A System for Profiling and Monitoring Database Access Patterns by Application Programs for Anomaly Detection

Lorenzo Bossi, Elisa Bertino, *Fellow, IEEE*, and Syed Rafiul Hussain, *Member, IEEE*

Abstract—Database Management Systems (DBMSs) provide access control mechanisms that allow database administrators (DBAs) to grant application programs access privileges to databases. Though such mechanisms are powerful, in practice finer-grained access control mechanism tailored to the semantics of the data stored in the DMBS is required as a first class defense mechanism against smart attackers. Hence, custom written applications which access databases implement an additional layer of access control. Therefore, securing a database alone is not enough for such applications, as attackers aiming at stealing data can take advantage of vulnerabilities in the privileged applications and make these applications to issue malicious database queries. An access control mechanism can only prevent application programs from accessing the data to which the programs are not authorized, but it is unable to prevent misuse of the data to which application programs are authorized for access. Hence, we need a mechanism able to detect malicious behavior resulting from previously authorized applications. In this paper, we present the architecture of an anomaly detection mechanism, *DetAnom*, that aims to solve such problem. Our approach is based the analysis and profiling of the application in order to create a succinct representation of its interaction with the database. Such a profile keeps a signature for every submitted query and also the corresponding constraints that the application program must satisfy to submit the query. Later, in the detection phase, whenever the application issues a query, a module captures the query before it reaches the database and verifies the corresponding signature and constraints against the current context of the application. If there is a mismatch, the query is marked as anomalous. The main advantage of our anomaly detection mechanism is that, in order to build the application profiles, we need neither any previous knowledge of application vulnerabilities nor any example of possible attacks. As a result, our mechanism is able to protect the data from attacks tailored to database applications such as code modification attacks, SQL injections, and also from other data-centric attacks as well. We have implemented our mechanism with a software testing technique called concolic testing and the PostgreSQL DBMS. Experimental results show that our profiling technique is close to accurate, requires acceptable amount of time, and the detection mechanism incurs low run-time overhead.

Index Terms—Database, Insider Attacks, Anomaly Detection, Application Profile, SQL Injection

1 Introduction

Data stored in databases is often critical to the organization's operations and also sensitive, for example with respect to privacy. Therefore, securing data stored in a database is a critical requirement. Data must be protected not only from external attackers, but also from users within the organizations [3]. A wide range of institutions from government agencies (e.g., military, judiciary etc.) to commercial enterprises are witnessing attacks by insiders at an alarming rate. The most important objective of these insiders is to either exfiltrate sensitive data (e.g., military plans, trade secrets, intellectual property, etc.) or maliciously modify the data for deception purposes or for attack preparation [1], [8], [16].

There are a number of facts that make the prevention of insider attacks more challenging compared with other conventional (external) attacks [4]. First, insiders are allowed to access resources, such as data and computer systems, and services inside the organization networks as they possess valid credentials. Second, the actions of insiders originate at a trusted domain within the network, and thus are not subject to thorough security checks in the same way as external actions are. For instance, there is often no internal firewall within the organization network. Third, insiders are often highly trained computer experts, who have knowledge about the internal configuration of the network and the security and auditing control deployed. Therefore, they may be able to circumvent conventional security mechanisms.

Protecting data from insider threats requires combining different techniques. One important such technique is represented by the access control system that is implemented as part of the database management system (DBMS) code. An access control system allows one to specify which users/applications can access which data for which purpose. In addition to the access control system implemented as part of the DBMS, applications may also perform their own "application-level" access control in order to implement more complex access control policies. In such cases, accesses by users to the data stored in a database are mediated by the application programs. However, whereas the use of DBMS-level and application-level access control mechanisms provide a first layer of defense against insider threats, these mechanisms are unable to protect against malicious insiders that have access to the applications and can thus modify the code to change the queries issued to the database and also modify the logics of the application-level access

- *L. Bossi, E. Bertino, and S.R. Hussain are with the Department of Computer Science, Purdue University, West Lafayette, IN, 47907.
E-mail: lbossi@purdue.edu, bertino@purdue.edu, hussain1@purdue.edu*

Manuscript received April 19, 2005; revised September 17, 2014.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2598336, IEEE Transactions on Software Engineering

control. Software-based attestation [32] or simple integrity measurement by a trusted platform module [23] could be used for detecting any unauthenticated change to the application source code by expert insiders. However, attestation is typically executed during the loading of the application's executable and hence it cannot detect changes of program behaviors at run-time. As a result, during execution if a program is compromised by an insider using known attack techniques, e.g., buffer overflow [9] or return-oriented programming (ROP) [26], attestation mechanisms cannot detect such malicious changes of behavior in the program. Also a malicious insider may be able to modify the information used for the attestation of the target application program, thus rendering attestation useless. Apart from that, using just a simple integrity measurement technique is not a viable solution because this technique cannot provide integrity for self modifying code (e.g., JAVA, C#) [35] which is widely used as front end database applications.

In order to address the above problem, one possible approach is to analyze the data access patterns of the application to create profiles of legitimate activities and then use at run-time these profiles to detect anomalous database accesses by application programs.

The design of such an *anomaly detection* system is challenging, as the system should fulfill the following requirements:

• It should require minimal modifications to the code of the application program and the DBMS.

• It should not introduce significant delays that may negatively impact the performance.

• It should have the least possible number of false positives and false negatives.

In this paper, we propose *DetAnom*, an *anomaly detection* mechanism able to identify malicious database transactions that addresses the above requirements. *DetAnom* consists of two phases: the *profile creation phase* and the *anomaly detection phase*. In the first phase, we create a profile of the application program that can succinctly represent the application's normal behavior in terms of its interaction (i.e., submission of SQL queries) with the database. For each query, we create a signature and also capture the corresponding preconditions that the application program must satisfy to submit the query. Note that an application program may execute different query sequences depending on different values of the input parameters. Hence, the profile of the application needs to consider all possible execution paths that lead to interaction with the database. Each query in the application belongs to one of these paths and has a set of preconditions (i.e., constraints) in order to be issued.

A major issue in our approach is that exploring all possible execution paths of an application program requires identifying all possible combinations of program inputs, which is sometimes not feasible. As a result, the unexplored paths introduce incompleteness in the application profile. The higher the number of paths explored, the more complete and accurate an application profile is. Hence, to make our profiling technique close to complete and accurate, we resort to a software testing methodology, known as concolic testing [31] [11], that ensures high coverage of the application's code as well as of the created profile. Concolic testing works with a combination of symbolic execution [6] and concrete execution. Symbolic execution is a classical software verification and dynamic program analysis technique where program variables are considered as symbolic variables and an automated constraint solver based on constraint programming logic is used to generate new concrete inputs (test cases) with the aim of maximizing code coverage. Concrete execution is commonly used for testing applications on a particular set of inputs along an execution path. As the program may have different behaviors for different values of input parameters, our approach with concolic testing generates inputs automatically to explore all such program behaviors. Note that we do not use our proposed mechanism in conjunction with concolic testing for finding bugs or verify the correctness of the program.

Using concolic testing we leverage the advantages of dynamic program analysis over static analysis which cannot detect malicious changes of program's behavior at run-time. Later in the *anomaly detection phase*, whenever the application issues a query, the corresponding query signature and constraints are checked against the current context of the application. If there is a mismatch, the query is considered as anomalous. The main advantage of our *anomaly detection* mechanism is that we do not need any knowledge about possible attacks to build the application profiles.

Note that we target our approach to securing internal enterprise software, because this is the most common category of applications which directly connect to the database. But we want to emphasize that our approach can be extended to protect also multi-tiered applications, by creating profiles of the application layer which directly communicates with the database using its API calls as *input* which can be generated by the concolic testing engine.

Moreover, we want to highlight that our goal is to protect the database by monitoring the queries submitted by clients. Even if our mechanism can identify hosts compromised by viruses or Trojan horses, this is not our main goal, because our primary goal is to detect malicious or compromised host administrators, who are supposed to access the database only through the application but who may explicitly disable or tamper host based anomaly detection tools.

In this paper, in addition to provide the details of the approach, we discuss the issues that we encountered in using the concolic testing technique and in capturing the application input at run-time to perform the anomaly detection. We also report experimental data showing the run-time performance overhead introduced by our anomaly detection technique. To the best of our knowledge, our approach is the first using software testing techniques for creating execution profiles of application programs for the purpose of detecting execution anomalies at run-time. Such anomalies may be indicative of application program tampering. Notice that our approach is complementary to techniques for static analysis. Such techniques aim at analyzing programs to detect bugs that can be exploited by attacks at run-time, such as buffer vulnerabilities. Our approach aims at preventing malicious changes to programs, after the completion of the static analysis, by insiders who have the ability to modify the application source code or the application binary.

The rest of the paper is organized as follows: Section 2 presents relevant preliminary concepts. Section 3 provides an overview of our system architecture. Section 4 describes the adversary model and explains some common attacks that our system can block. Section 5 and 6 describe the *profile creation phase* and the *anomaly detection phase*, respectively. Section 7 discusses implementation details. Section 8 analyses the security of the proposed approach. Section 9 presents an experimental evaluation of *DetAnom*. Section 10 surveys related work. Section 11 concludes the paper with a discussion on future work.

2 Preliminaries

Software Testing is the process of examining the quality of a software product. It involves monitoring the actual program execution with the goal of observing unexpected behavior (e.g., wrong output values, program crashes or early termination) which implies the existence of bugs. It can also give a perspective about the security and risks in the product or service under test. One of the main challenges in software testing is the capability of testing all possible program inputs of an application to achieve high code coverage. *Concolic testing* is one of the widely used techniques addressing this challenge.

Concolic Execution is a program analysis technique [11], [19], [31] that tries to explore all possible execution paths of a program by acting according to the following steps. The program to be tested is first concretely executed with some initial random inputs. Then the concolic execution engine examines the branch conditions along the executed path's control-flow and uses a decision procedure to find an input that would reverse the branch conditions from true to false or vice-versa. This process is repeated to discover more inputs that trigger new control-flow paths, and thus more program states are tested. This technique is particularly useful for the automatic generation of high-coverage test inputs and for software vulnerability discovery.

3 DetAnom Architecture

The system architecture consists of several components, supporting the two phases of *DetAnom*, that we describe in what follows.

3.1 Profile Creation Component

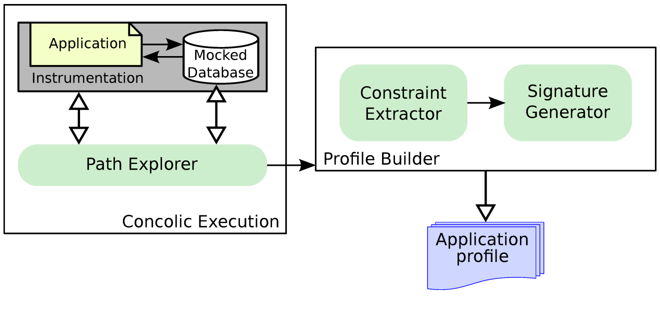

Fig. 1. System architecture for profile creation

Figure 1 shows the modules supporting the *profile creation phase* and their interactions.

This phase starts by providing the application program as input to the *concolic execution* module which first instruments the application. Note that the concolic execution does not require the application source code. The bytecode is inspected using reflection to find the branches and track the input sources to the branch conditions. Then, the application is started inside an instrumented virtual machine which links the concolic execution engine to the channels used to interact with the user. In this way the concolic engine can generate input to force the execution of different branches.

Therefore, the *concolic execution* module executes the instrumented application for a number of times with the aim of exploring as many execution paths as possible. Since there is no guarantee that the application terminates on each input, the concolic execution uses a depth bounded search to limit the profiling time. The depth of the search is a configurable parameter.

Each time the application program issues a query to the database, the *constraint extractor* in the *profile builder* module extracts the constraints that lead the application program to follow the current path. These constraints compose a part of the *application profile*. In addition, each query submitted to the database is also forwarded to the *profile builder* module where the *signature generator* sub-module generates the signature of that query.

Since the values returned by the database may change the application control flow, these values are considered as the database inputs to the application program. Hence, in order to automatically generate database inputs for concolic execution, the instrumentation library hacks the standard database connection library and mocks the behavior of the real database to let the concolic execution generating the values required to force different execution flows of the application.

Section 5 discusses details about the *constraint extractor* and *signature generator* sub-modules. Finally, the *profile builder* module binds the query signature with its corresponding constraints and inserts this record into the *application profile*.

3.2 Anomaly Detection Component

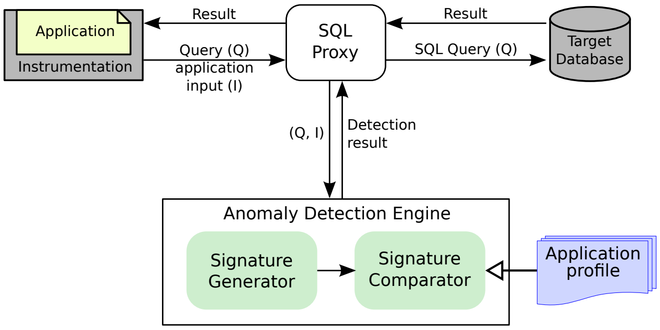

Fig. 2. System architecture for anomaly detection

The main modules supporting the *anomaly detection phase* are: the *anomaly detection engine (ADE)*, the *SQL proxy*, the *signature comparator*, and the *target database* as shown in Figure 2.

The data to protect is stored in the *target database*. We assume that the database server is already secured to the best of current security technology and can be accessed only through our proxy. The monitored application interacts with the database through SQL queries which are intercepted by the *SQL proxy* and forwarded to the *ADE* for anomaly detection. Moreover, the instrumented environment collects the application input and adds it as meta-data to the query. The *ADE* also includes the *signature generator* sub-module that generates the signature of the received query. Upon receiving the query, the *ADE* checks whether the current program inputs satisfy the constraints of some possible execution paths. If the constraints are satisfied, the *signature comparator* compares the signature of the query associated with the satisfied constraint to that of the received query. If there is a match, the query is considered legitimate, otherwise an anomaly is detected. This information is then sent back to the proxy, where a custom logic is used to decide the actions to be executed in order to manage the anomaly. Examples of such actions include rejecting the query, sending an alarm to a security administrator, revoking the application program authorizations etc.

4 Adversary Model

We assume that at run-time the application program can be tampered and thus become untrusted. Therefore, we assume that while the program is executing, the program may issue a query that:

(a) has never been encountered in the profile creation phase, i.e., the query does not belong to the application at all;

(b) belongs to the application but is not relevant to the current execution path;

(c) is relevant to the current execution path, but the program input variables do not satisfy that query's corresponding constraints.

All of these cases can be easily mapped to well known security attacks.

In case (a), an attacker may simply use a network sniffer or perform a man-in-the-middle attack to steal the credentials that the application uses to connect to the database. Once the credentials are stolen, the attacker may use any other client to connect to the database, elude all the application level security checks, and issue queries that do not belong to the application.

In case (b), an attacker may obtain the credentials as described in the previous case and can use a similar technique to record the queries that the application issues. By repeating an allowed query the attacker can pass through simpler security checks and thus can retrieve sensitive data. Let us assume that a query retrieves only a row of sensitive data after the application has performed some sanity checks on the values used to retrieve the row. An attacker may replay the query several times, changing only the values used to filter the result in order to retrieve all the data he/she wants.

In case (c), the attacker compromises the application and changes its access control policy. For example, most of the applications add an extra layer of security which requires the user to provide a pair of username and password. Usually, such applications retrieve a database table for the provided credentials to retrieve the set of permissions granted to the user. Note that this level of security is usually implemented outside of the database. All the instances of the same application use the same database credentials for the connection and handle the extra layer of security internally. If an application is compromised so to return a successful authentication, on the database side we see only a sequence of allowed queries for which the constraints may not be satisfied with the program inputs.

We assume that every component involved in the *profile creation phase* and *anomaly detection phase* is trusted. We also assume that profiles are stored in a secure storage and are not tampered by an insider or database administrator.

5 Profile Creation Phase

In the *profile creation phase*, the application program interacts with the *mock database* through SQL queries. We represent the queries internally in a specific format which we refer to as *signatures*. Queries' signatures and corresponding constraints are used to build the profile of the application. For each query, we record its signature and constraints, and refer to this pair as *query record*. All *query records* of the program are organized in a hierarchical data structure which represents the control-flow of the application. We refer to this data structure as the *application profile*.

Before explaining the application profiling technique, we discuss the model that describes the applications' normal behavior, i.e., the fingerprint with respect to the queries issued to the database. For our purpose, an application can be ideally represented using a directed graph where the nodes represent the application states in which the application issues queries to the database, and the edges represent the application inputs required to change the state. We use cycles in the graph to represent the loops in the application code.

The challenge in creating such profiles is in representing correctly the dynamic behavior of the application, as the application may change its own code, or dynamically download code from internet, or use reflection to dynamically choose which code to invoke. For this reason we use a dynamic analysis technique to create the profile.

The problem, therefore, is that when we deal with complex applications it is difficult to map the actual code to the graph representation we need. A loop in the code may dynamically creates different queries, being mapped as a sequence in the graph; while a sequence of different functions may issue the same query, being better represented using a cycle in the graph. When we create the profiles, using the concolic execution, what we do is to unroll the abstract graph recording an execution tree. This is the reason why we need a bounded search and why our profiles may be incomplete.

Following in this section, we discuss the format of the query signatures and constraints, and the detailed procedure for building the *application profile*.

5.1 Query Signature Representation

In our system, we consider a subset of the SQL Data Manipulation Statements commands. Specifically, we focus on the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands.

SQL syntax is usually represented using Backus Normal Form [20] and allow one to specify very complex queries, typically nesting them at different levels. In order to describe our query representation techniques, we organize the presentation in two parts. In the first part we describe how we create the signature of simple queries; in the second we focus on how we deal with advanced queries which contain nested sub-queries, arithmetic operators and function calls.

5.1.1 Simple Queries

Consider as example the format of a simple SELECT command:

```
SELECT [DISTINCT] {TARGET-LIST}
FROM {RELATION-LIST}
WHERE {QUALIFICATION}
```

Our system internally represents an SQL query as a signature of the form $\langle c, t, r, q, n \rangle$. Here, c represents the type of the SQL command which takes one of the values: 'S', 'I', 'U', and 'D' in case of SELECT, INSERT, UPDATE, and DELETE commands, respectively. The second field, t, is a list that contains the identifiers (IDs) of the attributes projected in the query, i.e., the attributes that appear in the query result or are modified by the query; this information is extracted from the TARGET-LIST of the query. Attributes have a unique ID among all the tables. The third field, r, is a list that contains the IDs of the tables being accessed in the query, i.e., the tables that appear in the RELATION-LIST. The next field, q, is a list of IDs of attributes referenced in the QUALIFICATION in the WHERE clause of the query. The last field, n, in the signature denotes the number of predicates in the WHERE clause.

As an example, consider the relation schema in Table 1. ID's of tables and attributes are as shown in the table. Now, consider the query:

```
SELECT employee_id, work_experience
FROM  WorkInfo
WHERE work_experience > 10;
```

The signature of the above query is:

$$\langle S, \{201, 202\}, \{200\}, \{202\}, 1 \rangle$$

We explain this signature construction in order from left to right. The leftmost S represents the SELECT command. 201, and 202 represent the IDs of attributes employee_id and work_experience, respectively. 200 represents the ID of the table WorkInfo. 202 represents the attribute used in the WHERE clause, i.e, work_experience. The rightmost 1 corresponds to the number of predicates in WHERE clause.

TABLE 1
Relation schema

Table ID	Table name	Attribute ID	Attribute name
100	PersonalInfo	101 102	employee_id employee_name
200	WorkInfo	201 202 203 204	employee_id work_experience salary performance
300	JobInfo	301 302 303	base_salary min_work_experience max_work_experience

For completeness, we briefly describe the other commands as well and we show how they differ from the main example.

The insert statement is in the form:

```
INSERT INTO {RELATION}
SET {TARGET-LIST}
```

An INSERT command can specify only one relation, that is, the table where the new values are going to be added. The target list is a list of the form of target = value where target is a column name and value is an expression that can be evaluated to the value to be added. A query signature of an insert statement has the form:

$$\langle I, \{\text{TARGET-COLUMNS}\}, \{\text{RELATION}\}, \emptyset, 0 \rangle$$

The update statement has the form:

```
UPDATE {RELATION}
SET {TARGET-LIST}
WHERE {QUALIFICATION}
```

An UPDATE statement can specify only one relation, that is, the table to be updated; the target list similar to the one of the INSERT case, with the newer values; and a qualification list, similar to the SELECT case, which specifies which rows are going to be updated. A query signature of an update statement is like the SELECT but specifies the U in the first position and has exactly one table in the relation list. As example:

$$\langle U, \{\text{TARGET-COLUMNS}\}, \{\text{RELATION}\},$$
$$\{\text{QUALIFICATION}\}, \{\#predicates\} \rangle$$

The delete statement has the form:

```
DELETE {RELATION}
WHERE {QUALIFICATION}
```

A DELETE statement specifies only one relation, that is, the table whose rows must be deleted and a qualification list specifying the rows to delete. A query signature of a DELETE statement is like the signature of a SELECT statement but specifies D in the first position, has exactly one table in the relation list and has an empty target list.

$$\langle D, \emptyset, \{\text{RELATION}\}, \{\text{QUALIFICATION}\}, \{\text{\# predicates}\} \rangle$$

5.1.2 Complex Queries

We focus on two different aspects: complex predicates in the WHERE clause and nested queries. Note that these two aspects are not strictly disjoint, because a sub-query can be nested also inside the WHERE clause.

Sub-queries can appear almost everywhere a value can appear. For example, the following query returns a list of employees with their working experience and the overall company maximum salary. This query includes a sub-query as part of the projection clause, that is, the list of data to be returned by the query.

```
SELECT employee_id, work_experience, (
  SELECT max(salary) FROM WorkInfo
) as maxSalary
FROM  WorkInfo
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2598336, IEEE Transactions on Software Engineering

Sub-queries can also appear in the `WHERE` clause. For example, the following query uses a nested query to retrieve the highest salary and uses this value to select the set of employees who earn it.

```
SELECT employee_id
FROM  WorkInfo
WHERE salary = (
  SELECT max(salary) FROM WorkInfo
)
```

Sub-queries can appear also in the `FROM` clause. In this example, a virtual table is materialized that contains the total salary paid for every performance level, and such table is used to check the quota that every employee earns compared to his/her performance level.

```
SELECT employee_id, performance,
    salary/total
FROM WorkInfo, (
  SELECT sum(salary) as total,
        performance as per_group
  FROM WorkInfo
  GROUP BY performance
) as SalaryInfo
WHERE performance = per_group
```

Eventually, sub-queries may use tables and columns used in the outer queries and mix query types. In the following example, the base salary is updated according to the average salary of the employees.

```
UPDATE JobInfo SET base_salary = (
  SELECT avg(salary)
  FROM WorkInfo
  WHERE min_work_experience <
      work_experience AND
       work_experience <=
           max_work_experience
)
```

Note that the inner query accesses two columns, `min_work_experience` and `max_work_experience`, of the table `JobInfo` which is not declared in its `FROM` clause, but in the parent's one. Thus, the inner query signature contains such columns ID but not their table ID, as shown by the following signature:

$$\langle S, \{203\}, \{200\}, \{302, 202, 303\}, 2\rangle$$

To create the global query signature we nest signatures as they appear in the query. Thus, the complete signature of the query in the last example is:

$$\langle U, \{301, \langle S, \{203\}, \{200\}, \{302, 202, 303\}, 2\rangle\}, 300, \emptyset, 0\rangle$$

Another way to create complex queries is to use functions or operators to manipulate data, as we can see in the following example.

```
SELECT *
FROM WorkInfo
WHERE filter(performance,
      work_experience / salary)
```

The `WHERE` clause of this query contains a custom function which returns a Boolean value starting from the performance and the work experience over salary ratio, which is obtained by using the division operator over two different columns. In this example we can see clearly why the profile contains both the columns used in the `WHERE` clause and the number of predicates. Even if such values are strictly correlated in trivial cases, it is important to know both in order to identify more complex queries. The signature of the query in the last example is:

$$\langle S, \{201, 202, 203, 204\}, \{200\}, \{204, 202, 203\}, 1\rangle$$

5.2 Concolic Execution

This section describes the basics of concolic execution used during the *profile creation phase* to explore the possible execution flows of the application.

The concolic execution takes the application as input and instruments it to log each operation that may affect a symbolic variable value or a path condition. This module then executes the program concretely with some initial default input. In order to explore other paths, it examines the branch conditions (i.e., constraints) along the executed path, and uses a constraint solver to find inputs that would reverse the branch conditions. The execution is repeated for a number of times until all the execution paths are explored or the depth search limit is reached in all the explored ones.

Considering that the goal of our system is to protect a database, we expect that the instrumented application issues queries along some of these execution paths. The issued queries are forwarded to both the *profile builder* and the *mocked database*. Upon receiving a query, the *constraint extractor* sub-module in the *profile builder* extracts the constraints that are prerequisite to execute that query. The *mocked database* uses the concolic engine to generate the query results that are required to explore newer execution paths.

We want to highlight that the queries are captured at the time when they are sent to the database; therefore we can create correct profiles even when queries are built by concatenating strings. For example, consider the following code:

```
String query = "SELECT ";
switch (what) {
 case 1: query += "name, surname ";
    break;
 case 2: query += "address ";
    break;
 default: query += "* ";
}
query += " WHERE ssn='" + ssn + "'";
s.executeQuery(query);
```

Depending on the value of the variable `what`, three different queries can be executed. During the profile creation, the constraint extractor collects the constraints until the `executeQuery` is executed and the signature generator gets actual string values as passed to the same function. Therefore, in this example, three nodes will be added to the profile, one for every concrete query that can be issued.

Before explaining in detail how the profiles are created, it is important to discuss about how the query signatures are extracted. The problem is that in order to create meaningful signatures it is necessary to know the database schema. Consider the following query:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2598336, IEEE Transactions on Software Engineering

```
SELECT a, b
 FROM t1 join t2
 WHERE t1.c = t2.c
```

Without knowing the schema it is impossible to map the columns a and b to their respective tables.

Therefore the *signature generator* module requires a setup which specifies the tables used by the application. For every issued query, this module parses the query to identify the composing tokens and resolves the columns name to match them to their tables. Once all the columns are correctly resolved, the query signature can be created according to the approach introduced in Section 5.1.

5.3 Profile Creation

In this section, we describe in details, with a running example, the profile creation phase: how the concolic engine executes the application; how the user input and the mocked database are used to explore newer paths; how the profile builder creates the *query records* and composes them to create the profile.

The definition of application profile is as follows:

Application Profile: The profile of an application program P is a directed tree $T(V_P, E_P)$. Each node $v_i \in V_P$ is a *query record* of query q_i represented as $\langle sig(q_i), c_i \rangle$, where $sig(q_i)$ is the signature of q_i, and c_i is the set of constraints to execute q_i. An edge $e_{ij} \in E_P$ denotes that the query q_j is executed after query q_i and hence, node v_j is a child of node v_i.

To illustrate the profile creation procedure, we continue with the examples given in Section 5.1 and 5.2, following the code shown in Figure 3.

```
1  public static void salaryAdjustment(int
       profit, int investment){
2     Statement s;
3     ...
4     int employee_count = 0;
5     if(profit >= 0.5 * investment){
6        String query1 = "SELECT employee_id,
            work_experience FROM WorkInfo
            WHERE work_experience > 10";
7        resultSet1 = s.executeQuery(query1);
8        resultSet1.last();
9        if(resultSet1.getRow() > 100){
10       String query3 = "SELECT employee_id
            FROM WorkInfo WHERE
            work_experience > 10 AND
            performance = 'good'";
11       resultSet3 = s.executeQuery(query3);
12       ... // do other operations
13       } else{
14          String query2 = "UPDATE WorkInfo
               SET salary = salary * 1.2";
15          s.executeUpdate(query2);
16    }else{
17       String query4 = "SELECT
            p.employee_name FROM
            PersonalInfo p, WorkInfo w WHERE
            performance = 'poor' AND
            p.employee_id = w.employee_id";
18       resultSet2 = s.executeQuery(query4);
19       ... // do other operations
20    }
21 }
```

Fig. 3. An example of database application

The concolic execution takes the java bytecode, instruments it to find the branch conditions, and executes it inside the instrumented environment. Consider the program in Figure 3 that asks the user to input the values profit and investment and passes them to the function salaryAdjustment. The concolic execution uses the environment instrumentation of such program to block the interactive requests by passing to the program automatically generated values.

The first time a numeric variable is encountered, the value returned is 0. Therefore, during the first execution, the function is called with both parameters set to 0. Following the code, at line 5, the condition will be evaluated true and the *constraint extractor* will compute the constraint c_1 as shown in Table 2. Once query1 is submitted, the profile graph node $QR_1 = \langle sig(query_1), c_1 \rangle$ is created and added as the first child of the root of the application profile as shown in Figure 4(a).

In this phase, the queries do not reach the real database, but they are blocked by the environment instrumentation which uses the concolic engine to generate the returned values. In the first execution, the instrumentation will return 0 rows; therefore the condition at line 9 will be evaluated to false, generating the constraint c_2, issuing the query query2 and creating a new node in the profile as shown in Figure 4(b).

At this point nothing is left to do in the function. Assuming that the application ends too, the concolic execution backtracks the execution to the last jump encountered with an unexplored branch. In this case it is the if statement at line 9. Therefore a new value for the variable is generated in order to negate the previous result and explore the new branch. In this case it means that the concolic engine must return 101 rows as result of the query1. Therefore query3 is executed and its features together with the constraint c_3 are added into the profile as shown in Figure 4(c).

Finally, the concolic execution has left unexplored only the else branch of the if at line 5. Therefore it uses a constraint solver to find values of profit and investment which negate the conditions. Assuming that it sets profit and investment variables to 49999 and 100000 respectively; the constraint c_4, as shown in Table 2, is then generated; query4 is executed and the node $QR_4 = \langle sig(query_4), c_4 \rangle$ is added into the profile as child of the root as shown in Figure 4(d).

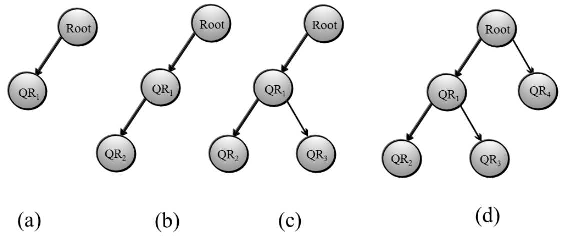

Fig. 4. Steps of profile graph construction

Note that the environment instrumentation blocks the execution of this query too and lets the concolic engine to generate the returned data. The concolic execution will try to explore all the possible paths starting from the unwritten code at line 19, but since none of these will issue any new query nothing is recorded inside the application profile. At this point, if the concolic execution has to stop because the maximum search depth has been reached, the node QR_3 will be marked as *incomplete*, otherwise it will be marked as *complete*.

At this point, since the *concolic execution* module has completed exploring the execution paths, the *profile creation phase* ends.

Table 3 and Table 4 show respectively the query signatures and the query records generated to represent the example application. Table 2 shows all the constraints generated profiling the application shown in Figure 3. Note that the constraints do not contain meaningful names for the variables. This happens because the concolic execution works using the compiled application and the variable names are not stored inside the bytecode. Therefore the concolic execution identifies variables just by their order of appearance.

TABLE 2
Constraints for queries

c_1	arithmetic: $1.0\ x_1\ -\ 0.5\ x_2$ >= 0.0
c_2	database: $x_3\ \leq$ 100.0
c_3	database: $x_3\ >$ 100.0
c_4	arithmetic: $1.0\ x_1\ -\ 0.5\ x_2$ <= -1.0

TABLE 3
Query signatures

Query	Signature
$query_1$	{S, {201, 202}, {200}, {202}, 1}
$query_2$	{U, {203}, {200}, ∅, 0}
$query_3$	{S, {201}, {200}, {202, 204}, 2}
$query_4$	{S, {102}, {100, 200}, {204, 101, 201}, 2}

TABLE 4
Query records

Query record	Contents
QR_1	$\langle sig(query_1), c_1\rangle$
QR_2	$\langle sig(query_2), c_2\rangle$
QR_3	$\langle sig(query_3), c_3\rangle$
QR_4	$\langle sig(query_4), c_4\rangle$

Note that the concolic execution uses a bounded depth-first search strategy to explore the execution paths. However, due to possible failures in solving complex constraints, the concolic execution may not be able to actually explore all execution paths of a large and complex application, thus leaving the profile incomplete. We handle this situation as follows.

During the concolic execution, we stop exploring a path when we reach either the maximum depth search limit or the end of a path whose depth is smaller than the maximum limit. If we reach the maximum limit and stop exploring that path, we mark the last state as incomplete. Note that a node can be incomplete without any regard of its number of children or distance from the root, because the depth used by the concolic execution counts the number of branches, while the depth of the profile tree counts the number of queries.

Knowledge about which nodes are incomplete is necessary for the detection phase. Unfortunately, we cannot infer the minimum depth to use to completely profile an application because this can be reduced to the halting problem. Therefore, we deal with the problem of incomplete profiles as follow. If a profile contains too many incomplete nodes, the administrator may decide to create the profile again by increasing the limit of the search, or to manually compute it using the logs obtained after the program execution.

6 Anomaly Detection Phase

We now describe how application program profiles are used to distinguish between legitimate and anomalous database queries. The steps of the *anomaly detection* procedure are presented in Algorithm 1.

Algorithm 1 Anomaly Detection

```
Input: Application Profile (AP)
$v_p$ = root of $AP$
while the program is executing do
    $q$ = issued query
    $c_i$ = input constraints
    signature generator generates $sig(q)$
    $found$ = false
    for each child $v_i$ of $v_p$ do
        if $c_i$ is satisfied then
            signature comparator compares $sig(q)$ to $sig(query_i)$
            if signatures match then
                response: NOT-ANOMALOUS
                $v_p$ = $v_i$
            else
                response: ANOMALOUS
            end if
            $found$ = true
            break
        end if
    end for
    if $found$ == false and $v_p$ is an incomplete node then
        response: WARNING
    end if
end while
```

6.1 Detection of Anomalous Queries

In the *anomaly detection phase*, whenever the application program issues a query, the proxy module intercepts and forwards it to the *ADE* module.

When an application program starts executing in the *anomaly detection phase*, the *ADE* module sets the root node of the *application profile* as the current parent node (v_p). Upon receiving the first query along an execution path of the program, the *ADE* considers all the children of v_p as *candidate nodes*. The *ADE* then takes the inputs from the executing application and for each *candidate node* it verifies whether the inputs satisfy the constraint in the *query record*. If the inputs satisfy constraint c_i, the program is expected to execute the query which is associated with the query record QR_i containing the satisfied c_i. As next step, the *signature* *generator* sub-module generates the signature of the received query and the *signature comparator* sub-module compares it with the signature stored in QR_i, i.e., $sig(query_i)$. For a legitimate query, the signatures match. The verification outcome is then passed to the proxy module which then sends the legitimate query to the *target database* for execution.

For subsequent queries issued by the program, the *ADE* module considers the *query record* of the most recently executed query as the current parent node, and verifies the signature and corresponding constraints in a similar way as described above.

As we already discussed, during the profile creation phase we use a depth bounded search to explore the execution paths. So it is possible to have incomplete profiles. This is the reason the *ADE* module can return three different results: `NOT-ANOMALOUS`, `ANOMALOUS` and `WARNING`.

During the profile creation, we know when we do not enable the backtrack because we reached the maximum search limit. Therefore we mark the last node as incomplete. When we receive a new query to analyze, if we cannot find any matching result we check if the last status was an incomplete node. If this is true, it means we are entering in an unseen state that may receive unexpected queries. In this case we return a warning, because our system is unable to decide if the query is anomalous or not. It is the duty of the SQL proxy to decide how to handle this case.

Ideally, whenever a program creates too many warnings in our system, an administrator should verify and edit the profile to fix the problem, or create a new profile using a deeper search.

6.2 Case Studies

In this section we present some case studies to illustrate how the *ADE* module works in the *anomaly detection phase*. We assume that the values of `profit` and `investment` variables are set to 60000 and 100000, respectively. We consider the following cases.

6.2.1 Execution of $query_1$ and $query_2$

According to the values of input variables, the application program is eligible to issue $query_1$. So in the *anomaly detection phase*, upon receiving the issued $query_1$, the *ADE* module takes the program inputs to check whether they satisfy the constraints of either QR_1 or QR_4. As c_1 is satisfied, the *signature generator* sub-module generates the signature of the input query and the *signature comparator* sub-module compares it with the signature part of QR_1. The match is positive and hence $query_1$ is assessed as non-anomalous. Now assume that the number of records returned by $query_1$ is less than 100. In this case, the constraint c_2 is satisfied and the attempt to execute $query_2$ is considered non-anomalous because the signature of $query_2$ matches to that of the QR_2.

6.2.2 Execution of $query_1$ and $query_3$

In this case, $query_1$ is executed legitimately as described in the previous case. Afterwards, when the program issues $query_3$, the *signature comparator* sub-module finds that the signatures of $query_3$ and that of the expected query do not match. As a result, the *ADE* module raises an alert indicating $query_3$ as anomalous.

6.2.3 Execution of a query that does not belong to the program

If a query is issued in the *anomaly detection phase* that has been never encountered in the *profile creation phase*, the signature of this query does not match with any of the query records. In this case, if the query is issued in a state that is profiled completely, the *ADE* module raises an *anomaly*. However, if the program execution reaches a state where the profile is incomplete (because the maximum depth search was reached during the testing), the *ADE* module raises a *warning*.

6.2.4 SQL injection attacks

Halfond et al. [14] classified different types of SQL injection attacks. The tautology attacks are the easiest to detect, because they introduce tautologies in the `WHERE` clause that can be easily detected by a simple SQL parser. Some attacks consist of sending illegal queries, because by analyzing the resulting error messages it is possible to infer meaningful information about the schema and the type of database. Such attacks can be identified by analyzing the error logs. The remaining attacks use special encoding or unescaped parameters to alter the executed query.

It is important to note that, as these attacks typically modify the queries by adding new predicates, they can be easily detected by our anomaly detection mechanism because the query signature contains both the columns used and the number of predicates in the `WHERE` clause.

We illustrate the detection of SQL injections with a sample application program. Such program has the function of displaying the medical records of an authenticated signed user. The user is authenticated by entering his `username` and `password`. The legitimate query execution would look like:

```
username = readInputUser();
password = readInputPassword();
query = "SELECT * FROM MedicalRecords
    WHERE uname = '" + username + "' AND
    password = '" + password + "'";
```

If the username is `John` and the password is `Smith`, then the query would be:

```
SELECT *
  FROM MedicalRecords
  WHERE uname = 'John' AND password = 'Smith';
```

However, such query is vulnerable to SQL injection attacks by which the attacker can display the medical records of other users. This can be achieved if the attacker enters in the `password` input field the string `password = ' OR uname = 'Carl'`. If so, the following query would be issued which would display the medical records of the username Carl to the attacker.

```
SELECT *
  FROM MedicalRecords
  WHERE uname = 'John' AND password = ' ' OR
      uname = 'Carl';
```

Such a vulnerability exists in any application that allows the user input to change the structure of an SQL query. Since

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2598336, IEEE Transactions on Software Engineering

SQL injection attacks are based on re-structuring the SQL query, our mechanism by comparing the query structure to the query signatures saved in *application profile* is able to detect changes in the query. More specifically, as we count the number of predicates of the WHERE clause as part of the query signature, we are able to detect any additional predicates introduced by SQL injection. In the example above, the number of predicates is 2 before the injection, and it becomes 3 after the injection.

6.2.5 Two-step SQL injection attacks

These attacks are also referred to as second-order injection attacks and represent a complex form of data-centric attacks. The purpose of these attacks is to create an SQL injection attack that can be processed at a later time. This is achieved by injecting malicious input into the database that is legitimately saved into the database, but will result in an SQL injection attack at a later time when other types of queries perform actions on the maliciously inserted data. To clarify, consider an example of a web application that registers its users upon using their service. If a malicious user chooses (’OR ’1’ = ’1’) as his username, then adding this user to the database will execute the SQL query:

```
INSERT INTO users VALUES ("' OR '1' = '1'");
```

This is a legitimate query and will not result in an SQL injection attack, and thus the username ’ OR ‘1’ = ‘1’ will be successfully created. However, if at a later time the malicious user or even the web administrator decides to delete this account, the executed SQL query is:

```
DELETE FROM users
  WHERE uname=' ' OR '1' = '1';
```

This is when the attack is effective as the query will result in deleting all the users in the database.

Our AD mechanism will be able to detect this type of attacks when the SQL injection is about to perform the intended attack action on the database. Consider the example above. Our AD mechanism will find a mismatch with the DELETE SQL query signature because of the change in the number of predicates in the WHERE clause. As a result, the *ADE* will assess the execution of such query as anomalous. Like the case of SQL injection attacks, additional predicates will result in a mismatch of the SQL injected queries when compared to the existing query signatures and therefore will result in the query being identified as anomalous.

6.3 Simple Detection

Instrumenting all the instances of the application to be secured is not always possible. Possible reasons may be related, but not limited, to:

- time constraints as the application may be already deployed on a large number of machines and update all of them may not be easy;
- technical reasons as the environment used may not expose any API for the instrumentation (i.e. JVMs for mobile devices);
- performance reasons as in applications with high numbers of user interactions, the overhead introduced by sending the user input may introduce significant delays.

Therefore we have also developed a simple version of our anomaly detection approach which does not require the instrumentation for the anomaly detection phase. We refer to this approach as *simple detection*, whereas we refer to the previous approach as *complete detection*.

Since the profile creation phase does not change, the same profile can be used for both the simple and complete detection phases. The main difference is that, without receiving the application input during the anomaly detection, we can verify only that an allowed sequence of queries is issued but without checking the constraints.

Hence, the simple approach can still verify that only allowed queries are submitted in the right order, but cannot enforce any longer that the sequence of queries is consistent with the input received by the application. Therefore, for example, a control flow attack that modifies the code from

```
if ( userInput() == 'y' ) {
  // delete a record
}
```

to

```
if ( true ) {
  // delete a record
}
```

cannot be detected any longer.

7 Implementation

In this section, we discuss the implementation details of the proposed system. In our implementation, we consider applications in the form of Java bytecode, which is mostly produced by compiling Java source code, but can also be generated starting from other languages, most notably Scala. However, our proposed *anomaly detection* mechanism can be used for other kinds of application programs.

7.1 Constraint Extractor

Our implementation of the *constraint extractor* is built on top of the JCute concolic testing framework [30]. This framework uses Soot [37] for instrumenting Java class files and lpsolve for solving linear programs.

In the profile creation phase the concolic execution engine takes the application and the depth search limit as inputs and instruments the application using Soot for branch analysis and backtracking. We instrument the runtime environment to generate user input and database content that direct the concolic execution to visit different branches of the application. Once all the branches have been explored, or the concolic execution has reached the maximum depth search limit, the concolic execution and the profile creation phase ends.

Our custom instrumentation allows one to capture all the queries issued to the target database. Every time a new query is issued, the *constraint extractor* first captures the constraints of the current path from the root to the intercepted query. Since the *constraint extractor* knows the constraints of the most recently executed query along this path, it extracts only the constraints that are extension of those of the most recent query and stores them into the *application profile* along with the signature of the intercepted query.

7.2 Signature Generator

We use PostgreSQL-9.1.8 [22] to implement the *signature generator* module. PostgreSQL delivers all issued queries to the parser to generate a query parse tree using the method `exec_simple_query()`. In this method, our customized function for *signature generator* imports necessary query information (command, target list, relation list, and qualifiers) from the parse tree and creates the query signature.

Application Profile: The *profile builder* module creates *query records* by combining the query signatures generated from *signature generator* and the constraints extracted by *constraint extractor*. These records are organized according to a hierarchical data structure (see Figure 5) and stored in PostgreSQL.

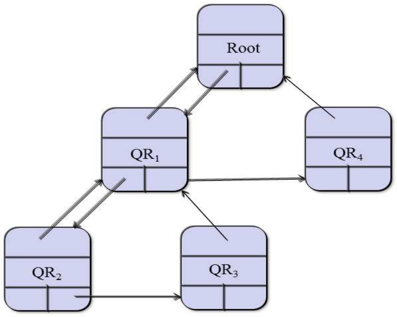

Fig. 5. Profile graph

7.3 SQL Proxy

The SQL Proxy module is in the charge of intercepting the queries before they reach the database, forwarding them to the *ADE* and handling the query responses.

The proxy is written so to offer an interface that is binary compatible to the target DBMS. Therefore a plugin must be implemented for every supported DBMS.

Since not all the database protocols support sending custom meta-data along with the query, our SQL proxy takes care of discarding the custom meta-data and the user input before forwarding the query to the target database.

It is important to note the the proxy can filter the connection in both the directions; therefore it is also possible to alter the returned data in case an anomaly is found.

The proxy is also in charge of taking actions in response to an anomalous query. As the most suitable response to anomalous query depends on many application related factors, the proxy provides a policy language by which the system administrator can define customized actions. Since not all the anomalous queries are attempts to steal sensitive data, in order to select a response the administrator may need to consider other factors, e.g., whether the query is generated during working hours, or from a trusted computer, whether the query retrieves some low relevance data, or whether the query is issued during the weekend, or from an intranet, or whether query is trying to retrieve sensitive data. Based on these factors, the application may be required to disconnect immediately before the data is returned.

Since the proxy filters the data in both directions, one can decide for performance reasons to let the query reach the target database as soon as it is recorded by the proxy and block the result returned by the query in case of an anomaly is detected. Obviously this can be safely and easily done only for queries that do not update the database.

Our current implementation consists of a Java program that supports the last version of Oracle. Whenever an anomaly is found, the query is put in hold and a message is prompted to an administrator asking the action to perform. The valid actions that can be chosen are: 1) log the query, 2) drop the query, 3) close the connection, and 4) redirect the program to an honeypot.

7.4 Architectural Techniques for Capturing Program Input

The system relies on the ability to instrument applications in order to inject custom code to generate – during the profile creation phase – or capture – during the anomaly detection phase – the user input.

Though our implementation focuses on Java, the same technique can also be applied to other application platforms and environments.

There are two well known approaches to instrument an application program: change the application itself or change its working environment. In our scenario we consider that we do not have the source code of the application; therefore, changing the application itself means having to change the compiled code (i.e., the binary of the application) or using some decompiling mechanism. Even if this is technically possible, it raises the following issues. The application binary may be obfuscated [2]; therefore, analyzing the control-flow to find the places where the application input is collected is not a trivial task which may even introduce bugs in the software. Another issue is that, especially in high security environments, the applications may require to be digitally signed. Since such kind of instrumentation breaks the application signature, a security check may block its execution. Taking into account of all these issues, we decided to adopt the second approach: instrumenting the environment.

The Java working environment can be easily instrumented by changing the Java Virtual Machine that is in charge of interpreting and translating the bytecode into the machine language. This can be easily done, in most of the virtual machine implementations, using the non-standard option `-Xbootclasspath/a:path` which let the user to override the standard classes with a custom provided jar file. The main advantages of this approach is that we do not change the application itself; therefore we do not break any signature based security check. Moreover, since OpenJDK is the reference implementation for Java SE, it is very easy to find the source code of the classes that we need to instrument and change them without the help of any decompiler. Last but not the least, by instrumenting the whole environment we can rewrite the database connection library to mock the behavior of a database. Implementing a fake

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2598336, IEEE Transactions on Software Engineering

database in software allows us to create the profiles of the application program faster and also to change the database content easily to explore new execution paths by the *concolic execution* module during the profile creation phase. We have thus created a new JDBC library which mocks the database by returning data generated by the concolic engine and using the instrumentation to replace, during the profile creation phase, the selected database connection library with our custom code. During the execution of the complete detection, we use the instrumentation to wrap the used JDBC library with a custom code which adds the user input as meta-data every time a query is sent to the database.

This approach is not free of drawbacks. The biggest challenge we had to solve is that, by instrumenting the whole environment, we do not change just the application behavior but also the behavior of all the libraries that it uses. Most of the time this is the desired behavior, but there are few exceptions. The problem is that Java input classes read from *streams* without knowing the real source of the data. When we instrument a class – i.e. `BufferedReader` – we do not know if the actual instance is reading from the console, from a file, from a network stream or if it is used as a wrapper for another already instrumented class. In order to filter out the input that we do not need, an inspection of the stack is required with a good heuristic that should be tuned according to the tested application.

Note that a complete instrumentation library must consider all the classes and functions that can be used to collect user input. As it is easy to guess, this is a very large set because it includes most of the GUI components; all the methods to read from network, files and console; uncommon input sources, like accelerometers, joy pads or any other set of sensors and so on. Even if providing a reasonable complete implementation of the instrumentation library is possible, it is outside the scope of our work. For this reason we limited our implementation to the input sources necessary for our tests only.

Concluding, we would like to mention again that our approach was tested on Java, but it can also be easily implemented in other languages. For example, the `LD_LIBRARY_PATH` can be used to instrument native Unix applications, forcing them to use custom libraries instead of the default ones.

8 Security Analysis

In what follows we analyze the security of our proposed system.

The attacker cannot execute queries that do not belong to the application: *DetAnom* enforces a policy that any query outside of the application is considered as `ANOMALOUS` or `WARNING`. Our approach checks the signature of the issued query against the signature stored in the profile. If the signatures do not match, the issued query is considered as `ANOMALOUS`. If the program reaches the maximum depth search limit and issues a query, the *ADE* generates an `WARNING` message and holds that query until the security administrator resolves the issue.

The attacker cannot execute a query that is irrelevant to the current execution path: If an attacker has knowledge about an allowed query, he/she may repeat that query a number of times to retrieve all the sensitive data. *DetAnom* detects such attempt by following the profile graph which maintains the order or the sequence of the queries. If the issued query is out of the order with respect to the current execution context, *DetAnom* flags the query as `ANOMALOUS`.

The attacker cannot execute a query that is relevant to the current execution path, but for which the program inputs do not satisfy the constraints: An attacker can exploit any vulnerabilities of the application and change the application level access control policy. In this case, the attacker may execute a query that is relevant to the current execution context but the input values do not satisfy the constraints. Therefore, whenever a query is issued, *DetAnom* checks whether the constraints associated with the candidate nodes (i.e, nodes which are reachable from the current state) are satisfied by the program inputs. If constraints are not satisfied the query is flagged as `ANOMALOUS`.

The attacker cannot tamper/change the profile: Our proposed approach stores the profile in the *ADE* which is outside the scope of the attackers. Only the security administrator can access the profile. However, we enforce a separation-of-duty policy to prevent any malicious security administrator from tampering the profile.

9 Experimental Evaluation

We have evaluated the performance of our proposed *DetAnom* mechanism. Our experiments have been performed on a virtual machine running Ubuntu-14 as operating system, with 10GB of RAM memory and 4 processors.

Considering the deterministic behavior of our approach, and considering that in case of a control-flow attack we expect to find all the queries after the attack to be flagged as anomalous, we focused the evaluation on the performance and the overhead required to send the user input and verify the constraints.

Since to the best of our knowledge there is no public available dataset suitable for our needs, we generated some test applications. The goal was to test *DetAnom* using applications with different size, in order to check the behavior in case of partial profiles. The details of such applications are listed in Table 5, ordered by increasing complexity; the first two use only binary branches, while the third contains also for loops. As we can see in the second column, the profile creation time increases very fast. The reason is that in the worst case this time is exponential in the number of branches. A limitation of the concolic testing tool we use is that the backtrack support is not implemented. Therefore every time a new branch has to be explored, a new execution of the application is required. Considering that we generated the test applications nesting binary branches evenly, profiling an application with an extra "if-else" requires twice the time. Adding loops slows down even more the profile creation because, as explained in Section 5, jCute actually unroll loops that can be seen as a series of nested "if"s where every "if", but the last one, contains the loop body and the next if.

To test the applications, a pseudo random input generator has been used to simulate the user input. Initializing the generator with the same seed makes it possible to test the same execution flow. We analyzed 100 different execution

TABLE 5
Test application details

Test application	Profile time	Code coverage	Profile size (nodes)	Number of "if"	Number of "for"	Number of nested blocks	Number of unique queries	Line of code
#1	40 seconds	100%	103	7	0	3	14	235
#2	5 minutes	70%	351	15	0	4	30	283
#3	4 days	20%	319	37	32	8	106	543

flows for each application. For each execution flow we recorded the execution time and the network usage of the application in both a normal execution and an execution protected by *DetAnom*.

Figure 6 shows the average execution time of the applications compared to their average execution time with our anomaly detection system enabled. As we can see the runtime overhead is small and around 20%. We can also notice that the average execution time of the longer application is just few milliseconds higher than the execution time of the smaller one. The reason is that the time required to start the JVM is considerably higher than the time required to send a query.

Figure 7 shows the network overhead introduced in order to send the application input to the anomaly detection engine in order to check the path constraints. We can see that in the first two applications the overhead is between 30% and 40% whereas in the third it reaches 60%. Such results match our expectation that larger size programs have more complicated control flows and therefore require more data to be transferred to check the constraints. Even if these percentages may appear very high, we should consider that the absolute values are small. In the last test the average overhead was only of 5.6 kilobytes. Whereas in the first two is respectively 1.1 and 1.2 kilobytes. Moreover, it is important to point out that the network overhead is not related to the amount of data transferred between the database and the application, but only to data required to transfer the application input to *DetAnom*. In our tests we used a very small database. We expect, however, that in real applications the actual data retrieved by the database is higher than the few rows retrieved by our tests. Therefore the overhead introduced by the transmission of the application input to *DetAnom* will likely be negligible compared to the overhead incurred by the transmission of the query results from the database to the application. In case, however, of applications retrieving very small datasets and for which the transmission overhead introduced by DetAnom may be too high, the simple detection mechanism can be used at the cost, however, of a weaker detection (see Section 6.3). The standard deviation of the last test is very high because this is the only test application that contains loops; therefore, depending on the input, the application may send more queries just because iterates more on some code block.

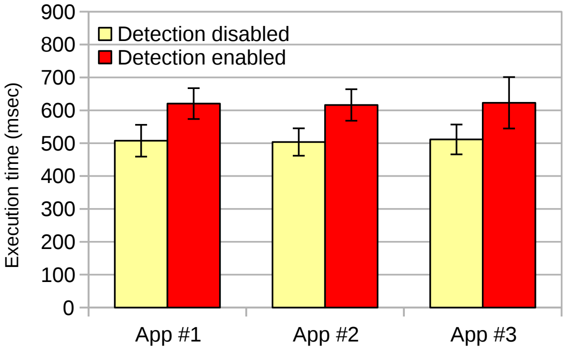

Fig. 6. Execution time overhead

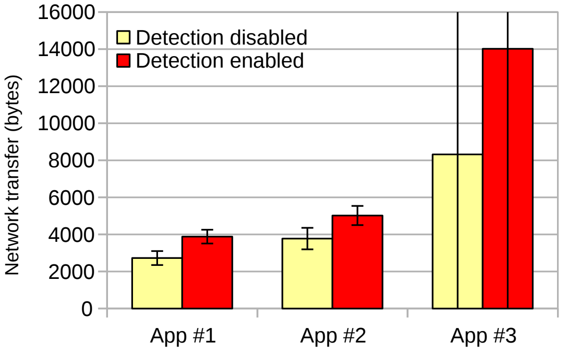

Fig. 7. Network overhead

9.1 Accuracy Limitations

Considering the query signature used and the anomaly detection techniques implemented – described respectively in Section 5.1 and 6.1 – we now analyze in detail what kind of attack cannot be detected by *DetAnom*.

As already discussed, we expect to have false positives only in case of incomplete profiles. Therefore, in this section we discuss only about the false negatives, that is, anomalies that *DetAnom* is unable to detect and approaches to address these types of false negative occurrences. It is important to notice that false negatives are due to the level of details according to which queries are represented in the profiles. In what follows we discuss the two limitations of the query signatures and techniques to address these limitations.

Consider the following fragment as example of code to attack.

```
int productivity = userInput();
sql = "UPDATE employee SET salary = salary
    * 1.1 WHERE productivity > " +
    productivity + " AND work_experience >
    5";
```

The query signature does not contain any information about the operators used in the `WHERE` clause. However changing such operators changes the semantics of the query. In the following example the operators "and" and "greater than" have been replaced respectively by "or" and "less than", respectively.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2598336, IEEE Transactions on Software Engineering

```
int productivity = userInput();
sql = "UPDATE employee SET salary = salary
    * 1.1 WHERE productivity < " +
    productivity + " OR work_experience >
    5";
```

Such change in the query can be easily detected by extending the signature to record the type of the operators together with the relative occurrence of these operators within the query. The occurrence of an operator can be determined based on a traversal of the query parse tree. Another approach is to include information about predicate selectivities. As well known from the large body of research on query optimization (see [29] for the pioneer work on query optimization), different operators result in different query selectivities. The selectivity of a query gives an indication of the expected cardinality of a query result. For example, a query with the logical "or" of two predicates returns more tuples than a query with the logical "and" of the same two predicates. Therefore, if the expected selectivity of the query is recorded in the query signature, changes to the operators would result in a different selectivity. As a result, an anomaly due to mismatch in query selectivity would be detected. We have developed an initial prototype of this technique for the case in which queries are issued directly by users [28]. We plan to further tune this technique and integrate it into *DetAnom* as part of future work. We did not include this technique in the current release of *DetAnom* to keep the system stable for release to our industry collaborators.

The query signature does not contain any information regarding the parameters used to compose the query. Therefore changing a parameter generates an anomaly that cannot be detected. In the example below a user input is ignored and replaced by a fixed value.

```
int productivity = userInput();
productivity = 0;
sql = "UPDATE employee SET salary = salary
    * 1.1 WHERE productivity > " +
    productivity + " AND work_experience >
    5";
```

A first line of defense against this attack is to add to the database some triggers which check that sensitive parameters are within a valid range. A second line to defense is to extend the instrumentation to check that the user input correctly reaches the SQL connection library; however, this in turn requires protecting the instrumentation from tampering. A third line of defense is to identify, during the profile creation phase, relationships (such as equality or some other mathematical relationship) between the input parameters of the application program and the parameters passed to the query. These relationships can be identified via some statistical analyses. At run-time, the actual values of the application input parameters and the values of the parameters passed to the query would analyzed to determine whether the relationships still hold. When this is not the case, an anomaly would be raised. We notice that these three defense techniques could be all applied to provide a strong defense against attacks that change the input parameters of queries.

Checking that user input is not arbitrarily changed is a general problem outside the scope of this project. We plan however to investigate this issue as part of future research to also determine available software engineering techniques that can help with this problem when attacks are carried out by insiders. We emphasize that insiders may have direct access to source and binary of applications and therefore would be able to compromise an application even when the application does not have any vulnerability (like buffer overflow).

9.2 Technical limitations

Our experiments have shown some technical limitations of our current approach. In what follows we discuss such limitations and outline possible solutions.

The profile creation phase is very slow. This is a limit of the testing technique we use which actually runs the program as many times as it is required to explore the possible execution paths. Moreover, the concolic testing tool we used, JCute [30], was developed to write small unit tests. Therefore it does not implement any mechanism to speed up the analysis of large programs. Multiple executions could be parallelized and distributed on different machines; moreover, saving a snapshot of the execution in order to being able to backtrack without the need of restarting the application from the beginning may result in a big improvement of the profile creation time. The use of a concolic engine, which supports backtracking doing snapshot of the application execution, may also be useful in supporting the incremental profile creation. This can be used both to quickly deploy a partial profile to start securing the application while building a more accurate profile, or to incrementally change the profile to reflect application updates.

Our current approach to deal with application updates is that an administrator should check if such updates change the execution flow (with respect of the issued queries). We expect that most of the updates will not contain substantial changes which impact the execution flow with respect of the queries issued; in this case there thus is no need of a new profile. Whenever the update changes the flow in a minor way, the profile can be manually fixed by an administrator. In case of major updates which heavily change the execution flow, a new profile must be created from scratch.

JCute can only solve numerical constraints. Whenever the application input is in form of strings, the solver cannot force the execution of different branches. To solve this problem a major extension of JCute is required to add a constraint solver for string values.

JCute can only analyze variables when the execution flow is inside the main code. However, whenever the execution moves to some external library, the solver loses control of what happens to the values and is not any longer able to generate inputs to solve the future constraints. For example, consider the code `if ( a > Math.max(b, c) ){...} else {...}`, where all the variables provided as input to the application are integer. The solver cannot generate values to force both the branches because JCute has no knowledge about what happens inside the `Math.max(int, int)` function. In the case of Java libraries this problem can be easily solved decompressing the jar file and letting JCute analyze it. But the standard Math library is implemented mostly using native code. A solution to this problem would be to provide an instrumented version completely written in Java to be used during the profile creation phase.

The Java Language Specification states some constraints that the code must fulfill [18], and some of them are related to the maximum size of classes and methods. JCute injects at runtime some code to follow the execution flow and correctly analyze the branches. Therefore, if a class is already close to the limits, with the injected code it may exceed these limits resulting in an invalid bytecode that cannot be executed and, consequently, analyzed. Luckily these limits are very high and very difficult to be reached, especially if the code is written following good coding style techniques.

10 Related Work

A formal framework to categorize anomaly detection systems has been proposed by Shu et al. [34]. According to this classification, our proposed approach uses a deterministic language defined on the top of the database interactions to perform the detection.

Several approaches have been proposed to protect databases against malicious application programs. DIDAFIT [17] is an intrusion detection system that works at the application level. Like our system, DIDAFIT works in two phases: training phase and detection phase. During the training phase, database logs are analyzed to generate fingerprints of the queries found in the log. Fingerprints are regular expressions of queries with constants in the `WHERE` clause replaced by place-holders that reflect the data types of the constants. During the detection phase, input queries are checked against such fingerprints. Queries that match some expression in the profiles are considered benign, and anomalous otherwise. DIDAFIT has however some major drawbacks. First, the system relies only on logs to create program profiles. There is therefore no guarantee that the log would contain all legitimate queries. To address this drawback, the authors propose a technique to generate new signatures from other signatures that are similar in all portions and have some predicates in common. While this solution works in some cases, the system would not be able to recognize queries that do not appear in the log. Another problem is that DIDAFIT does not take into account the control flow and data flow of the program, i.e., the algorithm neither checks the correct order of the queries, nor the constraints that have to be verified for a query to be executed. The approaches proposed by Bertino et al. [5] and Valeur et al. [36] also analyze training logs for creating profiles of queries. Therefore they have the same drawbacks mentioned earlier. These approaches focus on the detection of web-based attacks, like SQL Injection and Cross-Site Scripting (XSS) attacks, and fail to detect other attacks performed through application programs, e.g., code modification attacks.

Securing a database can be a difficult task, Paleari et al. [21] described a new category of attacks which rely on race conditions. Such kind of attacks are easier in web applications, where the tools used (mostly PHP and MySQL) offer a poor set of synchronization primitives but provide a highly parallel environment. Therefore, when multiple simultaneously requests are executed, it is possible to interleave the SQL queries in a way that generates unexpected behavior. Such a kind of attack may be mitigated by an approach, like the one we propose in this paper, which can enforce the correct order of the queries.

Our previous poster paper [27] outlines some preliminary ideas to protect against data exfiltration through malicious modification of the application program. However, the approach proposed in this paper reduces the performance overhead by allowing the *ADE* to simply traverse the *application profile* instead of concretizing of the symbolic execution tree of the application program. Such concretization in the detection engine results in extra delay when verifying a query. In addition, our preliminary approach does not cover the combination of testing-based techniques with program analysis techniques nor cover implementation and assessment of the proposed approach.

The current paper is an extended version of conference paper [15]. Compared with this previous paper, the current paper has the following novel contributions. We have created a stronger architecture which can easily support different target databases. We have adopted the approach of instrumenting the environment of the application instead of the application itself, with the benefits described in Section 7.4. We extended the profile signature to represent sub-queries. We proposed a simpler version of the anomaly detection which does not require receiving the application input and thus does not require instrumenting the environment nor the application. Such simple version can be used in environments where deploying a new instrumented application is difficult or impossible, and we argued that it still gives a reasonable level of safety against different kinds of attacks (see Section 6.3). We introduced the important notion of confidence for the profiles which let us decide whether to issue alerts or warnings according to the confidence obtained during the *profile creation phase*. This last extension removes the difference between flexible and strict policy previously introduced to deal with incomplete profiles. Our previous approach was based on the idea that, after the profile creation, an administrator had to check the code coverage as reported by jCute and decide whether to use the strict or the flexible policy. If the flexible policy were chosen, the administrator had to also choose a number of "safe" anomalies with the idea that, if an anomalous query had been issued a number of times greater than a given threshold, a stronger alert had to be raised asking the administrator to revise the profile. But, considering that in high security environments, even one anomaly can be a problem, the flexible policy would not adequate. By contrast, the current approach based on the confidence degree, allows us to explicitly mark the portion of profiles from where we expect unseen code to be executed, thus clearly differentiating anomalies and warnings also in very poor coverage profiles and without requiring any administrator decision. We performed a new set of experiments to evaluate the network usage overhead. Finally we have discussed in details the limitations we have identified in the use of jCute and outlined approaches to address such limitations.

Programs profiling techniques have also been proposed for many other purposes, such as debugging and collecting

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2598336, IEEE Transactions on Software Engineering

usage statistics [25], monitoring system calls [13] [39] [12], and enhancing the performance of database applications. For example, the Pyxis system [7] uses static analysis of application code to partition the code into two pieces: one to be executed on the application server and the other on the database server, trying to reduce the control transfers and amount of exchanged data between the two components.

Dasgupta et al. [10] propose a static analysis to database applications that use ADO.net APIs in order to extract features of SQL queries, query parameters, and usage of query results in order to detect SQL injection attacks and potential data integrity violations. Ramachandra and Sudarshan have developed DBridge [24], a tool that optimizes the performance of database applications by prefetching query results. Control-flow and data-flow analysis are used to find locations in the program where instrumented code can be added; at program runtime this code sends requests to the database to prepare results of queries predicted to be sent by the program at later points.

Many other approaches have been proposed to detect abnormal execution behavior. Xu et al. [40] propose a tool which can detect an abnormal control flow with respect to system and library calls. They use static analysis combined together with a probabilistic model to evaluate the likelihood that a sequence of calls has been issued by a compromised program. Shu at al. [33] argue that a new category of control flow attacks exists, namely aberrant path attacks, that are difficult to detect because they do not directly change the flow of the execution, but change some data which is used by the program itself to decide the execution flow. Such kind of attacks can generate montage anomalies, when we can observe multiple legitimate control flows that are incompatible in a single execution, or frequency anomalies, that is, a legitimate code block that is called too frequently. They argue also that usually such attacks happen in a large-scale execution window, being unseen by classical detection techniques that, for performance reason, can analyze only a small portion of the execution window. They propose a probabilistic method that does not suffer by combinatorial explosion and can scale to analyze the flow on larger execution windows. It is important to notice that, in proposing our anomaly detection technique, we considered a totally different scenario. Approaches, like the one by Shu et al. [33] aim at protecting the user against exploited or compromised applications. In our scenario we aim at protecting the database also against users who may intentionally alter applications and/or disable locally installed security tools in order to steal or alter data stored in the DBMS.

Finally, we would like to point out that security must be approached by combining different techniques, each protecting against specific types of attack, and that a comprehensive intrusion detection system should aggregate multiple warnings from different sources. In this respect, our anomaly detection system would be one of such warning sources. Vigna et al. [38], for example, have shown that it is possible to significantly increase the detection accuracy by combining a web-based and a database anomaly detection system. The idea of our approach is to provide another warning source and anomaly detection tool, that can be used together with existing tools to increase the overall protection.

11 Conclusion and future work

Though access control mechanisms deployed in DBMS are able to prevent application programs from accessing the data for which they are not authorized, they are unable to prevent data misuse caused by authorized application programs. In this paper, we have proposed an anomaly detection mechanism that is able to identify anomalous queries resulting from previously authorized applications. Our mechanism builds close to accurate profile of the application program, without the need of its source code, and checks at run-time incoming queries against that profile.

In addition to anomaly detection, our *DetAnom* mechanism is capable of detecting any injections or modifications to the SQL queries. We want to emphasize two benefits of our approach compared to other more conventional techniques. The first is that by using the concolic testing technique instead of static analysis techniques, we can profile the actual execution of the code which includes queries executed by self-modifying or dynamically downloaded code. The second is that we are able to enforce the actual order of the queries sent to the database, unlike conventional SQL injection detection approaches which are unable to determine whether a query is added or removed from an application program.

We have implemented *DetAnom* with JCute and PostgreSQL which results in low run-time overhead and high accuracy in detecting anomalous database accesses.

We are currently extending our work along several directions. Our current implementation of *DetAnom* exploits the constraints that JCute [30] supports, i.e., arithmetic, pointer, and thread constraints. We plan to improve our signature generation scheme by incorporating information about program constants, variables, logical and relational operators used in the `WHERE` clause of a query as this information may enhance the accuracy of detection. We also plan to enhance the completeness and accuracy of our profile creation mechanism using both static and dynamic analysis of the program. In this approach, we will first analyze the program statically to find all the execution paths that contain SQL queries and then guide the concolic execution dynamically so that it does not leave any paths unexplored.

Acknowledgments

The work reported in this paper has been funded in part under subcontract to Northrop Grumman Systems Corporation in support of a contract with Department of Homeland Security (DHS) Science and Technology Directorate, Homeland Security Advanced Research Projects Agency, Cyber Security Division. The views expressed in this work are those of the authors and do not necessarily reflect the official policy or position of the Department of Homeland Security or of Northrop Grumman Systems Corporation.

References

[1] Cybersecurity watch survey: How bad is the insider threat? Technical report, Carnegie Mellon University, 2012. http://resources.sei.cmu.edu/asset_files/Presentation/2013_017_101_57766.pdf.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2598336, IEEE Transactions on Software Engineering

[2] A. Balakrishnan and C. Schulze. Code obfuscation literature survey. *CS701 Construction of compilers*, 19, 2005.

[3] E. Bertino. *Data Protection from Insider Threats*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, San Rafael, 2012.

[4] E. Bertino and G. Ghinita. Towards mechanisms for detection and prevention of data exfiltration by insiders: Keynote talk paper. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 10–19, New York, NY, USA, 2011. ACM.

[5] E. Bertino, A. Kamra, and J. P. Early. Profiling database application to detect sql injection attacks. In *IEEE International Performance, Computing, and Communications Conference, IPCCC 2007*, pages 449–458, April 2007.

[6] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[7] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *VLDB Endow.*, 5(11):1471–1482, July 2012.

[8] M. Collins, D. M. Cappelli, T. Caron, R. F. Trzeciak, and A. P. Moore. Spotlight on: Programmers as malicious insiders (updated and revised). Technical report, Carnegie Mellon University, 2013. http://resources.sei.cmu.edu/asset_files/WhitePaper/2013_019_001_85232.pdf.

[9] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119–129 vol.2, 2000.

[10] A. Dasgupta, V. Narasayya, and M. Syamala. A static analysis framework for database applications. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1403–1414, Washington, DC, USA, 2009. IEEE Computer Society.

[11] M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 151–162, New York, NY, USA, 2007. ACM.

[12] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 318–329, New York, NY, USA, 2004. ACM.

[13] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium NDSS*, 2004.

[14] W. G. Halfond, J. Viegas, and A. Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[15] S. R. Hussain, A. M. Sallam, and E. Bertino. Detanom: Detecting anomalous database transactions by insiders. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 25–35. ACM, 2015.

[16] C. Huth and R. Ruefle. Components and considerations in building an insider threat program. Technical report, Carnegie Mellon University, 2013. http://resources.sei.cmu.edu/asset_files/Webinar/2013_018_101_69083.pdf.

[17] S. Y. Lee, W. L. Low, and P. Y. Wong. Learning fingerprints for a database intrusion detection system. In *Proceedings of the 7th European Symposium on Research in Computer Security*, ESORICS '02, pages 264–280, London, UK, UK, 2002. Springer-Verlag.

[18] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[19] R. Majumdar and K. Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering, ICSE 2007*, pages 416–426, May 2007.

[20] J. Melton and A. R. Simon. *SQL: 1999: understanding relational language components*. Morgan Kaufmann, 2001.

[21] R. Paleari, D. Marrone, D. Bruschi, and M. Monga. On race vulnerabilities in web applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 126–142. Springer, 2008.

[22] PostgreSQL Global Development Group. *PostgreSQL-9.1.8*. http://www.postgresql.org/docs/9.1/static/release-9-1-8.html.

[23] PostgreSQL Global Development Group. *Trusted Platform Module*. http://www.trustedcomputinggroup.org/developers/trusted_platform_module.

[24] K. Ramachandra and S. Sudarshan. Holistic optimization by prefetching query results. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 133–144, New York, NY, USA, 2012. ACM.

[25] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC '97/FSE-5, pages 432–449, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[26] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. volume 15, pages 2:1–2:34, New York, NY, USA, Mar. 2012. ACM.

[27] A. Sallam and E. Bertino. Poster: Protecting against data exfiltration insider attacks through application programs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1493–1495, New York, NY, USA, 2014. ACM.

[28] A. Sallam, E. Bertino, S. R. Hussain, D. Landers, R. M. Lefler, and D. Steiner. Dbsafe – an anomaly detection system to protect databases from exfiltration attempts. *IEEE Systems Journal*, PP(99):1–11, 2015.

[29] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[30] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Berlin, Heidelberg, 2006. Springer-Verlag.

[31] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[32] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272–282, May 2004.

[33] X. Shu, D. Yao, and N. Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 401–413. ACM, 2015.

[34] X. Shu, D. D. Yao, and B. G. Ryder. A formal framework for program anomaly detection. In *Research in Attacks, Intrusions, and Defenses*, pages 270–292. Springer, 2015.

[35] R. Srinivasan, P. Dasgupta, T. Gohad, and A. Bhattacharya. Determining the integrity of application binaries on unsecure legacy machines using software based remote attestation. In *Proceedings of the 6th International Conference on Information Systems Security*, ICISS'10, pages 66–80, Berlin, Heidelberg, 2010. Springer-Verlag.

[36] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of sql attacks. In *Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'05, pages 123–140, Berlin, Heidelberg, 2005. Springer-Verlag.

[37] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[38] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel, and E. Kirda. Reducing errors in the anomaly-based detection of web-based attacks through the combined analysis of web requests and sql queries. *J. Comput. Secur.*, 17(3):305–329, Aug. 2009.

[39] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy, S&P 2001*, pages 156–168, 2001.

[40] K. Xu, D. Yao, B. Ryder, and K. Tian. Probabilistic program modeling for high-precision anomaly classification. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*, pages 497–511, July 2015.

Lorenzo Bossi is a postdoc at the department of computer science at Purdue University. His research interests are on data protection from insider threat. He got his PhD in computer science from the Insubria University in Italy.

Elisa Bertino is professor of computer science at Purdue University, and serves as Director of Purdue Cyber Center and Research Director of the Center for Information and Research in Information Assurance and Security (CERIAS). She is also an adjunct professor of Computer Science & Info tech at RMIT. Prior to joining Purdue in 2004, she was a professor and department head at the Department of Computer Science and Communication of the University of Milan. She has been a visiting researcher at the IBM Research Laboratory (now Almaden) in San Jose, at the Microelectronics and Computer Technology Corporation, at Rutgers University, at Telcordia Technologies. Her recent research focuses on data security and privacy, digital identity management, policy systems, and security for the Internet-of-Things. She is a Fellow of ACM and of IEEE. She received the IEEE Computer Society 2002 Technical Achievement Award, the IEEE Computer Society 2005 Kanai Award, and the ACM SIGSAC 2014 Outstanding Contributions Award. She is currently serving as EiC of IEEE Transactions on Dependable and Secure Computing.

Syed Rafiul Hussain is a PhD student at the department of computer science at Purdue University. His research interests are on data protection from insider threat and provenance techniques for sensor networks.