard to find by construction.

4. RELATED WORK

Most related to our work is the large body of techniques that focus on detecting or preventing SQL injections, and the search-based security testing techniques.

4.1 Detection or Prevention of SQL injections.

Several techniques have been proposed to either *prevent* or *detect* SQLI vulnerabilities in Web applications. Techniques to prevent these kind of vulnerabilities typically resort to sanitize user inputs (i.e., escape special characters such as quotes) before passing them to query statements. Since it is easy to miss unsanitized inputs, some techniques automatically support developers in this task. However, they either statically analyze and change the server side code before deployment [27, 19, 18, 3], or they monitor and sanitize queries at runtime on the deployed system [8, 7, 10, 21, 22, 28]. The first ones require access to the server side code, which is not always available, and are bound to specific programming languages. The second ones, instead, have the disadvantage of incurring overhead, and since they filter queries on the basis of either a set of predefined rules, or on models that were built on observed valid queries, or thanks to machine learning techniques, they may also incur into false positives.

Techniques that detect SQLI vulnerabilities, instead, can be broadly divided into white-box and black-box techniques. White-box techniques usually resort to taint-analyses to identify which user inputs can flow into query statements [13, 12,

[7]http://sqlmap.org

[8]*sqlmap* only detects vulnerable parameters, not vulnerabilities as such.

[9]http://docs.seleniumhq.org

6, 1, 25, 16, 17]. The main limitations of these techniques are that they require access to the server side code, and are also bound to specific programming languages. Black-box techniques such as BIOFUZZ, instead, are less invasive since they do not require access to the server side code, and also work independently from programming languages and DBMSs [11, 14]. However, these techniques can usually detect only the most simple attacks, since they test Web applications using typical attack patterns. Moreover, since they usually rely on simple heuristics to understand whether an input field is vulnerable to SQLI attacks, they often do not detect vulnerabilities when the code uses basic sanitization techniques. Finally, even when they can successfully generate SQLI attacks to expose vulnerabilities, they sometimes may not realize that the attack has been successful, since they rely on other simple heuristics. Bau et al. [2] analyzed several black-box scanners for web applications. They conducted their study based on a testset containing XSS as well as SQL type I and SQL type II vulnerabilities. The average scanner vulnerability detection rate for SQL type I attacks was 21.4%, whereas none of the scanners was able to detect SQL II attacks. This shows that other black-box techniques are significantly less effective than BIOFUZZ.

4.2 Search-based Security Testing

The only other technique we are aware of that makes use of genetic algorithms in the context of SQL injections is by Liu et al. [15]. Their approach is quite different from BIOFUZZ, though, since they aim to prevent SQL injections, and they adopt search-based techniques only to validate user inputs.

Other techniques rely on genetic programming to address different types of vulnerabilities. Rawat et al. and use search-based techniques to find buffer overflows in C programs [23, 5]. They combine static analysis and mutation-based evolutionary strategies. The core idea is to use taint analysis to determine the vulnerable endpoints (e.g. unsafe functions such as `sprintf()`), and then generate inputs that follow the identified path, and observe if the inputs have any negative impact on the program behavior. Similarly, Del Grosso et al. combine static and synamic analysis with evosutionary algorithms to detect buffer overflows. Sparks et al. [26] proposed a vulnerability detection technique that uses a genetic algorithm based on a Dynamic Markov Model fitness heuristic. This technique, although more general than the previous one, also mainly targets buffer overflows.

5. CONCLUSION AND FUTURE WORK

We have presented BIOFUZZ, a search-based black-box security tester for Web applications. To the best of our knowledge, this is the first time evolutionary testing has been used to detect SQL injections. Using black-box testing, BIOFUZZ is easily applicable, as it requires no analysis or instrumentation of server code. By systematically evolving form inputs based on SQL query feedback, BIOFUZZ is highly effective: In our evaluation, it detected more vulnerabilities than ARDILLA, the state of the art white-box analysis tool.

The significance of these results is that evolutionary algorithms clearly belong into the portfolio of security research techniques: Practitioners in security analysis now have another resort in case symbolic or dynamic analysis techniques are not applicable, available, or yield insufficient results. Furthermore, the current state of security is not a question of which single technique to choose, but rather of maximizing the numbers of techniques to apply and intertwine. This spirit of integration is also what will determine our future work:

- **Better crawlers.** BIOFUZZ inherits all limitations of the underlying Web crawler. We are currently investigating alternative crawlers such as *Procrawl,* which supports multiple user roles [24], or *Webmate,* which has extensive support for exploring Web 2.0 applications [4].

- **Symbolic analysis.** If symbolic analysis of server code or some components is available, we can evolve inputs to specifically cover the resulting constraints. The result would be a precise concrete exploit rather than a set of abstract warnings, which would be of great use to developers.

- **Dynamic analysis.** Standard approaches to search-based testing aim for increasing coverage by measuring how close the input gets to flip a branch condition. We can apply similar techniques to specifically search for inputs that flip conditions in sanitizers and input checks—and thus explore boundaries, which are known to be specifically error-prone.

- **Known attacks.** At this point, all our inputs are produced from a SQL grammar, starting with seeded random inputs. As an alternative, we could also start with known inputs for vulnerabilities, and mutating these instead. Such a strategy has shown particularly successful for testing JavaScript interpreters [9].

More information in BIOFUZZ is available at

`http://www.st.cs.uni-saarland.de/testing/`

Acknowledgments. This work was funded by an European Research Council (ERC) Advanced Grant "SPECMATE – Specification Mining and Testing".

6. REFERENCES

[1] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP)*, pages 387–401, Washington, DC, USA, 2008. IEEE Computer Society.

[2] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP)*, pages 332–345, Washington, DC, USA, 2010. IEEE Computer Society.

[3] W. R. Cook and S. Rai. Safe query objects: statically typed objects as remotely executable queries. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 97–106, New York, NY, USA, 2005. ACM.

[4] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. Webmate: Generating test cases for web 2.0. In D. Winkler, S. Biffl, and J. Bergsmann, editors, *SWQD*, volume 133 of *Lecture Notes in Business Information Processing*, pages 55–69. Springer, 2013.

[5] C. Del Grosso, G. Antoniol, E. Merlo, and P. Galinier. Detecting buffer overflow via automatic test input data generation. *Journal Computers and Operations Research*, 35(10):3125–3143, Oct. 2008.

[6] W. G. J. Halfond, S. R. Choudhary, and A. Orso. Penetration testing with improved input vector identification. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 346–355. IEEE Computer Society, 2009.

[7] W. G. J. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing SQL-injection attacks. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 174–183. ACM, 2005.

[8] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 175–185, New York, NY, USA, 2006. ACM.

[9] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[10] Y.-W. Huang, S.-K. Huang, T.-P. Lin, and C.-H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *International World Wide Web Conference (WWW)*, pages 148–159, New York, NY, USA, 2003. ACM.

[11] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *International World Wide Web Conference (WWW)*, pages 247–256, New York, NY, USA, 2006. ACM.

[12] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 105–116, New York, NY, USA, 2009. ACM.

[13] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 199–209. IEEE, 2009.

[14] Y. Kosuga, K. Kono, M. Hanaoka, M. Hishiyama, and Y. Takahama. Sania: Syntactic and semantic analysis for automated testing against SQL injection. *Computer Security Applications Conference, Annual*, 0:107–117, 2007.

[15] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou. Sqlprob: a proxy-based architecture towards preventing SQL injection attacks. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC)*, pages 2054–2061, New York, NY, USA, 2009. ACM.

[16] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[17] M. Martin and M. S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium (SS)*, pages 31–43, Berkeley, CA, USA, 2008. USENIX Association.

[18] R. A. McClure and I. H. Krüger. SQL DOM: compile time checking of dynamic SQL statements. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 88–96, New York, NY, USA, 2005. ACM.

[19] E. Merlo, D. Letarte, and G. Antoniol. Automated protection of php applications against SQL-injection attacks. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSRM)*, pages 191–202, Washington, DC, USA, 2007. IEEE Computer Society.

[20] A. Mesbah, A. van Deursen, and S. Lenselink. Crawling Ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web (TWEB)*, 6(1):3:1–3:30, 2012.

[21] A. Nguyen-tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[22] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the international conference on Recent Advances in Intrusion Detection (RAID)*, pages 124–145, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] S. Rawat and L. Mounier. Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 531–533, Washington, DC, USA, 2011. IEEE Computer Society.

[24] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *European Software Engineering Conference held jointly with ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 422–432. ACM, 2013.

[25] Y. Shin, L. Williams, and T. Xie. Sqlunitgen: SQL injection testing using static and dynamic analysis. In *The 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006)*, 2006.

[26] S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 477–486, 2007.

[27] S. Thomas, L. Williams, and T. Xie. On automated prepared statement generation to remove SQL injection vulnerabilities. *Journal of Information and Software Technology*, 51(3):589–598, Mar. 2009.

[28] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of SQL attacks. In *Proceedings of the Second international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 123–140, Berlin, Heidelberg, 2005. Springer-Verlag.

[29] Webchess web application, visited in October 2013. http://sourceforge.net/projects/webchess.