SEPTIC: Detecting Injection Attacks and Vulnerabilities Inside the DBMS

Ibéria Medeiros, *Member, IEEE,* Miguel Beatriz, Nuno Neves, *Member, IEEE,* and Miguel Correia, *Senior Member, IEEE*

Abstract—Databases continue to be the most commonly used backend storage in enterprises, but they are often integrated with vulnerable applications, such as web frontends, that allow injection attacks to be performed. The effectiveness of such attacks stems from a semantic mismatch between how SQL queries are believed to be executed and the actual way in which databases process them. This leads to subtle vulnerabilities in the way input validation is done in applications. We propose SEPTIC, a mechanism for DBMS attack prevention, which can also assist on the identification of the vulnerabilities in the applications. The mechanism was implemented in MySQL and evaluated experimentally with various applications and alternative protection approaches. Our results show no false negatives and no false positives with SEPTIC, on the contrary to other solutions. They also show that SEPTIC introduces a low performance overhead, in the order of 2.2%.

Index Terms—Injection attacks, DBMS self-protection, security, software security.

1 Introduction

Web applications have been around for more than two decades and are now an important component of the economy, as they often serve as an interface to various business related activities. Databases continue to be the most commonly used backend storage in enterprises, and they are often integrated with web applications. However, web applications can have vulnerabilities, allowing the data stored in the databases to be compromised.

SQL injection attacks (SQLI), for example, continue to rise in number and severity [3], [15], [31]. Commonly used defenses are validation functions, web application firewalls (WAFs), and prepared statements. The first two inspect web application inputs and sanitize those that are considered dangerous, whereas the third bounds inputs to placeholders in the SQL queries[1]. Other anti-SQLI mechanisms have been developed but less adopted. Some of these monitor and block SQL queries that deviate from specific models, but the inspection is made without full knowledge about how they are processed by the DBMS [7], [8], [18], [28], [42]. In all these cases, developers and system administrators make assumptions about how the server-side scripting language and the DBMS work and interact, which sometimes are simplistic while in others are blatantly wrong. For example, programmers usually assume that the PHP function `mysql_real_escape_string` always effectively sanitizes inputs and prevents SQLI attacks, which is not true. Also, they often assume that values retrieved from a database do not need to be validated before being inserted in a query, leading to second order injection vulnerabilities. This is visible when, for instance, the code `admin' --` is sanitized by escaping the prime character before sending it to the database, but the DBMS unsanitizes it before actually storing it. Later, the code is retrieved from the database and used unsanitized in some query, carrying out the attack.

Such simplistic/wrong assumptions seem to be caused by a *semantic mismatch* between how a SQL query is expected to run and what actually occurs when it is executed (e.g., the programmer expects it to be sanitized but the DBMS unsanitized it). This mismatch may lead to vulnerabilities, as the protection mechanisms may be ineffective (e.g., they may miss some attacks). To avoid this problem, SQLI attacks could be handled *inside*, after the server-side code processes the inputs and the DBMS validates the queries, reducing the amount of assumptions that are made. The mismatch and this solution are not restricted to web applications, meaning that the same problem can be present in other business applications. In fact, injection attacks are a generic form of attack, transversal to all applications that use a database as backend.

This idea of handling attacks *inside* has been quite successful in the realm of binary applications, to stop attacks irrespectively of the developers ability to follow secure programming practices or not. In that case, *inside* means that protection mechanisms are inserted in programming libraries or operating systems. Examples include address space layout randomization (ASLR), data execution prevention (DEP), or canaries/stack cookies [20], [24].

In this paper, we propose a similar idea for applications backed by databases. We propose to block injection attacks *inside* the DBMS at runtime. We call this approach *SElf-ProtecTIng databases from attaCks* (SEPTIC). The DBMS is an interesting location to add protections against such attacks because it has an unambiguous knowledge about what will be considered as clauses, predicates and expressions of a

- *I. Medeiros and N. Neves are with LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal (e-mail: imedeiros@di.fc.ul.pt and nuno@di.fc.ul.pt).*
- *M. Beatriz and M. Correia are with INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal (e-mail: miguel.beatriz@tecnico.ulisboa.pt and miguel.p.correia@tecnico.ulisboa.pt).*

Manuscript received February XXX, 2018; revised March XX, 2018.

1. We use the term *SQL query*, or simply *query*, to designate any SQL statement (e.g., SELECT, INSERT).

SQL statement. No mechanism that actuates outside of the DBMS has such knowledge.

We address two categories of database attacks: *SQL injection attacks*, which continue to be among those with highest risk [17] and for which new variants continue to appear [37]; and *stored injection attacks*, including stored cross-site scripting, which also involve SQL queries. For SQLI, we propose to catch the attacks by comparing queries with query models, improving an idea that has been previously used only outside of the DBMS [7], [8], [18], [42], and by comparing queries with validated queries with a similarity method, improving detection accuracy. For stored injection, we employ plugins to deal with specific attacks before data is inserted in the database.

SEPTIC relies on two new concepts. Before detecting attacks, the mechanism can be trained by forcing calls to all queries in an application. The result is a set of query models. However, as training may be incomplete and not cover all queries, we introduce the notion of putting in *quarantine* queries at runtime for which SEPTIC has no query model. The second concept, *aging*, deals with updates to query models after a new release of an application, something that is inevitable in real world software. Both concepts allow a reduction of the false negative (attacks not detected) and false positive (alerts for non-attacks) rates.

We demonstrate the approach with a common deployment scenario: MySQL, probably the most popular open-source DBMS [38], and PHP, the language most used to build web applications (more than 80%) [45]. We also explore Java/Spring, the second most employed programming language, and the Gambas language, used to develop many business applications. SEPTIC is evaluated experimentally to assess its effectiveness to block attacks, including in the tests a set of non-trivial SQLI attacks [36], [37]. SEPTIC is also compared with a number of alternative solutions, including the ModSecurity WAF and recent anti-SQLI mechanisms proposed in the literature, with SEPTIC showing neither false negatives nor false positives, on the contrary of the others. The impact of our approach on the performance of MySQL is analyzed by running BenchLab [10]. The experiments give evidence of very low overheads, around 2.2%.

2 DBMS Injection Attacks

As we stated before, we denominate *semantic mismatch* as an incorrect perception about how the SQL queries are executed by the DBMS – the developer expects queries to be processed in a certain way but they are actually run in a different manner. This mismatch often leads to mistakes in the implementation of protections in the source code of applications, making these vulnerable to SQL injection and other attacks involving the DBMS. The problem is subjective in the sense that it depends on the programmer, but some mistakes are usual. A common way to try to prevent SQLI consists in sanitizing user inputs before they are used in SQL queries.

The PHP function `mysql_real_escape_string`[2], for instance, precedes special characters (like prime or double prime) with a backslash, transforming these delimiters into normal characters. However, sanitization functions do not behave as envisioned when the special characters are represented differently from expected, e.g., `'` (prime) is encoded as `%27`. In such case, the DBMS decodes and executes the queries with the prime character. We identified several DBMS injection attacks in the literature, including a variety of cases related to semantic mismatch [11], [13], [14], [29], [36], [37], [39]. Table 1 classifies these attacks. The first three columns identify the classes, whereas the fourth and fifth explain how the PHP sanitization functions and the DBMS process the example malicious inputs of the sixth column.

As mentioned in the introduction, we consider two main classes of attacks: *SQL injection* and *stored injection* (first column). These classes are divided in sub-classes corresponding to common designations of attacks targeting the DBMS, namely, classes A to E for the former and class F to H for the latter. However, class E might also fit on the latter class of attacks. Classes S.1 and S.2 are related with classes A to E and separate the attacks based on the way they affect the syntactic structure of the SQL query. Class S.1 is composed of attacks that modify this structure, while class S.2 encompasses attacks that change the query but mimic its original structure.

```
1 $user = mysql_real_escape_string($_POST['username']);
2 $pass = mysql_real_escape_string($_POST['password']);
3 $query = "SELECT * FROM users WHERE username='$user' AND
      password='$pass'";
4 $result = mysql_query($query);
```

Listing 1: Script vulnerable to SQLI with encoded characters.

Class A – obfuscation – contains five subclasses that represent cases of semantic mismatch. As an example, consider the code excerpt in Listing 1 implementing a login script that checks the user credentials (username, password) in the database.[3] Both user inputs are sanitized by the `mysql_real_escape_string` function (lines 1-2) before inserting them in the query (line 3) and submitting the request to the DBMS (line 4). If an attacker injects the `admin'--` string as username (line 1), the `$user` variable receives this string sanitized, with the prime character preceded by a backslash. The user `admin\'--` does not exist in the database, and so this SQLI attack is not successful.

On the contrary, this sanitization is ineffective if the input uses URL encoding [6], leading to an attack of class A.1. Imagine that the attacker inserts the same username URL-encoded: `%61%64%6D%69%6E%27%2D%2D%20`. `mysql_real_escape_string` function does not sanitize the input because it does not recognize `%27` as a prime. However, MySQL receives that string as part of a query, and decodes it, thus executing `SELECT * FROM users WHERE username='admin'-- ' AND password='foo'`. The attack is effective because this query is equivalent to `SELECT * FROM users WHERE username='admin'` (no password has to be provided as the two characters `--` indicate that the rest of the code in the line should be ignored). This is also an attack of class S.1 as the structure of the query is

2. Notice that PHP 7 recommends the use of function `mysqli_real_escape_string` to escape strings. This function modifies strings in the same manner as `mysql_real_escape_string`, and therefore leads to the same problems as the latter.

3. All examples included in the paper were tested with Apache 2.2.15, PHP 5.5.9 and MySQL 5.7.4

TABLE 1: Classes of attacks against DBMSs.

Class		Class name	PHP sanit. func.	DBMS	Example malicious input
SQL injection	**A**	Obfuscation			
	A.1	- Encoded characters	do nothing	decodes and executes	`%27, 0x027`
	A.2	- Unicode characters	do nothing	translates and executes	`U+0027, U+02BC`
	A.3	- Dynamic SQL	do nothing	completes and executes	`char(39)`
	A.4	- Space character evasion	do nothing	removes and executes	`char(39)/**/OR/**/1=1--`
	A.5	- Numeric fields	do nothing	interprets and executes	`0 OR 1=1--`
	B	Stored procedures	sanitize	executes	`admin' OR 1=1`
	C	Blind SQLI	sanitize	executes	`admin' OR 1=1`
	D	Insert data	sanitize	unsanitizes and executes	`admin' OR 1=1--`
	E	Second order SQLI	–	executes	any of the above
St inj	**F**	Stored XSS	–	–	`<script>alert('XSS')</script>`
	G	Stored RCI, RFI, LFI	–	–	`malicious.php`
	H	Stored OSCI	–	–	`; cat /etc/passwd`
	S.1	Syntax structure	sanitize	executes	`admin' OR 1=1`
	S.2	Syntax mimicry	sanitize	executes	`admin' AND 1=1--`

XSS: Cross-Site Scripting; RCI: Remote Code Injection; RFI:Remote File Inclusion; LFI: Local File Inclusion; OSCI: OS Command Injection

modified as the part that checks the password disappears. The other subclasses of A involve alternative masquerading techniques. In class A.2, the attacker encodes some characters in Unicode (e.g., the prime as `U+02BC`). In class A.3, a function is inserted and called dynamically (e.g., the prime is encoded as `char(39)`). Class A.4 uses spaces and equivalent strings to manipulate queries (e.g., concealing a space with a comment like `/**/`) [11]. In classes A.3, A.4, and A.5, the DBMS decodes the obfuscated code before executing the query. Class A.5 abuses the fact that numeric fields do not require values to be enclosed with primes, and therefore a tautology can be created without these characters (similar to the example for A.1), fooling sanitization functions like `mysql_real_escape_string`.

Class B – stored procedures – could be exploited in a similar way as queries constructed in the application code. These procedures may take inputs that modify or mimic the syntactic structure of the query, leading to attacks of classes S.1 or S.2. Class C – blind SQLI attacks – aims to extract information from the database by observing how the application responds to different inputs. These attacks may also fall in classes S.1 or S.2.

Class D – insert data – aims to add crafted data to the database (`INSERT`, `UPDATE`), so that later it can be retrieved and used in another query of the application. This class of attack is another case of semantic mismatch and it is the base of stored injection attacks (see next classes). For example, if an attacker provides the `admin' OR 1=1--` string, then it might be sanitized with `mysql_real_escape_string` in the application. However, once the string reaches the DBMS, the input will be unsanitized before being saved in the database. These attacks may fall in classes S.1 or S.2.

Classes E to H – stored injection – are characterized by being executed in two steps: the first involves doing an SQL query that inserts attacker data in the database; the second uses this data to complete the attack. The specific attack depends on the data and how it is used. In a second order SQLI attack (class E), the data is a string specially crafted to be included in another SQL query, which is then executed in the second step. This second query is the attack itself and it may fall in classes S.1 or S.2. This is another case of semantic mismatch as the sanitization created by functions like `mysql_real_escape_string` is removed by the DBMS when the string is put in the database (first step of the attack – class D). A stored XSS (class F) involves placing a script (typically JavaScript) in the database in the first step, and then returning it to the browser of one or more users in the second step. The automatic execution of the script at the client causes some malicious action to be performed. In class G the data inserted in the database can be a malicious PHP script or an URL of a website containing such a script, resulting in a local or remote file inclusion, or on remote code injection. In class H, the attack inserts data in an operating system command, which is executed in the second step.

3 SEPTIC Approach and Architecture

SEPTIC is implemented by a module inside the DBMS, allowing every query to be checked for attacks. The semantic mismatch problem is circumvented because queries are evaluated for detection purposes near the end of the DBMS data flow, just before the query is executed. As SEPTIC is inside the DBMS, it is independent from the application (e.g., from the application programming language) and the way it builds queries (e.g., dynamically). This lets SEPTIC analyze queries issued by any kind of application. However, with support from the application, SEPTIC can also contribute to the identification of the vulnerabilities that are being exploited (Section 7).

3.1 Approach overview

Figure 1 shows the architecture of a web and a non-web application with a backend database. When the system starts, SEPTIC may undergo a training phase in order to obtain the query models for the application. We designate *administrator* the person or persons in charge of managing the DBMS and SEPTIC (e.g., decides when the training mode ends).

Later on, when SEPTIC is put in normal operation, it works basically in the following way:

1) An application requests the execution of a query. Optionally, the query instruction may contain an (external) identifier produced by the server-side language engine or the application;

2) The DBMS receives, parses and validates the query. Before it executes the query, SEPTIC is called to retrieve its associated query model, which is used to detect and block a potential incoming attack. If an external identifier arrives with the query, it is extracted to get context information about the places in the source code of the application where the query was built. This information can be helpful to locate a vulnerability in case an attack is found.

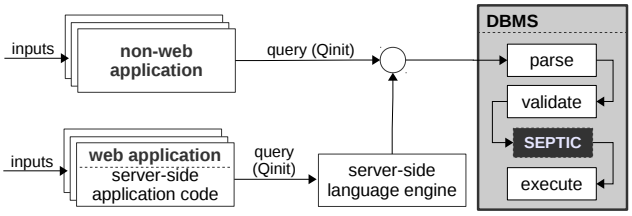


Fig. 1: Two kinds of applications backed by a DBMS with SEPTIC.

3.2 Architecture

SEPTIC runs in three modes, one for training during the set up of the system (*training mode*) and two during normal operation (either *prevention* or *detection mode*). Figure 2 displays the various steps carried out by SEPTIC. The figure should be read starting from the black arrow at the top/left. Dotted-dashed arrows and processes represent the training mode, whereas solid arrows and processes represent common operations of normal mode for both prevention and detection modes. Thin dotted arrows and processes represent alternative paths for prevention mode, while double-solid arrows represent detection mode.

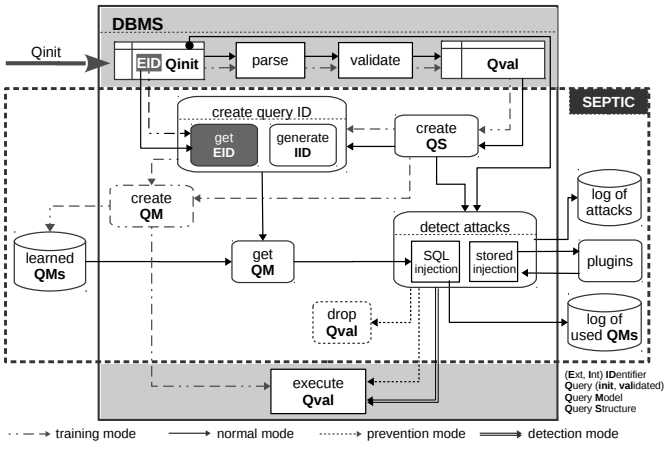


Fig. 2: Architecture and data flows of SEPTIC.

The query execution request received by the DBMS is called the *initial query* (Qinit), while the query resulting after the internal processing (parsing and validation) is named *validated query* (Qval). SEPTIC operates mainly with Qval but also resorts to Qinit in some cases.

Training is done by putting SEPTIC in *training mode* and by running the application for some time without attacks (Section 6.1). Training creates a set of *query models* (QMs), each one associated with a *query identifier* (ID), and saves this data in the *learned QMs* data store (Section 6). ID is composed by an *internal query identifier* (IID) generated by SEPTIC, optionally complemented with an *external query identifier* (EID grey box in Figure 2) provided by the application (Section 4.2).

In normal operation, SEPTIC generates a *query structure* (QS) and an IID for every arriving request. The IID is used to build the ID. In addition, if an EID is embedded in Qinit, it is obtained and included in ID. SEPTIC enforces attack detection firstly by comparing the QS with the QM that was previously learned for that ID and secondly by looking for disparities between the QS and the Qinit. A SQLI attack is found if there is no match. Otherwise, SEPTIC runs a set of *plugins* that look for specific stored injection problems. Queries deemed valid are allowed to proceed with the DBMS processing, but before SEPTIC logs information about the QM that matched the QS.

The action that is taken when an attack is found depends on the mode of execution. In *prevention mode*, the attack is aborted, i.e., the query is dropped to interrupt processing. In *detection mode*, queries are run. In both modes, SEPTIC logs information about the attacks that were caught.

4 Query Representations and Identifiers

SEPTIC processes queries validated by the DBMS and represents them by query structures (QSs) and query models (QMs), depending if it executes in normal operation (prevention or detection mode) or in training mode. Also, each query model is known by a query identifier (ID). Therefore, the core of SEPTIC relies on queries, their representations and identifiers. This section presents detailed information about them.

4.1 Queries, query structures and query models

A query is an SQL statement to be executed by a DBMS. A query is composed by a set of elements, namely SQL clauses, fields, operators and functions that act on data. In an application, the great majority of these elements are fixed, and the exceptions are the data fields that contain inputs dynamically set by the applications (e.g., based on user provided data).

Applications typically handle a query simply as a string, since they have no need to separate or distinguish the elements of the SQL statement (i.e., elements are just parts of the string)[4]. On the other hand, SEPTIC needs more information about these elements to be able to make the various comparisons between the QS / QM / Qinit. Fortunately, the DBMS also requires that information, and assigns each of the queries' elements (clause, field, etc.) to a category. Therefore, from the point of view of SEPTIC, a query is a SQL statement sent by an application, which it can analyse with the same level of detail as the DBMS.

Arriving queries are parsed and validated by the DBMS before they are executed. Qinit is received in the form of a string and suffers several modifications until it becomes Qval. Namely, it is parsed, the SQL syntax is checked,

4. An exception occurs with prepared statements.

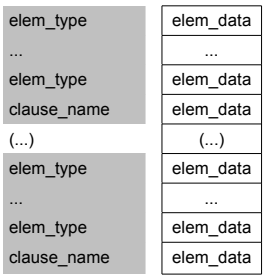


Fig. 3: A generic query structure.

the comments are removed, and encoded characters are decoded. The query is finally executed iff no error is found.

SEPTIC assumes that Qval is in the form of a parse tree, represented as a *list of stacks* data structure, which is the usual way to maintain queries internally to the DBMS [5]. Every stack of the list corresponds to a clause (e.g., `SELECT`, `FROM`, `WHERE`) or statement (e.g., `INSERT`, `UPDATE`) of the query, and each of their nodes contains information about a query element, such as category/type (e.g., field, function, operator), data type (e.g., integer, string), and data value (i.e., the value itself). Table 2 presents examples of these elements that may compose a query.

TABLE 2: Examples of elements that can compose a query.

Clause/Statement	Element		Data type
	Category	Data	
SELECT	operator	+, -, between, like	integer
FROM	condition	and, or, not	real
WHERE	field	field_name, table_name	string
ORDER BY	function	char, average, sum	
GROUP BY			
DELETE			
UPDATE			
INSERT			

The *query structure* (QS) of a query is constructed by merging the content of all stacks in the list into a single stack. Figure 3 depicts a generic QS, showing from bottom to top the clauses and their elements. In the figure, each row represents a clause of the query or a query element. Clauses have a name and data: ⟨CLAUSE_NAME, ELEM_DATA⟩. An element of the query is represented by the element type and the element data: ⟨ELEM_TYPE, ELEM_DATA⟩. The single exception is the alternative format ⟨DATA_TYPE, DATA⟩ that represents an input value inserted in the query (DATA) and its (primitive) data type (DATA_TYPE). A part of the query is considered to be an input if its type is primitive (e.g., a string or an integer) or if it is compared to something in a predicate. For the clauses with conditional expressions (e.g., `WHERE`) the elements are inserted in the QS by doing a post-order traversal of the parse tree of the query (i.e., the left child is visited and inserted in the stack first, then the right child, and so on until the root). QS also contains a label with the main SQL clause (`SELECT`, `DELETE`, `UPDATE`,...) to easily identify the type of the query. This label is designated as the SQL command.

As mentioned in the previous section, in training mode SEPTIC creates the query models (QMs). It builds a QM whenever the DBMS processes a query, but the model is only stored the first time the associated query ID is observed. The QM is the query without input data and it is constructed using the QS. The process consists simply in substituting DATA (input data) by a special value $\perp$ in all ⟨DATA_TYPE, DATA⟩ nodes. This allows representing any input data independently of its content and length, since benign inputs do not alter the query model. On the other hand, the nodes without this special value are those that represent the static part of the query. Moreover, this special value is used to denote that these fields should not be compared during attack detection since their content can be different for each query received by SEPTIC for the same QM. In contrast, all the other nodes are identical in the QM and the QS, and so they must be compared during the attack detection (Section 5).

Take as example the query `SELECT name FROM users WHERE user='alice' AND pass='foo'`. Figure 4 represents its (a) parse tree, (b) QS and (c) QM. In Figure 4(b) and (c) the gray items at the bottom have the clauses `SELECT`, `FROM` and `WHERE`. In Figure 4(b) the user input values are represented in bold and in Figure 4(c) they have the special value $\perp$ as explained. In the left-hand column, each element of the query takes a category (field, data type, condition operator, etc.), whereas the right-hand column has the query's keywords, variables and primitive data type values. Notice that primitive data type elements (real, integer, decimal and string) also take a specific category, such as STRING_ITEM (e.g., in the fourth row).

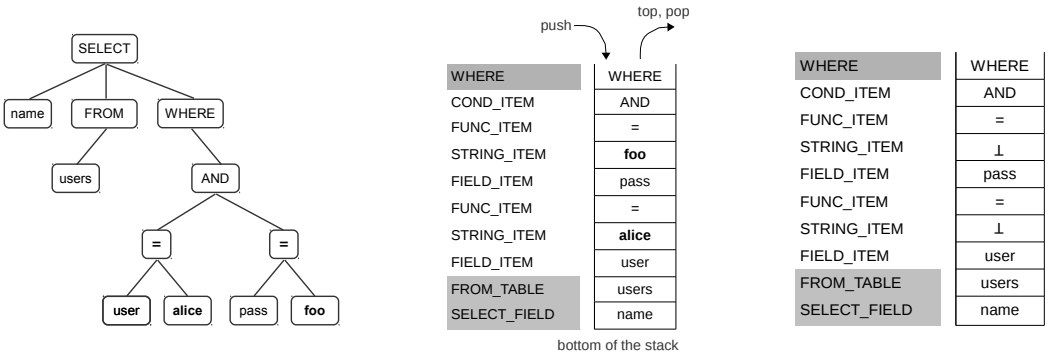
Remark 1. SEPTIC processes any query that reaches the DBMS, after it is parsed and validated. This means that the DBMS is the component that handles the complexity of queries, which can be simple (e.g., the usual `SELECT` statements) or complicated (e.g., queries containing several parameters and sub-queries, including aggregated functions). Consequently, it is the DBMS, not SEPTIC, that performs the potentially hard job of identifying the different elements of queries and representing them as stacks. SEPTIC, for its part, does not need to deal with such difficulties, since it receives the stacks, leverages from the query element identification and categorization to construct QSs and QMs. Therefore, SEPTIC deals with complex queries, but the initial part of the processing is offloaded to the DBMS.

4.2 Identifiers

Query identifiers (IDs) are used to match queries with their models. They are opaque, i.e., their structure is not relevant for SEPTIC, but the information that they carry lets them identify queries uniquely. SEPTIC always generates an *internal query identifier* (IID) inside of the DBMS for every query it receives. However, it can also handle other kinds of identifiers, passed from the outside (the *external query identifier*, EID). In this case, SEPTIC appends its own IID to the EID in order to compose a single query identifier, the ID (otherwise, the ID is the IID).

4.2.1 Internal query identifiers

The DBMS is arguably the best place to create an identifier for transparency, as programmers/administrators can remain oblivious to their existence. Since the QM captures the unique characteristics of a query (e.g., clauses, data elements and data values), SEPTIC leverages this fact to produce



(a) Query parse tree (b) Query structure (c) Query model

Fig. 4: Representation of a query as a parse tree, a structure (QS) and a model (QM).

distinct IIDs. In training mode, a model is constructed and the related IID is calculated for every new query that will be monitored. In normal operation, when a query with a query model already known by SEPTIC is issued, it will have the same IID and QM. This allows queries to be compared against the original QM without confusion. Also, this means that similar queries created at different places of the application source code will be compared against the same model.

The format of the IID can be any that represents the QM and captures the characteristics that are considered relevant. In our current implementation, the IID is formed by the database name concatenated with the SQL command (e.g., `SELECT`), the SQL keywords for `ELEM_DATA` and `DATA` elements (in the second column of Fig. 3), and the `CLAUSE_NAME` of the QM (in the first column of Fig. 3). For the query example presented in Figure 4, and considering that the database name is DB, the IID is `DB_SELECT_WHEREAND=⊥pass=⊥userWHERE usersFROM_TABLEnameSELECT_FIELD`.

4.2.2 External query identifiers

This kind of identifier is produced outside the DBMS, for example in the server-side language engine (SSLE) or in the source code of the application (see Figure 1). It can have an arbitrary value. For instance, it can contain information about the places of source code where the query is composed and/or is issued to the database. The EID is transmitted with the Qinit (see Section 8).

4.2.3 Query identifiers

An ID is built by joining the EID with the IID, in case the Qinit has an EID; otherwise, it is just the IID. This combination of identifiers is interesting because the EID can describe the query inside the application, whereas the IID can find the model of a query for a given database. In this way, it is possible to have identifiers that provide contextual information and that are directly related to queries in a singular way.

5 Injection Attack Detection

This section explains how SEPTIC discovers ongoing attacks. This is achieved by dividing the classes of Table 1 in two groups that are processed differently: SQL injection and stored injection.

5.1 SQLI detection

SQLI attacks are detected by finding out if queries fall in either class S.1 or S.2. These classes are called *primordial for SQL injection* because any SQLI attack belongs to one of them. The rationale is that if an SQLI attack neither modifies the query structure (class S.1) nor changes the query mimicking the structure (class S.2), then it must leave the query unmodified, i.e., it is not an SQL injection attack.

SEPTIC detects the attacks by checking Qval with the associated query model *structurally* (for class S.1) and *syntactically* (for class S.2), plus handling the case of non-unique IDs by comparing Qval with Qinit (*query similarity*). An attack is flagged if there are differences in any of these tests.

```
1 num_nodes_QS <- get number of nodes of QS
2 num_nodes_QM <- get number of nodes of QM
3
4 if num_nodes_QS <> num_nodes_QM then
5   return report a SQLI attack
6 else
7   foreach node_QS in QS and node_QM in QM do
8     if node_QS <> node_QM then
9       return report a SQLI attack
10    end
11  end
12  elements <- null
13  foreach node_QS in QS do
14    if node_QS is a clause_node then
15      if elements is not null then
16        if elements in [a..z, A..Z, comment tokens] then
17          return report a SQLI attack
18        else
19          elements <- get string elements from Qinit
20        end
21      else
22        elements <- get string elements from Qinit
23      end
24    else
25      elem_data <- get elem_data from node_QS
26      remove elem_data from elements
27    end
28  end
29 end
```

Listing 2: Algorithm for detecting SQLI attacks.

The attack detection algorithm (Listing 2) performs these three tests in that order, ending when any of them fails, i.e., when any test detects an attack. Therefore, given a Qval with a certain ID and the corresponding QS, detection involves iterating over the nodes of QS and matching them with the

ones of the stored QM (for that ID). Something equivalent also has to be done with the Qinit:

1) *Structural verification*: if the number of nodes in QS is different from the number of nodes in QM, then Qval does not correspond to the model and detection for QM ends (lines 1 to 5).
2) *Syntactical verification*: if the `ELEM_TYPE` and `DATA_TYPE` of any of the nodes of QS is different from the ones in QM for the same position (except primitive types), then Qval does not match the model and detection for QM ends (lines 6 to 11). Nodes are compared starting at the top and going down the QS and QM stacks as represented in Figures 4(b)–(c). Primitive data types (real, integer, decimal and string) are an exception because DBMSs implicitly make type-casting between them (e.g., integer to string), so these types are considered equivalent.
3) *Query similarity verification*: if any item (string element) of Qinit is distinct from the nodes of QS, then this disagreement causes an attack to be flagged (lines 12 to 28). In detail, the test is executed in this way: i) in QS, SEPTIC identifies the clause appearing in the first place from the top to bottom of the stack; ii) in Qinit, it extracts the string elements corresponding to that clause; iii) for each node of QS from that clause, SEPTIC gets its `ELEM_DATA` and removes it from the extracted string elements; iv) after removing all the `ELEM_DATA`, if the reminiscent string elements contain any word or comment tokens (e.g., `--`, `#`, `/**/`) an attack is flagged, otherwise the process is repeated for the next clause in QS.

There is no attack if all checks are valid. Otherwise, there is an attack and in such case the action to be taken depends on the mode in which SEPTIC is running: in prevention mode the query processing is aborted; in detection mode the query is executed.

Remark 2. The *query similarity verification* avoids the following undesirable situation. A malicious query arrives and after the validation the corresponding Qval (thus also the QS) is equal to one of a benign query already stored by SEPTIC. In this case, QS and QM would match structurally and syntactically, causing the attack to go unnoticed. The query similarity verification avoids this problem by comparing the validated query with the initial query without any processing. This means that the elements on the malicious query that were removed in the validation process (e.g., the commented elements) are noticed by the test because they no longer appear in the query structure.

Example 1. Consider a query `SELECT name FROM users WHERE user=? AND pass=?` where question marks represent inputs. Figure 4(c) illustrates the corresponding QM. Imagine a second-order SQLI attack carried out in the following steps: (i) a malicious user provides an input that leads the application to insert `adminU+02BC--` in the database (i.e., `admin'--` with the prime represented in unicode as `U+02BC`); (ii) later this data is retrieved from the database and inserted in the `user` field in the query above; (iii) the DBMS parses and validates the query, decoding `U+02BC` into the prime character; the resulting query `SELECT name FROM users WHERE user= admin` falls in class S.1 as it modifies the structure of the query. Figure 5(a) presents the QS for this query. SEPTIC compares the QS with the QM and during structural verification observes that they do not match, as the number of nodes of both structures is different, enabling the attack detection.

WHERE	WHERE
FUNC_ITEM	=
STRING_ITEM	**admin**
FIELD_ITEM	user
FROM_TABLE	users
SELECT_FIELD	name

(a) Structural attack

WHERE	WHERE
COND_ITEM	AND
FUNC_ITEM	=
INT_ITEM	**1**
INT_ITEM	**1**
FUNC_ITEM	=
STRING_ITEM	**admin**
FIELD_ITEM	user
FROM_TABLE	users
SELECT_FIELD	name

(b) Mimicry attack

Fig. 5: QSs resulting from a structural and a mimicry attack.

Example 2. Consider a syntax mimicry attack, the query from the previous example and the malicious input `admin' AND 1=1--` inserted as `user`. The resulting query is `SELECT name FROM users WHERE user= admin AND 1=1`. Figure 5(b) represents its QS. SEPTIC compares the QS with the QM (Figures 5(b) and 4(c)). First, during structural verification it observes that they match, as the number of nodes of both structures is equal; then during syntactical verification it sees that the ⟨INT_ITEM, 1⟩ nodes from QS (fourth and fifth rows in Figure 5(b)) do not match with the ⟨STRING_ITEM, ⊥⟩ and ⟨FIELD_ITEM, PASS⟩ nodes from QM (Figure 4(c)), respectively. Although the first test passes, as casting between data types is allowed, the second comparison is considered invalid and the attack is flagged.

Example 3. Regarding query similarity verification, consider an application that has two kinds of users: administrators and normal users. The application also has two different queries to validate the two types of users. Suppose also that SEPTIC stores the QMs for these queries, namely `SELECT name FROM users WHERE user='bob'` (for administrators) and `SELECT name FROM users WHERE user='alice' AND pass='foo'` (for normal users). Later on, the DBMS receives the query (Qinit) `SELECT name FROM users WHERE user='admin'-- ' AND pass='foo'`, which denotes an attempt of a normal user to get access as administrator. The resulting Qval for this query is `SELECT name FROM users WHERE user='admin'`. Since SEPTIC has a similar QM, there is a match for QS and QM. Next, the similarity verification test is applied as explained above, i.e., i) the `WHERE` clause is identified in QS; ii) the `user='admin'-- ' AND pass='foo'` string elements are extracted from Qinit; iii) the `ELEM_DATA` from the `WHERE` clause in QS (`user` and `'admin'`) are removed from the extracted string elements; iv) at the end, there are string elements `-- ' AND pass='foo'` that remain. Therefore, the query similarity verification step of the algorithm detects that the query is an attack.

5.2 Stored injection detection

Stored injection attacks are carried out in two steps. First, some malicious data is inserted in the database; second, that data is taken from the database and is used in some erroneous way. For example, consider a stored XSS (class F) where the data includes a bad script. In the first step, the script is saved, while in the second it is obtained from the database and placed in a web page to be returned to a browser. These attacks cannot be detected as SQLI attacks because they do not work by modifying queries.

SEPTIC detects stored injection attacks in their first step, i.e., it searches for malicious data included in queries that insert data in the database (i.e., `INSERT` and `UPDATE`), and then tests these data with plugins. Therefore, a set of *plugins* is used for this task, typically one for each type of attack. The plugins analyze the queries searching for code that might be executed by browsers (JavaScript, VB Script), by an operating system (shell script, OS commands, binary files) or by server-side applications (php). Since running the plugins may introduce some overhead, a preliminary check is done by the detection algorithm. The algorithm, therefore, works in two steps:

1) *Filtering* – looks for suspicious strings such as: <, >, `href`, and *javacsript* attributes (F); protocol keywords (e.g., `http`) and extensions of executable or script files (e.g., *exe, php*) (G); special characters (e.g., ; and |) (H). If none is found, detection ends.
2) *Testing* – consists in passing the input to the proper plugin for inspection. For example, if the filtering phase finds the `href` string, the data is provided to a plugin that detects stored XSS attacks. This plugin inserts the input in a simple HTML page with the three main tags (`<html>`, `<head>`, `<body>`), and then calls an HTML parser to determine if other tags appear in the page indicating the presence of a script.

Example 4. Consider a web application that registers new users and that a malicious client inserts as his first name the JavaScript code `<script>alert(''Hello!'');</script>`. When SEPTIC receives the query, it does the filtering step and finds two characters associated with XSS, < and >, so it calls the plugin that detects stored XSS attacks. This plugin inserts this input in a web page, calls an HTML parser, and finds that the input contains a script. Thus, it flags a stored XSS attack.

6 Learning query models

Whenever an organization wants to protect a new application, SEPTIC needs to create the models (QMs) for the various queries. Although so far we have suggested that training is mandatory, in fact SEPTIC supports two alternative ways to learn models: *training* and *incremental*. Whereas training is used specifically for SEPTIC to create models, the incremental method allows SEPTIC to create them in an ad-hoc manner, in normal operation. The incremental method can be viewed as a complement of the former due to the reasons: First, it can extract those QMs that were missed by the training method. Second, it avoids the need to train again SEPTIC when new releases of applications replace previous ones. The incremental method creates the need for the two concepts: *quarantine* to address suspicious QMs (Section 6.2); and *aging* to deal with QMs that are not used by the new releases of applications (Section 6.3).

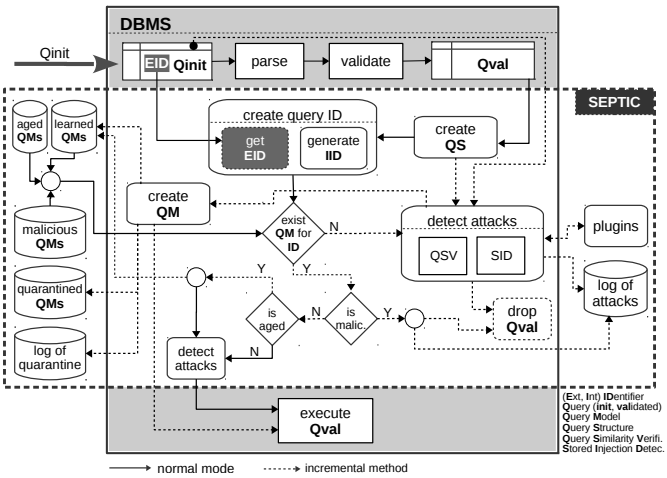


Fig. 6: Data flows for the incremental method.

6.1 Learning methods

Figure 2 and Figure 6 depicts the data flow for the *training* and *incremental* methods, respectively. Dotted-dashed arrows correspond to the training mode and dotted arrows to the incremental method.

6.1.1 Training method

This method involves putting SEPTIC in training mode and trying to execute all defined queries of the application with valid inputs (i.e., inputs that are not attacks). A training task results in the creation and storage of a model with an ID for every novel query, as described in Section 4. The execution of the defined queries of the application can be achieved in two fashions: (1) using the unit tests of the application defined by the software developers. These tests target the application in order to exercise its functionalities, which can involve calls to the databases for the execution of SQL queries with benign inputs; or (2) with the assistance of an external module that attempts to force all queries to be called. We have developed one of such modules, called *septic_training*. This module targets web applications and it works like a crawler. For each web page, it searches for HTML forms and collects information about the submission method, action, variables and values. Then, it issues HTTP requests for each form, causing the queries to be transmitted. These queries can be static, or can depend on the results of other queries, or contain inputs generated by the training module emulating a user. Both approaches are triggered by the administrator but otherwise executed automatically. Their execution time depends on the size/complexity of the application as well as the coverage of the unit tests and/or external modules. When the training phase is completed, SEPTIC can be put in normal operation and typically no further intervention from the administrator is required.

6.1.2 Incremental method

SEPTIC runs the incremental method in normal operation. This allows dealing with incomplete training (some queries not issued) and new releases of applications (Section 6.3). The basic idea is that when SEPTIC processes a query for which no QM is known, besides flagging as a possible attack it also notifies the administrator that a new query was observed.

```
1 if ID not in {learned QMs, malicious QMs, aged QMs,
      quarantined QMs} then
2     execute query similarity verification
3     execute stored injection detection
4     if any test fails then
5         return report attack
6     else
7         generate QM
8         if quarantine is off then
9            save <ID, QM> in learned QMs data store
10        else
11           save <ID, QM> in quarantined QMs data store
12           notify administrator
13        end
14    end
15 else
16    if ID in malicious QMs then
17       return report attack
18    if ID in aged QMs then
19       move <ID, QM> to learned QMs data store
20    if ID in quarantined QMs then
21       return drop Qval
22 end
23
24 when administrator classifies QMs in quarantined QMs
      data store do
25    if administrator classifies QM as valid then
26        move <ID, QM> to learned QMs data store
27    else
28        move <ID, QM> to malicious QMs data store
29    end
30 end
```

Listing 3: Incremental method algorithm.

Listing 3 presents the algorithm for the incremental method (it also includes quarantine and aging, which we leave for the following sections). As there is no QM for the query, SEPTIC first verifies if the query is an attack using the mechanisms presented in Section 5 (lines 2-3), namely the query similarity verification and stored injection detection (`INSERT` and `UPDATE`). If so, an attack is flagged (lines 4-5). If not, SEPTIC builds a model QM for the query and stores it in the learned QMs or *quarantined QMs* data store.

6.2 Quarantine

SEPTIC includes the quarantine mechanism to handle QMs that are created in normal operation by the incremental method. The idea is that SEPTIC cannot know if queries that match such QMs are benign or attacks, so they have to be analyzed by the administrator. This mechanism can be turned on or off. The latter means that new QMs are all considered benign and saved in the learned QMs data store (Listing 3, lines 8-9).

The normal configuration is quarantine set to *on*. In that situation, when a query is received for which there is no model in the learned QMs data store, a QM is generated and is saved in the quarantined QMs data store, and the administrator is notified (lines 7 and 11-12). The quarantined QMs data store serves as a temporary storage, where such QMs are saved while the administrator does not intervene. When the administrator evaluates these QMs, they are either moved to the learned QMs data store or to the malicious QMs data store, meaning that from now on queries matching those QMs will be considered, respectively, benign or attacks (lines 22 to 28).

This explanation leads to an extra rule that SEPTIC applies — queries that match a model in the malicious QMs data store are immediately flagged as an attack (lines 16-17).

6.3 New releases of applications

Attack detection has to continue to be effective when new releases of applications replace older ones. When an application is updated, queries may be added, removed, or changed in the source code, leading to different queries being made to the DBMS at runtime. This implies that SEPTIC may possibly need to change the QMs it has for that application. To address this issue, an administrator might simply retrain SEPTIC to ensure that all QMs are rebuilt by using the training method. However, this may be unfeasible or unpractical, if the application needs to be put in production immediately.

SEPTIC has a mechanism to allow updating applications without retraining. SEPTIC can be maintained in normal operation and left constructing the new QMs gradually (incremental method). One needs, however, a solution to *age* the stored models in order to ensure the (eventual) removal of QMs associated to queries that no longer exist. SEPTIC implements an *aging* mechanism for this purpose.

The mechanism registers at runtime the moments when QSs are matched with QMs (*log of used QMs* in Figure 2) and it is configured with a senescence period of time (e.g., 1 or 2 months). Models that are not utilized for the senescence period are considered to belong to previous versions of the applications and are moved from the learned QMs data store to the *aged QMs* data store. However, an old model can be brought back to life — in the incremental method, if SEPTIC observes an unknown query whose QM belongs to the aged QMs data store (Listing 3, lines 18-19), it moves the QM back to the learned QMs data store. SEPTIC understands such query as being a query of the current application release that was not issued for a long time. Also, in such case, an entry in the log of used QM is made.

The aging mechanism can also be configured to erase the models that remain a long intervals in the aged QMs data store (e.g., 6 months or 1 year) or to leave this task to the administrator. This approach may lead QMs whose queries are rarely issued to be wrongly aged and erased. Nevertheless, when such a query is eventually issued, the incremental method allows inserting the corresponding QM again in the learned QMs data store, but after passing by the quarantine procedure.

Remark 3. It is interesting to understand the impact of an update on the identifiers. This is particularly relevant if an EID (external query ID) carries context information about the application. As the next section suggests, the EID are defined by the application and may be related to the places in the code where the queries are composed and/or where the DBMS is called. Consider as an example a query that is moved from a line x to a line y in the source code, which used to have identifier IDx. We envisage two scenarios: (1) no query existed in line y, which means that a new QM

will be created with IDy (the query has a distinct EID but a similar IID to the one in IDx); (2) there was a query in line y previously; again a new QM will be built (even though the EID may stay the same, the IID is now different). In both cases, the old QMs are no longer used by the new version of the application and are aged as usual. On the other hand, if only the IID is used. Although the IID is not related with the places in the code where the queries are composed, it shows if the queries suffered changes between application versions or if new queries were developed. In both cases, new QMs will be built and old ones will be aged.

7 Vulnerability Diagnosis and Removal

This section describes how to identify vulnerabilities in the source code of applications by taking advantage of the attack detection and the information carried in the EIDs. In addition, it explains how the programs could be fixed by providing a few rules.

7.1 Diagnosis

We propose two kinds of EIDs depending on where they are generated: in the server-side language engine (SSLE) (for web-applications) or in the source code of the application (for any application).

7.1.1 SSLE-generated EIDs

In Figure 1 consider the scenario in which the server-side application code issues the query to be executed by the database. In this case, the SSLE observes a call to a function like `mysql_query`. Therefore, the SSLE can intercept the function call to add the EID to the query and then it can let the request proceed.

The EID may include information about the places in the source code where the query is composed (e.g., it may contain the filename and line number in which the query is issued). However, sometimes this might be not enough to distinguish queries because some applications have a single function that makes all calls to the database. Here, the queries are built in various parts of the code and then the single function is invoked. To address this issue, an alternative EID format could be a sequence of $file : line$ pairs. In more detail, the first pair corresponds to the line where the database is called and the rest to the lines where the query is passed as argument to some function. *file* could contain the complete pathname to distinguish queries from different applications to the same DBMS.

Example 5. Consider that the code sample of Listing 1 is in file `login.php`. The query is created in the function that calls `mysql_query`, so the EID is simply *"login.php:4"* (the filename is shown without the full pathname for readability). This means that the DBMS is called in line 4 of file `login.php`.

Example 6. Consider that line 4 is substituted by `$result = my_db_query($query)`. Also, consider that function `my_db_query` is defined in file `my_db.php` and it calls the DBMS using `mysql_query` in line 10. In this case, the EID is *"my_db.php:10 | login.php:4"*.

7.1.2 Application-generated EIDs

The developers of the application can also define their own EIDs. These EIDs can have any format, e.g., a sequential number or something similar to *file:line*. They can be added to the queries in a few ways: (1) appended to the query string when it is defined or when the database is called; or (2) a wrapper is used as an indirection to the call to the DBMS, whose responsibility is to add the identifier.

7.2 Removal with simple rules

When SEPTIC detects an attack, it logs the query (i.e., Qinit), the ID and the test that was violated (both in detection and prevention modes). The developers can use this log to diagnose the vulnerability. The EID (included in the ID) can correctly identify the query in the source code and the attack query (Qinit) shows how the vulnerability was exploited. Some rules of thumb on how to fix the application are:

- SQLI attack and user inputs are not sanitized: any of the attacks of classes S.1 or S.2 in Table 1 may have happened. Sanitization has to be inserted in the source code;
- SQLI attack and user inputs were apparently sanitized: the attack probably belonged to class A, and there was possibly a case of semantic mismatch. The sanitization has to be checked and re-implemented to deal with the problem;
- Stored injection: the attack most probably belonged to classes F–H. The programmer has to develop validation routines to apply to the inputs.

8 Implementation

This section explains the implementation of SEPTIC in MySQL. In addition, it shows how external identifiers (EID) can be added to queries in three quite diverse scenarios: for web applications developed in PHP, by modifying the runtime support in the Zend engine; for web applications implemented in the Spring framework in Java, using aspect oriented programming; and for business applications built in Visual Basic and the Gambas platform, by employing a wrapper. The first approach does not involve any changes to the application, while the remaining two require small modifications. Table 3 summarizes the changes made to those software packages.

In all cases, the external identifiers are placed inside a SQL comment to reduce the impact on the various components and maximize transparency. Specifically, SEPTIC assumes that if a query starts with a comment then the content of this comment is the identifier.

8.1 Protecting MySQL

We implemented SEPTIC in MySQL 5.7.4. There was an effort to minimize changes (i.e., number of MySQL files altered and lines of code added) to facilitate the porting of our approach to newer releases of MySQL. Overall the main modifications were the following: a single MySQL file had to be altered, `sql_parser.cc`; two new header files were included (SEPTIC detector and SEPTIC setup); a few modules were added, namely to support the configuration

TABLE 3: Summary of modifications to software packages.

Software	sfm	sfc	loc	sa
MySQL 5.7.4				
- `sql_parser.cc`	1	–	20	–
- SEPTIC detector	–	1	1740	plugins
- SEPTIC aging	–	1	180	–
- SEPTIC setup	–	1	184	–
- SEPTIC configuration	–	1	23	–
- *septic_training*	–	1	380	–
Zend engine / PHP 5.5.9				
- mysql extension	1	–	6	–
- mysqli extension	2	–	21	–
- SSLE identifier	–	1	249	–
Spring 4.0.5 / Java				
- `JdbcTemplate.java`	1	–	16	–
- Spring identifier	–	1	230	–
Gambas 3.5.1				
- Gambas identifier	–	1	187	–

sfm: source file modified loc: lines of code
sfc: source file created sa: software added

(SEPTIC configuration) and the aging of query models (SEPTIC aging); plus the plugins, which are external to the DBMS and rely on open source tools (e.g., for stored XSS the plugin is essentially the [21]). The *septic_training* module also runs separately from the DBMS.

The 20 lines added to the `sql_parser.cc` file call the SEPTIC detector with two inputs corresponding to Qval and Qinit. These lines were inserted in function `mysql_parse`, just before the call to the function `mysql_execute_command` that finishes the processing of the query. These two functions are native to MySQL.

In more detail, the SEPTIC detector is executed by the `compareQueryStructure` function. This function calls the `processSelect_Lex` and `insertElementTemplate` functions to check the query command (e.g., `SELECT`, `DELETE`, `INSERT`, `UPDATE`) and to build the QS. At the same time, this function creates the IID, gets the EID (if applicable), and composes the query ID. Then, it determines if there is a QM for that ID in the learned QMs data store. If the QM exists, and SEPTIC is in normal operation, the QM is loaded and function `compareQueryToTemplate` is called to check the QS with the QM and Qinit. If the QM is not stored in the learned QMs data store, SEPTIC applies the incremental method (or the training method if SEPTIC is in training mode) as explained in Section 6. In both cases, the QM is built from the QS and then saved either in the quarantine or the learned QMs data stores.

Comparing the QS with the QM corresponds to the first two steps of detection for SQLI attacks (Section 5.1). First, there is a verification on the number of items in both stacks (structural verification), followed by the checks per item with function `processItem` (syntactical verification). This function analyzes the 27 different types of items defined in MySQL, i.e., the items that allow MySQL to distinguish the different kinds of SQL keywords to categorize each one (e.g., function, condition, field) and represent them as a list of stacks data structure. It uses two auxiliary functions – `processField` and `isPrimitiveTypeBenign` – to detect differences between fields and to find out if an item is a primitive data type (integer, real, string or decimal), allowing casts between them. Lastly, the `processItemQuery` function is called to determine if each item of Qinit is present in QS (query similarity verification). If any item is not present, a SQLI attack is flagged. In a similar way, the tests for stored injection attacks are performed by the function `processItem` for the `INSERT` and `UPDATE` SQL commands, calling the appropriate plugins if special characters are observed in the query.

The mechanisms for aging QMs runs in background periodically and when MySQL is started, by calling function `agingQM`. It accesses the log file that registers the QMs that were matched with QS, and gets from the learned QMs data store those QMs that do not appear in the log. Next, it moves them to the aged data store. Finally, it schedules the date for the next rotation.

SEPTIC is configured by setting five switches in the SEPTIC configuration file. The first decides the mode of operation, either in training phase, detection (logs attacks), or prevention (logs and blocks attacks). The incremental method is used in these two last modes. Other two enable/disable the detection of SQLI and stored injection attacks. The fourth corresponds to the quarantine, and can be on or off. The last allows to configure the time interval between checking for aged model. When MySQL starts, the switch values are loaded.

8.2 Inserting identifiers in Zend

We implemented SSLE-generated EIDs for the PHP language by modifying the Zend engine. EIDs are formed by pairs of *file:line* separated by |, and they are placed as a comment at the beginning of the query. Overall, the format of the query instruction becomes: `/* file:line | file:line | ... | file:line */ query`.

All modifications to Zend could be concentrated in two engine extensions (see Table 3), where a few lines of code were added to call the module that implements the EIDs. Extensions are used in Zend to group related functions. A new header file was also developed to create and insert the query EID identifier.

The identifiers have to be added when the DBMS is called, so we modified in Zend the 11 functions used for this purpose (e.g., `mysql_query`, `mysqli::real_query`, and `mysqli::prepare`). Specifically, the identifier is inserted in these functions just before the line that passes the query to the DBMS. This involved modifying three files: `php_mysql.c`, `mysqli_api.c` and `mysqli_nonapi.c`.

Zend keeps a function call stack for running PHP programs. This stack contains data about the functions that are executed, such as the function name, full pathname of the file and line of code where the function was called, and the array of the arguments of the function. This stack allows backtracking until a function is found that does not contain the query as argument. This provides the places where the query was composed and/or was argument of a function, letting query identifiers to be constructed in the format above.

In Zend, we implemented the `generate_EID` function to built the EID and to append the query to the identifier. Listing 4 presents the algorithm to get the EID. Using the call stack, the algorithm starts in the sensitive sink (e.g., `mysql_query`) and continues while the query is an argument of a function call, composing the backtrace. Therefore, the stack is accessed by a `TOP` stack operation, getting the call function in the top of the stack (line 5). The function

name and the array of the function arguments are retrieved (lines 6 and 7). Then, if the function is a sensitive sink, the algorithm gets the query argument to start backtracking it (lines 9 and 10). Otherwise, the algorithm checks if the query belongs to the array of the arguments (line 12). If not, the backtracking stops (line 13), otherwise the filename and the line number where the function call was made are retrieved (lines 18 and 19) to compose the pair *file:line* and concatenate it with the previous identifier (lines 20 and 21). Next, a POP stack operation is made (line 24) and a new loop iteration is performed. After the loop, the identifier is concatenated to the query and the result is passed to the function that calls the DBMS.

```
1 identifier <- NULL
2 query <- NULL
3 backtrace <- true
4 while backtrace and is not empty stack do
5     func <- get the function of the TOP of the stack
6     function_name <- get function name of the func
7     array_args_func <- get arguments of the func
8
9     if function_name is equals a sensitive sink then
10        query <- get query from array_args_func
11    else
12        if query is not in the array_args_func then
13            backtrace <- false
14        end
15    end
16
17    if backtrace then
18        file <- get filename where the func is called
19        line <- get line number where the func is called
20        pair <- file:line
21        identifier <- concatenation(identifier, pair)
22    end
23
24    POP func from the top of the stack
25 end
```

Listing 4: Algorithm to compose the EID.

8.3 Inserting identifiers in Spring/Java

We implemented the second kind of EIDs, application-generated EIDs (Section 7.1), in Spring/Java. Spring is a framework aimed at simplifying the implementation of enterprise applications in the Java programming language [1]. In Spring applications connect to the DBMS via a JDBC driver.

We used three different methods to insert the EIDs to show the flexibility of doing it. The first solution consists in inserting the EID directly in the query in the source code of the application. Before the query is issued a comment with the EID is added. This is a very simple solution that has the inconvenient of requiring modifications to the application. The second form uses a *wrapper* to catch the query request before it is sent to the JDBC, and to insert the EID in a comment prefixing the query. Using a wrapper avoids the need to modify the source code of the application, except for the substitution of the calls to the JDBC by calls to the wrapper.

The third method does not involve modifications to the application source code. We use *Spring AOP*, an implementation of Aspect-Oriented Programming, essentially to create a wrapper [40]. Spring AOP allows the programmer to create *aspects* for the application. These aspects support the interception of method calls from the application, and the insertion of code to be executed before the methods. In our prototype, we use aspects for intercepting in runtime calls to JDBC, inserting the EID in the query and proceeding with the query request to MySQL.

TABLE 4: Code (attack) and non-code (non-attack) cases defined by Ray and Ligatti [36], [37].

	Case	Attack/code
1	`SELECT balance FROM acct WHERE password='' OR 1=1 -- '`	Yes
2	`SELECT balance FROM acct WHERE pin= exit()`	Yes
3	`...WHERE flag=1000>GLOBAL`	Yes
4	`SELECT * FROM properties WHERE filename='f.e'`	No
5	`...pin=exit()`	Yes
6	`...pin=aaaa()`	Yes
7	`SELECT * FROM t WHERE flag=TRUE`	No
8	`...pin=aaaa`	Yes
9	`SELECT * FROM t WHERE password=password`	Yes
10	`CREATE TABLE t (name CHAR(40))`	No
11	`SELECT * FROM t WHERE name='x'`	No
12	`SELECT * FROM files WHERE numEdits > 0 AND name='f.e'`	No
13	`INSERT INTO users VALUES('evilDoer', TRUE)-- ', FALSE)`	Yes
14	`INSERT INTO trans VALUES(1, -5E-10);` `INSERT INTO trans VALUES(2, 5E+5)`	Yes

8.4 Adding identifiers in non-web applications

We also implemented the application-generated EIDs in business applications developed in Gambas. Gambas is a platform offering a programming environment similar to .NET / Visual Basic for Linux [16]. We used the two first methods described for Spring/Java, i.e., inserting the EID directly in the query in the source code of the application and resorting to a *wrapper*.

9 Experimental Evaluation

The objective of the experimental evaluation was to answer the following questions: (1) Is SEPTIC able to detect and block attacks against code samples and (real) applications? (2) Is it more effective than other tools in the literature? (3) Does it solve the semantic mismatch problem better than other tools? (4) How does it perform in terms of false positives and false negatives? (5) Is SEPTIC able to learn query models (resorting to the learning methods and quarantine and/or aging functionalities)? (6) Is SEPTIC able to identify vulnerabilities in application code? (7) Is the performance overhead acceptable?

9.1 Attack detection

9.1.1 Detection with code samples

We evaluated SEPTIC with sets of 66 code samples of web applications, namely with: (1) a set of simple queries that are vulnerable to attacks from all classes in Table 1 (17 for the semantic mismatch problem, 7 for other SQLI attacks, 5 for stored injection); (2) 23 code samples from the *sqlmap* project [41], unrelated with semantic mismatch and comprising both simple and complex queries (i.e., queries composed of different SQL clauses, beside the usual `SELECT`, `FROM`, and `WHERE` clauses, and including sub-queries); (3) 14 samples with the code and non-code injection cases presented in [36], [37] (Table 4).

We compare SEPTIC with a Web Application Firewall (WAF) and four anti-SQLI tools. Figure 7 shows the place where the WAF and the anti-SQLI tools intercept, respectively, the user inputs sent in HTTP requests and the query produced by the web application. SEPTIC acts inside the DBMS. The WAF was ModSecurity 2.9.1 [43], which was

Fig. 7: Placement of the protections considered in the experimental evaluation: SEPTIC, anti-SQLI tools, and WAF.

configured with two OWASP core rule sets (CRSs), CRS 2.2.9 and CRS 3.0. ModSecurity is the most adopted WAF worldwide, with a stable rule set developed by experienced security administrators. In fact, it has been argued that its ability to detect attacks is hard to exceed [30]. It discovers SQLI and other types of attacks by inspecting HTTP requests. The anti-SQLI tools were: CANDID [4], AMNESIA [18], DIGLOSSIA [39] and SQLrand [7]. More information about them can be found in the related work (Section 12). Table 5 shows an estimation of the human effort needed to deploy and run SEPTIC in comparison to these tools. All require some effort, but SEPTIC seems to be the one that requires less.

TABLE 5: Features and human effort to deploy and use SEPTIC, the anti-SQLI tools, and ModSecurity.

		SEPTIC	SQLrand	AMNESIA	CANDID	DIGLOSSIA	ModSecurity
Features	Server-side language dependence	(X)	X	X	X	X	
	Vulnerability diagnosis	(X)					
	Quarantine	X					
	Aging	X					
	Detects SQLI attacks	X	X	X	X	X	X
	Detects stored injection attacks	X					X
	Monitors web applications	X	X	X	X	X	X
	Monitors non-web applications	X					
Human effort	Client configuration		(X)	(X)	(X)	(X)	
	Application source code modification		X	X		X	
	Application source code analysis			X	X		
	Training phase	(X)*	X	X	X	X	
	Re-training phase for new app versions		X	X	X	X	
	Analyze logs	X	X	X	X	X	X
	Modify the DBMS	X					

(X) optional
(X)* the training method is optional, but not the incremental method

In the experiments described next, we want to study the detection capacities of SEPTIC when it learns the models through both training and incremental methods, and resorting to the quarantine functionality. To achieve this, we split the experiments in four phases to, respectively, confirm the existence of vulnerabilities, test the capacity of learning using both training methods individually and mixed with quarantine, and analyze of results in terms of detection.

Phase 1: Confirming the vulnerabilities. With SEPTIC turned *off*, we injected malicious user inputs created manually to confirm the presence of the vulnerabilities in the first set of code samples. Also, we injected the inputs (code and non-code) defined in the third set of samples (Table 4) to exploit the vulnerabilities from this group. We also employed the *sqlmap* tool to exploit automatically the vulnerabilities from the first two groups of code samples. *sqlmap* is widely used to perform SQLI attacks, both by security professionals and hackers, by injecting pre-defined malicious inputs coming with the tool and malcrafted inputs that it generates automatically.

Phase 2: Learning the models using the training mode. With SEPTIC setup in *training mode*, we provided manually benign inputs to the code samples for the mechanism to build the models of *all* queries. We performed experiments both when SEPTIC employed only its own identifier (i.e., ID = IID) and when the Zend identifier was added to the IID as an EID (i.e., ID = IID + EID). This means that the training phase was carried out two times, for SEPTIC to learn the QM using the two identifiers. In this way, the tests explained next were also done two times to determine the efficiency of each ID. Then, with SEPTIC in *detection mode* we run (i) the queries with benign inputs (different from those used in the training mode), to verify if SEPTIC learned the QMs correctly, and (ii) we run the attacks from the first phase to determine if they could be discovered. We observed that any query of (i) was not flagged as attack independently of the type of identifier used, meaning that SEPTIC learned and handled the QMs correctly. In addition, regarding queries of (ii) that detection outcomes were equivalent irrespective of the type of identifier. Moreover, all attacks run with SEPTIC knowing the QMs that were only identified by IID were detected by the *query similarity verification*, whereas the attacks deployed when SEPTIC only used QMs identified by both IID and EID were detected by *structural verification* or *syntactical verification*. Therefore, in the *analysis of the results* phase (see below), we only discuss the tests carried out when the external identifier was provided (i.e., ID = IID + EID).

Phase 3: Learning the models using both training methods. As a third experiment, we setup SEPTIC using a mix of the training and incremental methods and only IID as query identifier (i.e., ID = IID). The training method was applied only to a subset of the code samples, leaving a group of queries unlearned. Afterwards, in normal operation (incremental method) with quarantine enabled, (i) we injected benign inputs in some of those unlearned queries, and (ii) we run the attacks of the first phase in the remaining unlearned queries. We observed that SEPTIC was able to put in quarantine the QMs of the queries belonging to (i), and reported as attacks the queries of (ii). This was possible because the *query similarity verification* check was enough to distinguish queries provided from (i) and (ii), and so enough to discover the attacks. Next, the QMs stored in the *quarantined QMs* data store were analyzed for correctness and were moved to the *learned QMs* data store, and the QMs resulting from the attacks (i.e., queries of (ii)) were put in the *malicious QMs* data store. Then, the attacks from the first phase were performed by exploiting the queries of (i) and (ii), confirming that they could be identified, respectively, by the query similarity verification check, since SEPTIC saw those queries for first time, and by their QMs belonging to the malicious QMs data store. Lastly, we run the queries of (i) and (ii) using benign inputs. We observed that the queries from (i) matched the QMs stored in the learned QMs data stored, where these QMs were previously learned through the incremental method and using the quarantine mechanism, and the queries from (ii) were put in quarantine, since they were new to SEPTIC.

Phase 4: Analysis of the results. The results of the second phase of experiments – *learning the models using training mode* – with the external identifier (EID) as part of the query ID (i.e., ID = IID + EID) are summarized in Table 6. There were 66 tests executed (third column), 61 of them corresponding to vulnerable code samples and the remaining 5 to valid codes (the 5 non-attack cases in Table 4).

SEPTIC found the 61 attacks (row 34) and did not flag the 5 non-attack cases (row 11). With regard to case 10 of Table 4, we highlight that although [36] considers it as being vulnerable, we are in disagreement because the input is an integer, which is the type expected by the *char* function. So, in our analysis, it is accounted as one of the 5 non-attacks. The last 3 cases of Table 4 were defined as being advanced cases of SQLI [37]. Case 12 is similar to case 4 and both were correctly found as non-attacks by SEPTIC. Case 13 mimics an `INSERT` query in its entirety; SEPTIC detected it via the *query similarity verification*. The last case is the most interesting as it transforms arithmetic operations (minus and plus) into scientific numbers. The attack was identified via *structural verification*, as SEPTIC considers the arithmetic operations as being nodes of the QM. Therefore, the QS of a query with a scientific number has less nodes than the QM. SEPTIC had neither false negatives nor positives (rows 35–36) and correctly handled the semantic mismatch problem by discovering the attacks that exploited vulnerabilities of classes A, D, and E (rows 17–21), B (row 7), C (rows 8–9), and F–H (rows 26–30).

Columns 5 to 10 contain the results for the anti-SQLI tools and ModSecurity with the two CRSs. These approaches were unable to locate a significant part of the attacks (around 50%). For example, most of them could not identify stored procedure (row 7) and stored injection (rows 26–30) attacks. The anti-SQLI tools only discovered one of the attacks from the semantic mismatch class (rows 17–21). ModSecurity did a bit better because it detected this attack plus 1st order SQLI attacks with encoding and space evasion (A.1 and A.4, rows 17 and 19). However, ModSecurity could not locate 2nd order SQLI because in the second step of these attacks the malicious input comes from the DBMS, and not from the outside. The majority of the approaches also had a few false positives (except DIGLOSSIA and ModSecurity CRS 3.0). Overall, most of the problems that were observed are justified by difficulties in dealing with the semantic mismatch and the Ray and Ligatti code samples (row 10), namely when the injected queries included non-code characters that are not recognized by the tools, but are at the base of the attacks.

The answer to the first five questions is positive. We conclude that the proposed approach to detect and block (SQL and stored) injection attacks is effective because it uses the same information as the DBMS execution engine, without the need of assumptions about how the queries are run, which is the root of the semantic mismatch problem. Moreover, the quarantine mechanism is beneficial to reduce false positives and false negatives, and a way of complementing SEPTIC's training mechanism. Although not shown in the table, the experiments with SEPTIC with the two types of identifiers gave similar results. This indicates that in terms of detection capability, the use of internal identifiers (IID) is as effective as the combination of internal and external identifiers (IID + EID). However, the second kind of identifier brings the extra benefit of assisting on the discovery of the exploited vulnerabilities. We used the identifiers produced by Zend to look for the bugs in the code samples, and they had a high level of accuracy to locate the source of the problem. Therefore, this allows us to answer positively to the question 6.

9.1.2 Detection with real software

SEPTIC was used to protect the database of 10 different open source PHP web applications (e.g., hospital and school management, message forums, and bibliographic references) and a non-web application. The *wapiti* scanner [46] carried out the attacks in the experiments with the web applications. *wapiti* searches web applications looking for scripts and forms where it can place data. Then, it acts as a fuzzer to do the attacks, injecting malicious data. When SEPTIC stopped an attack, we resorted to the EIDs to help locate the vulnerabilities in web applications code. Table 7 summarizes the detection results with web and non-web applications. SEPTIC identified 91 attacks associated with the exploitation of 31 distinct vulnerabilities – 22 SQLI and 9 stored injection. The stored injections were RFI, OSCI, RCI or stored XSS.

In the experiments described next, we want to study the SEPTIC behavior when it resorts to the *aging* functionality, processes complex queries such as dynamic queries (e.g., queries that are built dynamically by users), and deals with non-web applications. To do so, the experiments are split in four phases:

Phase 1: Aging functionality and vulnerability identification. *wapiti* could successfully exploit three SQLI vulnerabilities in *ZeroCMS*, which SEPTIC was able to stop before corrupting the database. The EIDs supported the discovery of the vulnerabilities in the source code. These vulnerabilities are actually not new as they appear in the public databases CVE [12] and OSVDB [32] with identifiers CVE-2014-4194, CVE-2014-4034 and OSVDB ID 108025.

We generated a new version of ZeroCMS by fixing the vulnerabilities. The changes did not alter the queries, but caused them to move from the original place in the files. Notice that even though queries were not modified, and therefore their IIDs remained the same, they had a new EID because of the novel location in the code. Therefore, they had an ID for which no QM existed. This new version of the application was utilized to study the *aging* functionality of SEPTIC. To do so, SEPTIC was kept in *normal operation* (incremental method) and we configured the aging time for three days (instead of the usual months). Then, ZeroCMS was tested with benign and malicious inputs (attacks) to exercise the queries that moved in the code.

The queries carrying benign inputs were learned (i.e., the QM), while the queries resulting from the malicious inputs were flagged as attacks by the *query similarity verification* check. Afterwards, we repeated the attacks and confirmed that they were immediately discarded. Three days later, we also observed that the *aged QMs* data store had the (new) QMs that were not tested again and the (old) QMs from the previous ZeroCMS version. We manipulated the

TABLE 6: Detection of attacks with code samples.

	Type of attack	N. Tests	SEPTIC	anti-SQLI tools				ModSecurity WAF	
				SQLrand	AMNESIA	CANDID	DIGLOSSIA	CRS 2.2.9	CRS 3.0
SQLI without sanitization and semantic mismatch (S.1, S.2, B, C, D, E)									
3	Syntax structure 1st order	1	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4	Syntax structure 2nd order	1	Yes	Yes	Yes	No	No	No	No
5	Syntax mimicry 1st order	1	Yes	No	No	No	Yes	Yes	Yes
6	Syntax mimicry 2nd order	1	Yes	No	No	No	No	No	No
7	Stored procedure	1	Yes	No	No	No	No	No	No
8	Blind SQLI syntax structure	1	Yes	Yes	Yes	Yes	Yes	Yes	Yes
9	Blind SQLI syntax mimicry	1	Yes	No	No	No	Yes	Yes	Yes
10	Ray & Ligatti code	9	9	3	4	4	8	3	2
11	Ray & Ligatti non-code	5 (non-attacks)	0	2	1	2	0	1	0
12	sqlmap project	23	23	23	23	23	23	23	23
13	Flagged as attack	–	39	31	31	31	35	31	29
14	False positives	–	0	2	1	2	0	1	0
15	False negatives	–	0	10	9	10	4	9	10
SQLI with sanitization and semantic mismatch (S.1, S.2, A.1–A.5, D, E)									
17	Syntax structure 1st order	4	4	0	0	0	0	2	2
18	Syntax structure 2nd order	4	4	0	0	0	0	0	0
19	Syntax mimicry 1st order	4	4	0	0	0	0	2	2
20	Syntax mimicry 2nd order	4	4	0	0	0	0	0	0
21	Numeric fields	1	1	1	1	1	1	1	1
22	Flagged as attack	–	17	1	1	1	1	5	5
23	False positives	–	0	0	0	0	0	0	0
24	False negatives	–	0	16	16	16	16	12	12
Stored injection (F–H)									
26	Stored XSS	1	Yes	No	No	No	No	No	Yes
27	RFI	1	Yes	No	No	No	No	No	Yes
28	LFI	1	Yes	No	No	No	No	No	Yes
29	RCI	1	Yes	No	No	No	No	No	Yes
30	OSCI	1	Yes	No	No	No	No	No	Yes
31	Flagged as attack	–	5	0	0	0	0	0	5
32	False positives	–	0	0	0	0	0	0	0
33	False negatives	–	0	5	5	5	5	5	0
34	**Flagged as attack**	–	61	32	32	32	36	36	39
35	**False positives**	–	0	2	1	2	0	1	0
36	**False negatives**	–	0	31	30	31	25	26	22

TABLE 7: Detection of attacks in the exploitation of distinct vulnerabilities in real applications.

Application	version	SQLI	Stored inj.	attacks
Care2x	2.4	2	4	6
Ceres CP	1.1.7	1	3	4
Churchinfo	0.1	–	–	–
Gambas application	–	6	–	10
measureit	1.1.4	–	1	1
mybb	1.6.08	3	–	10
PHP Address Book	8.1.19	2	–	20
refbase	0.9.6	–	–	–
Schoolmate	–	–	1	1
WebChess	1.0.0	5	–	13
ZeroCMS	1.0	3	–	26
Total		22	9	91

application in order to force queries for those (new) QMs, and we saw that they were moved back to the *learned QMs* data store. Based on these experiments, we confirmed that the aging functionality is beneficial for handling new application releases. In addition, these results allow us to answer positively to question 5.

Most of the other applications also had security problems. For example, in *measureit* and *WebChess* was found respectively one attack that would exploit a stored injection vulnerability and thirteen different attacks for the five SQLI. SEPTIC managed to block all these attacks. In addition, we inspected the source code with the assistance of the EID identifiers registered in the log file, and they provided accurate indications about the location of the bugs. No problems were found in the *Churchinfo* and *refbase* applications. So, overall these results allow us to answer affirmatively to questions 1 and 5.

Phase 2: Analysis of results for false positives and negatives. To extend the analysis on false positives/negatives, we looked at three kinds of information kept by SEPTIC: (1) a log with all analyzed queries was checked to determine if there were malicious queries that had remained unblocked (false negatives); (2) the log of attacks was verified to find out if SEPTIC had erroneously flagged a benign query as malicious (false positives); (3) the notifications of the queries that were put in *quarantine* were inspected. We did not find any anomaly in points (1) and (2), meaning that SEPTIC did not report false positives and did not miss detections (false negatives). For point (3), we observed that SEPTIC correctly quarantined those queries for which there was no QM and that passed all checks (namely the query similarity verification). Most of these queries were actually benign, but a few of them were malicious. This is the desired behavior, as it prevents SEPTIC from making mistakes, allowing the administrator to take the final decision with regard to the validity of the queries. Therefore, these results give a positive answer to question 4.

Phase 3: Process complex and dynamic queries. Here we want to check how SEPTIC deals with complex and dynamic queries in terms of learning QMs and detecting attacks. *refbase* is a web application for managing bibliographic references. Besides allowing to insert, delete and update references, it also lets users search for references based on several criteria. This means that it is possible to create from simple reference searches (such as obtaining all references from a given author) to more elaborated ones, as for instance getting the references that contain 5 terms from a certain area, authors,

and publisher. The application implements the queries associated with these searches dynamically, which sometimes can result in complex queries. These search queries are built at a single point of the source code, meaning that their EID is always the same. On the other hand, their IID can be different because dynamic queries can have diverse parameters, thus resulting in statements that are syntactically distinct. Therefore, when issued they may have an ID for which no QM exists.

We setup SEPTIC in training mode and performed bibliographic reference searches with different parameters in order to obtain simple and complex queries. Then, with SEPTIC in normal operation, we repeated the same searches and new ones for which there were no QMs. We observed that these latter queries caused their QMs to be put in quarantine, while the former queries matched the QMs in the *learned QMs* data store. In addition, we executed some attacks based on these queries and other new ones. SEPTIC correctly detected all of them: the attacks that tried to exploit queries corresponding to QMs that SEPTIC knew were discovered by the first two SQLI verifications; the novel attacks were found by the third SQLI verification. Therefore, we can conclude that SEPTIC processes correctly complex and dynamic queries, both by building their QMs and detecting attacks.

Phase 4: Detection in non-web applications. We developed a vulnerable Gambas application to manage contacts, i.e., an address book [16]. The application contains eight queries from which six are vulnerable to SQLI. We trained SEPTIC using the incremental method (Section 6), i.e., by forcing the application to issue non-malicious queries to the database. Then, we injected different kinds of attacks, which were correctly identified by SEPTIC. Row 5 of Table 7 shows these results, where ten attacks were issued against the SQLI bugs. Therefore, these results give a positive answer to question 1.

9.2 Performance overhead

To answer question 7, we evaluated the overhead of SEPTIC using BenchLab v2.2 [10] with the *PHP Address Book*, *refbase* and *ZeroCMS* applications. BenchLab is a testbed for web application benchmarking. It generates realistic workloads, then it replays their traces using web browsers while measuring the application performance.

We have set up a network composed of six identical machines: Intel Pentium 4 CPU 2.8 GHz (1-core and 1-thread) with 2 GB of RAM, running Linux Ubuntu 14.04. Two machines played the role of servers: one run the MySQL DBMS with SEPTIC; the other executed an Apache web server with Zend and the web applications, and Apache Tomcat to run the BenchLab server. The other four machines were used as client machines, running BenchLab clients and Firefox web browsers to replay workloads previously stored by the BenchLab server, i.e., to issue a sequence of requests to the web application being benchmarked. The BenchLab server has the role of managing the experiments.

We evaluated SEPTIC with its four combinations of protections turned on and off (SQLI and stored injection on/off) and compared them with the original MySQL without SEPTIC installed (base)[5]. For that purpose, we created several scenarios, varying the number of client machines and browsers. The *ZeroCMS* trace was composed of 26 requests to the web application with queries of several types (`SELECT`, `UPDATE`, `INSERT` and `DELETE`). The traces for the other applications were similar but for *PHP Address Book* the trace had 12 requests, while for *refbase* it had 14 requests. All traces involved downloading images, cascading style sheets documents, and other web objects. Each browser executes the traces in a loop many times.

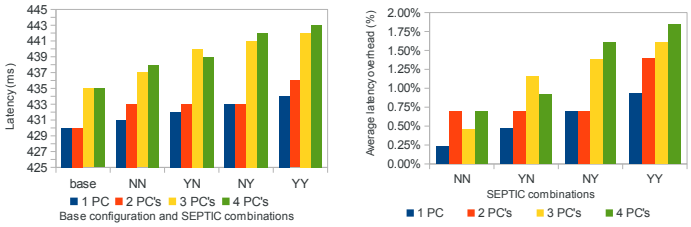
Table 8 summarizes the performance measurements. The main metric assessed was the *latency*, i.e., the time elapsed between the browser starts sending a request and finishes receiving the corresponding reply. For each configuration the table shows the *average latency* and the *average latency overhead* (i.e., the average latency divided by the latency obtained with MySQL without SEPTIC, multiplied by 100). These values are presented as a pair *(latency (ms), overhead (%))* and are shown in the 4th to 8th columns of the table. The 1st column characterizes the scenario, varying the number of client machines (*PCs*) and browsers (*brws*). The next two columns show the number of times that each configuration was tested with a trace (*num exps*) and the total number of requests done in these executions (*total reqs*). Each configuration was tested with 5500 trace executions, in a total of 87,200 requests (last row of the table). The latency obtained with MySQL without SEPTIC is shown in the 4th column and the SEPTIC combinations in the next four.

TABLE 8: Performance overhead of SEPTIC measured with Benchlab for three web applications: *PHP Address Book*, *refbase* and *ZeroCMS*. Latencies in ms, overheads in %.

N. PCs & brws	Num exps	Total reqs	Base	SEPTIC: SQL injection – stored injection			
				off–off	on–off	off–on	on–on
***refbase* varying the number of PCs, one browser per PC**							
1 PC	70	980	430, –	431, 0.23	432, 0.47	433, 0.70	434, 0.93
2 PCs	120	1680	430, –	433, 0.70	433, 0.70	433, 0.70	436, 1.40
3 PCs	170	2380	435, –	437, 0.46	440, 1.15	441, 1.38	442, 1.61
4 PCs	220	3080	435, –	438, 0.69	439, 0.92	442, 1.61	443, 1.84
***refbase* with four PCs and varying the number of browsers**							
8 brws	420	5880	504, –	506, 0.40	510, 1.19	513, 1.79	516, 2.38
12 brws	620	8680	530, –	532, 0.38	535, 0.94	539, 1.70	544, 2.64
16 brws	820	11480	540, –	541, 0.19	545, 0.93	550, 1.85	553, 2.41
20 brws	1020	14280	570, –	573, 0.53	575, 0.88	581, 1.93	584, 2.46
***PHP Address Book* with four PCs**							
20 brws	1020	12240	79, –	79.26, 0.33	79.50, 0.63	80.60, 2.03	81, 2.53
***ZeroCMS* with four PCs**							
20 brws	1020	26520	239, –	240, 0.42	241, 0.84	243, 1.67	245, 2.51
AO/Total	5500	87200	–, –	*0.41%*	*0.82%*	*1.65%*	*2.24%*

The first set of experiments evaluated the overhead of SEPTIC with the *refbase* application (rows 3–6). We run a single Firefox browser in each client machine but varied the number of these machines from 1 to 4. For each additional machine we increase the number of experiments (*num exps*) by 50. Figure 8 represents graphically these results, showing the (a) latency measurements and the (b) latency overhead of the different SEPTIC configurations. SQLI and stored injection on/off is represented by Y/N. The most interesting conclusion taken from the figure is that the overhead of running SEPTIC is very low, always below 2%. Another interesting conclusion is that SQLI detection has

5. Notice that the *off–off* combination is not the same as the *base* because some code of SEPTIC is executed to check if protections are turned on or off.



(a) Latency (b) Overhead

Fig. 8: Latency and overhead with *refbase* varying the number of PCs, each one with a single browser.

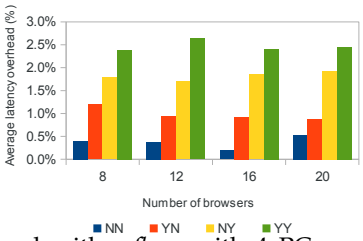


Fig. 9: Overhead with *refbase* with 4 PCs and varying the number browsers.

less overhead than stored injection detection, as the values for configuration NY are just slightly higher than those for YN. Finally, the overhead tends to grow with the number of PCs and browsers as the load increases.

The second set of experiments were again with *refbase*, this time with the number of client machines (PCs) set to 4 and varying the number of browsers (Table 8, rows 8–11). Figure 9 shows how the overhead varies when going from 1 to 4 PCs with browsers varying from 8 (2 per PC) to 20 (5 per PC). The results lead to similar conclusions as the first set of experiments. They also show that raising the number of browsers initially increases the overhead (Figure 8(b)), then stabilizes (Figure 9), as neither the CPU at the PCs nor the bandwidth of the network were the performance bottleneck.

The third and fourth sets of experiments used the *PHP Address Book* and *ZeroCMS* web applications and 20 browsers in 4 PCs (Table 8, rows 13 and 15). The overhead of all applications is similar for each SEPTIC configuration. This is interesting because the applications and their traces have quite different characteristics, which suggests that the overhead imposed by SEPTIC is independent of the server-side language and web application.

The average of the overheads varied between 0.82% and 2.24% (AO in last row of the table). This seems to be a reasonable overhead when compared to the overheads (as reported in original the papers) of the anti-SQLI tools used in Section 9.1.1: 3.35% for SQLrand; between 3.2% and 42.8% for CANDID; and a maximum of 13% for DIGLOSSIA. This suggests that SEPTIC is usable in real settings, answering positively question 7.

10 Protecting other DBMSs

The SEPTIC approach is not specific to MySQL. To show that this is the case, we discuss how to implement the approach in two other DBMSs, based on an analysis we have made of their source code. We analyzed MariaDB 10.0.20 [27] and PostgreSQL 9.4.4 [35]. MariaDB is a fork of MySQL created around 2009 due to concerns over Oracle's acquisition of MySQL. PostgreSQL is the second most popular open source DBMS, after MySQL [38].

10.1 MariaDB

MariaDB has essentially the same architecture as MySQL. When a query is received, it parses, validates, and executes it (see Figure 2). The outcome of the parsing and validation phases is the same as in MySQL, a *list of stacks* where each stack of the list represents a clause of the query, and each of its nodes contains data about the query element. Moreover, the file that contains the calls to the functions that perform parsing, validation and execution of a query is the same as in MySQL: `sql_parser.cc`. Therefore, SEPTIC can be implemented in MariaDB similarly to how it was in MySQL (Section 8.1).

10.2 PostgreSQL

The implementation of SEPTIC in PostgreSQL has some differences but also many similarities to the MySQL and MariaDB cases. The processing of a query in PostgreSQL involves four phases: parsing/validation, rewriting, planning/optimization, and execution. Again the SEPTIC module is inserted after the parsing phase, before the rewriting phase. Similarly to MySQL, a single file has to be modified (`postgresql.c`), adding essentially the same 20 lines of code that were added to MySQL. That file contains the function `exec_simple_query` that runs the four processing phases of a query. The code would be inserted after the call to function `pg_parse_query` that parses and validates the query, just before the call to the function that executes the rewriting phase (`pg_analyze_and_rewrite`). SEPTIC might also be inserted after the rewriting phase, but the adaptation would be harder as rewriting produces a different data structure, a query tree.

The data structure resulting from the parsing phase is slightly different from MySQL's but still a *list of stacks*. Again each stack of the list represents a clause of the query (e.g., `SELECT`, `FROM`) and its nodes a query element. PostgreSQL tags the query elements with their types and distinguishes the primitive types (e.g., integer, float/real, string). The nodes of the stacks contain this information similarly to what happens in MySQL, but the tags, the structure of the nodes, and the way they are organized in the stack are different from MySQL. Therefore, the data structures used in PostgreSQL and MySQL are similar, but the current implementation of the module *SEPTIC detector* has to be modified, specifically: (1) the navigation in the *list of stacks*; (2) the identification of the data about the query elements in the nodes; and (3) the collection of this data. These modifications are related with the construction of query structure for every query.

11 Discussion and Future Work

The detection of injection attacks to databases has deserved a significant attention by the research community, with several approaches and tools being proposed in the past.

SEPTIC explores a new point in the design space by identifying the attacks inside the DBMS, which has the benefit of precluding the semantic mismatch problem. The current design, implementation and evaluation has several limitations that suggest interesting open problems for future research:

- Our design assumes that the DBMS represents a query as a list of stacks. Although this is the most common method, other DBMSs could resort to different data structures. In this case, either it is possible to perform a translation between data structures or the tests for attack detection would have to be adapted to leverage from the available information.
- SEPTIC still requires some manual effort by the administrator, for instance to initiate the training or to assess the QM in the quarantine data store. A significant effort was made to eliminate this sort of tasks from the critical path of putting an application in production, but it would have been nice if a fully automated solution could have been created.
- The aging process allows queries to proceed if they correspond to a QM of a previous version of the application. However, it is possible that these models are no longer acceptable, as they may let attacks fit these QM. One solution to avoid this limitation is to employ a more aggressive senescence period, but this introduces trade-offs that need to be better understood.
- The current evaluation focuses mostly on SQL injection. The detection of stored injection attacks, including XSS, would need extra work to be thoroughly studied (but this probably requires a new, equally longer, paper).

12 Related Work

There is a vast corpus of research in web application security, so we survey only related runtime protection mechanisms, which is the category in which SEPTIC fits.

All the works we describe have a point in common that makes them quite different from our work: their focus is on *how to do detection or protection*. On the contrary, our work is more concerned with an architectural problem: *how to do detection/protection inside the DBMS*, so that it runs out of the box when the DBMS is started. None of the related works does detection inside the DBMS.

AMNESIA [18] and CANDID [4] are two of the first works about detecting SQLI by comparing the structure of an SQL query before and after the inclusion of inputs and before the DBMS processes the queries. Both use query models to represent the queries and do detection. AMNESIA creates models by analyzing the source code of the application and extracting the query structure. Then, AMNESIA instruments the source code with calls to a wrapper that compares queries with models and blocks attacks. CANDID also analyses the source code of the application to find database queries, then simulates their execution with benign strings to create the models. On the contrary, SEPTIC does not involve source code analysis or instrumentation. With SEPTIC we aim to make the DBMS protect itself, so both model creation and attack detection are performed inside the DBMS. Moreover, SEPTIC aims to handle the semantic mismatch problem, so it analyses queries just before they are executed, whereas AMNESIA and CANDID do it much earlier. These two tools also cannot detect attacks that do not change the structure of the query (syntax mimicry).

Buehrer et al. [8] present a similar scheme that manages to detect mimicry attacks by enriching the models (parse trees) with comment tokens. However, their scheme cannot deal with most attacks related with the semantic mismatch problem. SqlCheck [42] is another scheme that compares parse trees to detect attacks. SqlCheck detects some of the attacks related with semantic mismatch, but not those involving encoding and evasion. Again, both these mechanisms involve modifying the application code, unlike SEPTIC.

DIGLOSSIA [39] is a technique to detect SQLI attacks that was implemented as an extension of the PHP interpreter. The technique first obtains the query models by mapping all query statements' characters to shadow characters except user inputs, and computes shadow values for all string user inputs. Second, for a query execution it computes the query and verifies if the root nodes from the two parsed trees are equal. Like SEPTIC, DIGLOSSIA detects syntax structure and mimicry attacks but, unlike SEPTIC, it neither detects second-order SQLI once it only computes queries with user inputs, nor encoding and evasion space characters attacks as these attacks do not alter the parse tree root nodes before the malicious user inputs are processed by the DBMS. Although better than AMNESIA and CANDID, it does not deal with all semantic mismatch problems.

Works based on anomaly intrusion detection systems also aim to detect SQLI attacks by comparing models with queries sent by web applications. Valeur et al. present one of these works [44]. The system also undergoes a training phase to create models (a set of profiles) of normal access to the database. In runtime it detects deviations from that model. SQL-IDS [22] is another system that compares queries against query specifications that define the query syntactic structure (a kind of model) implemented in the application. However, there is no information about how such specifications are created, despite the authors arguing that their source code does not need instrumentation. SQLProb [26] is a proxy-based system that also uses models previously extracted by a specific data collection phase. Afterwards, the system evaluates the queries produced by applications, parsing them, and extracting their user inputs, then validates the inputs against the parse tree, resorting to an input repository. For web services Laranjeiro et al. [25] propose a similar approach to discover SQL and XPath injection attacks. In a first phase, their approach learns regular requests by representing them into invariant statements (a kind of models), and later protects web services by matching incoming requests with those collected in the learning phase. Moreover, the approach uses heuristics to deal with incoming requests which the approach does not learn as invariant. All these systems, like the previously mentioned tools, are external to the DBMS, so they do not use our approach to deal with the semantic mismatch problem.

Machine learning approaches for detection SQLI have been emerging. idMAS-SQL is one of these works [34].

SOFIA [9] also uses machine learning to classify queries issued by applications, resorting to a clustering algorithm. The tool has a training phase to get the parse tree from legitimate queries and to create clusters with these trees. Afterwards, in evaluating phase it classifies as attack the queries that do not fit any cluster.

Dynamic taint analysis tracks the flow of user inputs in the application and verifies it they reach dangerous instructions. Xu et al. [47] show how this technique can be used to detect SQLI and reflected XSS. They annotate the arguments from source functions and sensitive sinks as untrusted and instrument the source code to track the user inputs to verify if they reach the untrusted arguments of sensitive sinks (e.g., functions that send queries to the database). ARDILLA [23] creates attack vectors which contain mutations of user inputs generated previously, and then deploys such vectors, tracking the inputs and verifying if they exploit SQLI and XSS vulnerabilities. A different but related idea is implemented by CSSE that protects PHP applications from SQLI, XSS and OSCI by modifying the platform to distinguish between what is part of the program and what is external (input), defining checks to be performed to the latter [33] (e.g., if the query structure becomes different due to inputs). WASP does something similar to block SQLI attacks [19]. SEPTIC does not track inputs in the application, but runs in the DBMS.

Recently, Masri et al. [28] and Ahuja et al. [2] presented two works about prevention of SQLI attacks. The first presents a tool called SQLPIL that simply transforms SQL queries created as strings into prepared statements, preventing SQLI in the source-code. The second, presents three new approaches to detect and prevent SQLI attacks based on rewriting queries, encoding queries and adding assertions to the code. However, these approaches are not even evaluated experimentally. Again, both works involve instrumenting and modifying the application code, unlike SEPTIC that works inside the DBMS.

13 Conclusion

The paper explores a new form of protection from attacks against web and business application databases. It presents the idea of catching attacks inside the DBMS, letting it protected from SQLI and stored injection attacks. Moreover, by putting protection inside the DBMS, we show that it is possible to detect and block sophisticated attacks, including those related with the semantic mismatch problem. As a second idea, it presents a form of identifying vulnerabilities in application code, when attacks are detected. The paper also presents SEPTIC, a mechanism implemented inside MySQL. In order to do detection, SEPTIC resorts to a learning phase, and quarantine and aging processes that deal with models of queries, creating and managing them. The mechanism was experimented both with synthetic code with vulnerabilities inserted on purpose and with open source PHP web applications, and other type of applications. This evaluation suggests that the mechanism can detect and block the attacks it is programmed to handle, performing better that all other tools in the literature and the WAF most used in practice, and can identify the vulnerabilities in code of applications, when the attacks attempted exploit them. The performance overhead evaluation of SEPTIC inside MySQL shows an impact of around 2.2%, suggesting that our approach can be used in real systems.

Acknowledgments. This work was partially supported by the EC through project FP7-607109 (SEGRID), and by national funds through Fundação para a Ciência e a Tecnologia (FCT)/MCTES (PIDDAC)/FEDER with reference to project AAC-2/SAICT/2017-029058 (SEAL), and through FCT with references UID/CEC/00408/2013 (LASIGE) and UID/CEC/50021/2013 (INESC-ID).

References

[1] Spring framework, 2014. http://spring.io/.
[2] B. Ahuja, A. Jana, A. Swarnkar, and R. Halder. On preventing SQL injection attacks. *Advanced Computing and Systems for Security*, 395:49–64, 2015.
[3] Akamai Technologies. Q1 2016 state of the internet / security report. June 2016.
[4] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: preventing SQL injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 12–24, Oct. 2007.
[5] C. A. Bell. *Expert MySQL*. Apress, 2007.
[6] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax. IETF Request for Comments: RFC 3986, Jan. 2005.
[7] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security Conference*, pages 292–302, 2004.
[8] G. T. Buehrer, B. W. Weide, and P. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, pages 106–113, Sept. 2005.
[9] M. Ceccato, C. D. Nguyen, D. Appelt, and L. C. Briand. SOFIA: An automated security oracle for black-box testing of sql-injection vulnerabilities. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 167–177, 2016.
[10] E. Cecchet, V. Udayabhanu, T. Wood, and P. Shenoy. Benchlab: An open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.
[11] J. Clarke. *SQL Injection Attacks and Defense*. Syngress, 2009.
[12] CVE. http://cve.mitre.org.
[13] A. Douglen. SQL smuggling or, the attack that wasn't there. Technical report, COMSEC Consulting, Information Security, 2007.
[14] M. Dowd, J. Mcdonald, and J. Schuh. *Art of Software Security Assessment*. Pearson Professional Education, 2006.
[15] J. Fonseca, N. Seixas, M. Vieira, and H. Madeira. Analysis of field data on web security vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 11(2):89–100, 2014.
[16] Gambas. http://gambas.sourceforge.net/.
[17] T. Gigler, B. Glas, N. Smithline, and A. van der Stock. OWASP Top 10: The ten most critical web application security risks – RC2. Technical report, OWASP Foundation, 2017.
[18] W. Halfond and A. Orso. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183, Nov. 2005.
[19] W. Halfond, A. Orso, and P. Manolios. WASP: protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34(1):65–81, 2008.
[20] M. Howard and D. LeBlanc. *Writing Secure Code for Windows Vista*. Microsoft Press, 1st edition, 2007.
[21] JSoup. http://jsoup.org.
[22] K. Kemalis and T. Tzouramanis. SQL-IDS: A specification-based approach for sql-injection detection. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 2153–2158, Mar. 2008.
[23] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering*, pages 199–209, May 2009.
[24] M. Koschany. Debian hardening, 2013. https://wiki.debian.org/Hardening.

[25] N. Laranjeiro, M. Vieira, and H. Madeira. A learning-based approach to secure web services from SQL/XPath injection attacks. In *Proceedings of the 16th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 191–198, Dec. 2010.
[26] 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*, pages 2054–2061, Mar. 2009.
[27] MariaDB. http://mariadb.org.
[28] W. Masri and S. Sleiman. SQLPIL: SQL injection prevention by input labeling. *Security and Communication Networks*, 8(15):2545–2560, 2015.
[29] I. Medeiros, N. F. Neves, and M. Correia. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the International World Wide Web Conference*, pages 63–74, Apr. 2014.
[30] G. Modelo-Howard, C. Gutierrezand, F. Arshad, S. Bagchi, and Y. Qi. Psigene: Webcrawling to generalize SQL injection signatures. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014.
[31] Naked Security by SOPHOS. The web attacks that refuse to die, June 2016. https://nakedsecurity.sophos.com/2016/06/15/the-web-attacks-that-refuse-to-die/.
[32] OSVDB. http://osvdb.org.
[33] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, pages 124–145, 2005.
[34] C. I. Pinzon, J. F. D. Paz, A. Herrero, E. Corchado, J. Bajo, and J. M. Corchado. idMAS-SQL: Intrusion detection based on MAS to detect and block SQL injection through data mining. *Information Sciences*, 231, 2013.
[35] PostgreSQL. http://www.postgresql.org/.
[36] D. Ray and J. Ligatti. Defining code-injection attacks. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 179–190, 2012.
[37] D. Ray and J. Ligatti. Defining injection attacks. In *Proceedings of the International Conference on Information Security*, pages 425–441, 2014.
[38] SolidIT. DB-Engines Ranking. http://db-engines.com/en/ranking, accessed Aug. 10th, 2015.
[39] S. Son, K. S. McKinley, and V. Shmatikov. Diglossia: Detecting code injection attacks with precision and efficiency. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1181–1192, 2013.
[40] Spring. http://docs.spring.io/spring/docs/2.5.4/reference/aop.html.
[41] sqlmap. https://github.com/sqlmapproject/testenv/tree/master/mysql.
[42] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, Jan. 2006.
[43] Trustwave SpiderLabs. ModSecurity - Open Source Web Application Firewall. http://www.modsecurity.org.
[44] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of SQL attacks. In *Proceedings of the 2nd Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 123–140, July 2005.
[45] W3Techs. Usage of server-side programming languages for websites. https://w3techs.com/technologies/overview/programming_language/all.
[46] wapiti. http://wapiti.sourceforge.net/.
[47] W. Xu, S. Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, 2005.

Ibéria Medeiros is an Assistant Professor in the Department of Informatics, at the Faculty of Sciences of University of Lisbon. She is a member of the Large-Scale Informatics Systems (LASIGE) Laboratory, and the Navigators research group. She holds a PhD in Computer Science by the Faculty of Sciences of University of Lisbon. Currently, she is the principal investigator of the SEAL national project, has been participating in DiSIEM European project, and participate in SEGRID European project. She is author of tools for software security, which WAP (Web Application Protection) is the most known and an OWASP project. Her research interests are concerned with software security, source code static analysis, vulnerability detection, data mining and machine learning, and security. More information about her at http://www.di.fc.ul.pt/∼imedeiros/.

Miguel Beatriz is a Developer at Sky Technology Centre - Portugal. He holds a MsC in Computer Science by Instituto Superior Técnico (IST) of Universidade de Lisboa (ULisboa). He is a young and passionate software engineer with interest in management systems.

Nuno Neves is Professor at the Department of Computer Science, Faculty of Sciences of the University of Lisboa. He leads the Navigators research group and he is on the scientific board of the LASIGE research unit. His main research interests are in security and dependability aspects of distributed systems. Currently, he is investigator in several national and EU projects, such as SEAL and uPVN. His work has been recognized in several occasions, for example with the IBM Scientific Prize and the William C. Carter award. He is on the editorial board of the International Journal of Critical Computer-Based Systems. More information about him can be found at http://www.di.fc.ul.pt/∼nuno/.

Miguel Correia is an Associate Professor with Habilitation at Instituto Superior Técnico (IST) of Universidade de Lisboa (ULisboa), and a Senior Researcher at INESC-ID in the Distributed Systems Group (GSD). He has been involved in several international and national research projects related to cybersecurity, including the SPARTA, QualiChain, SafeCloud, PCAS, TCLOUDS, ReSIST, CRUTIAL, and MAFTIA European projects. He has more than 150 publications and is Senior Member of the IEEE. More information about him at http://www.gsd.inesc-id.pt/∼mpc/