DetAnom: Detecting Anomalous Database Transactions by Insiders

Syed Rafiul Hussain
Dept. of Computer Science
Purdue University, USA
hussain1@purdue.edu

Asmaa Sallam
Dept. of Computer Science
Purdue University, USA
asallam@purdue.edu

Elisa Bertino
Dept. of Computer Science
and Cyber Center
Purdue University, USA
bertino@purdue.edu

ABSTRACT

Database Management Systems (DBMSs) provide access control mechanisms that allow database administrators (DBA) to grant application programs access privileges to databases. However, securing the database alone is not enough, as attackers aiming at stealing data can take advantage of vulnerabilities in the privileged applications and make applications to issue malicious database queries. Therefore, even though the access control mechanism can prevent application programs from accessing the data to which the programs are not authorized, 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 design and implement an anomaly detection mechanism, *DetAnom*, that creates a profile of the application program which 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, the profile keeps a signature and also the corresponding constraints that the application program must satisfy to submit that query. Later in the detection phase, whenever the application issues a query, the corresponding signature and constraints are checked 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 we need neither any previous knowledge of application vulnerabilities nor any example of possible attacks to build the application profiles. As a result, our *DetAnom* 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, and requires acceptable amount of time, and that the detection mechanism incurs low run-time overhead.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
CODASPY'15, March 2–4, 2015, San Antonio, Texas, USA.
Copyright © 2015 ACM 978-1-4503-3191-3/15/03 ...$15.00.
http://dx.doi.org/10.1145/2699026.2699111 .

Categories and Subject Descriptors

H.2.7 [**Database Management**]

General Terms

Security

Keywords

Database, Insider Attacks, Anomaly Detection, Application Profile, SQL Injection

1. INTRODUCTION

Data stored in databases is often critical to the organization's operations and sensitive, for example with respect to privacy. Therefore, securing data stored in a database is a critical security requirement. Data must be protected not only from external attackers, but also from users within the organizations [2]. 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, 6, 11].

There are a number of facts that make the prevention of insider attacks more challenging compared with other conventional (external) attacks [3]. First, insiders are allowed to access machines 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 subjected 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 is mediated by the application programs. However, whereas the use of DBMS-level and application-level access control mechanisms provides 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 control.

In order to address the above problem, one possible approach is to analyze the data access patterns of the application to create profiles on legitimate activities and then use at run-time these profiles to detect anomalous 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 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 interactions 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 adopt concolic testing [19] [8], a widely used software testing methodology which ensures good coverage of the created profile. 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. Later in the *anomaly detection phase*, whenever the application issues a query, the corresponding signature and constraints are checked against the current context of the application. If there is a mismatch, the query is considered as anomalous. However, depending on the number of paths covered in concolic execution, the *anomaly detection phase* follows either the *'strict'* or the *'flexible'* policy. If the number of execution paths covered in the application profile is high, the *anomaly detection phase* verifies a query with more confidence and thus enforces the *'strict'* policy. On the other hand, if the profile fails to cover a minimum number of execution paths, the *anomaly detection phase* is comparatively less confident and thus enforces the *'flexible'* policy. The *'strict'* policy raises an alert immediately upon detecting an anomalous query, whereas the *'flexible'* policy raises a flag for that query and observes any further query. The main advantage of our *anomaly detection* mechanism is that we need neither any previous knowledge of application vulnerabilities nor any example of possible attacks to build the application profiles.

The rest of the paper is organized as follows: Section 2 presents relevant preliminary concepts. Section 3 provides an overview of our system model. Section 4 and 5 describe the *profile creation phase* and the *anomaly detection phase*, respectively. Section 6 discusses implementation details. Section 7 presents an experimental evaluation of *DetAnom*. Section 8 surveys related work. Section 9 concludes the paper with a discussion on the future work.

2. PRELIMINARIES

In this section, we present some concepts that are used in this paper.

Software Testing is the process of examining the quality of a software product. It involves monitoring the actual program execution in the hope of observing unexpected behaviour (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 that arises 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 in addressing this challenge.

Concolic Execution is a program analysis technique [8, 13, 19] that explores 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. SYSTEM MODEL

In this section, we introduce the *DetAnom* architecture and the adversary model that we consider.

3.1 DetAnom Architecture

The system architecture consists of several modules, supporting the two phases of *DetAnom*.

Profile creation phase: Figure 1 shows the modules supporting the *profile creation phase* and interactions between them. This phase starts with providing the application program as input to the *concolic execution (CE)* module which first instruments the application. Next, the *CE* module executes the instrumented application for a number of times and with different inputs, until all possible execution paths of the program are explored. Each time the application pro-

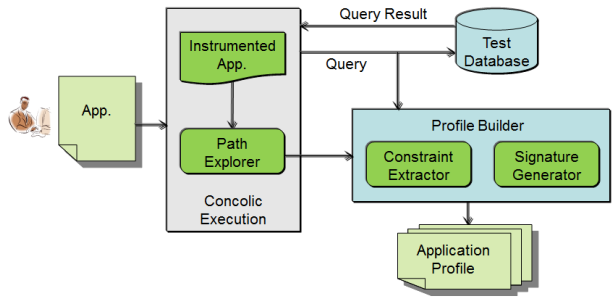

Figure 1: System architecture for profile creation

gram issues a query to the database, the *constraint extractor (CEx)* in the *profile builder (PB)* module extracts the constraints that lead the application program to follow the current path. These constraints compose a part of the *application profile*. On the other hand, each query submitted to the database is also forwarded to the *PB* module where the *signature generator (SG)* sub-module generates the signature of that query. Section 4 discusses details about the *CEx* and *SG* sub-modules. Finally, the *PB* module binds the query signature with its corresponding constraints and inserts this record into the *application profile*.

The database used in this phase is a *test database* that may be updated according to the requirement of concolic execution. This *test database* is necessary in the *profile creation phase* since the results returned by the database could be used in control-flow decisions later in the program.

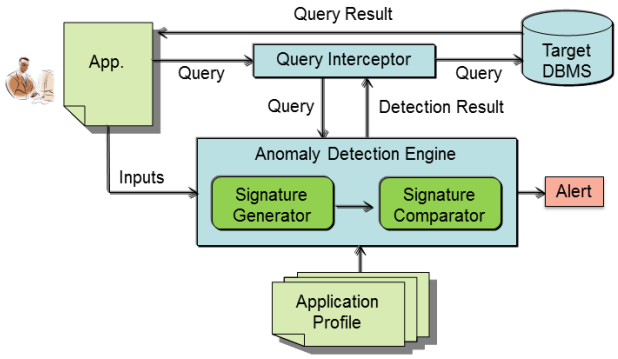

Figure 2: System architecture for anomaly detection

Anomaly detection phase: The main modules supporting the *anomaly detection phase* are: the *anomaly detection engine (ADE)*, the *query interceptor (QI)*, the *signature comparator (SC)*, and the *target database* as shown in Figure 2. The *target database* stores the data to be protected from insiders. The application interacts with this database through SQL queries. However, in this phase, any query issued by the application does not reach the *target database* directly; it is instead intercepted by the *QI* and forwarded to the *ADE* for anomaly detection. The *ADE* also includes the *SG* sub-module in order to generate the signature of the received query. Upon receiving the query, the *ADE* checks whether the current inputs of the program satisfy the constraints of some possible execution paths. If the constraints are satisfied, the *SC* compares the signature associated with the satisfied constraint to that of the received query . If there is a match, the query is considered legitimate. This information is then sent to the *QI* to let it forward the legitimate query to the *target database* for execution. On the other hand, if the signatures do not match, the *ADE* module considers the query as anomalous and raises an alert.

3.2 Adversary Model

In our *DetAnom* system, we assume that every component involved in the *profile creation phase* is trusted. However, we assume that at run-time the application program can be tampered with and thus become untrusted. Therefore, we assume that while the program is executing, the program may issue a query that:

(a) has never 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.

4. PROFILE CREATION PHASE

In the *profile creation phase*, the application program interacts with the *test 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 (QR)*. All *QR*s 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*. In this section, we discuss the format of the query signatures and constraints, and the procedure for building the *application profile*.

4.1 Query Signature Representation

In our system, we consider standard SQL commands of types `SELECT, INSERT, UPDATE,` and `DELETE`. For instance, the format of a `SELECT` command is:

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

Our system internally represents an SQL query as a signature of the form *(c, t, r, q, n)*. Here, *c* represents the type of the SQL command which takes one of the values: '1', '2', '3', and '4' 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; this information is extracted from the `TARGET-LIST` of the query. Attributes are identified by two values: the ID of the table that the attribute belongs to and the ID of the attribute relative to that table. 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 `QUALIFICATION` which corresponds to the `WHERE` clause of the query. And the last field, *n*, in the signature denotes the number

Figure 3: An example of database application

```
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 }
```

of predicates in the `WHERE` clause. Note that all attributes of the signature are extracted by parsing the query.

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 as follows: `{1, {{200, 1}, {200, 2}}, {200}, {{200, 2}}, 1}`

We explain this signature construction in order from left to right. The leftmost 1 represents the `SELECT` command. {200, 1}, and {200, 2} represent the IDs of attributes `employee_id` and `work_experience`, respectively. 200 represents the ID of the table `WorkInfo`. {200, 2} 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	Type
100	PersonalInfo	1	employee_id	varchar(10)
		2	employee_name	varchar(50)
200	WorkInfo	1	employee_id	varchar(10)
		2	work_experience	number
		3	salary	number
		4	performance	varchar(20)

4.2 Constraint Extraction

This section describes how the constraints for executing a query are extracted during the *profile creation phase*. The *CE* module takes the application program 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 random 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. This concolic execution is repeated for a number of times until all the execution paths are explored. Note that the instrumented program may issue queries along some of these execution paths. The issued queries are forwarded to both the *PB* and the *test database*. Upon receiving a query, the *CEx* sub-module in the *PB* extracts the constraints that are prerequisite to execute that query.

We now explain the constraint extraction procedure by using as example the application program shown in Fig. 3. The `salaryAdjustment` program takes two inputs: `profit`, and `investment`. Depending on the values of inputs and results returned from the database, this application program issues different sets of queries. At first, assume that the *CE* module sets the values of `profit`, and `investment` to 60000 and 100000, respectively. When the program execution reaches the `if` statement at line 5, it encounters a branch condition that consists of these input variables. Assume that the *CE* module denotes the `profit`, and `investment` variables as x_1 and x_2, respectively. It then represents the constraint of the `if` branch as c_1, using x_1 and x_2, as shown in Table 2. As the initial inputs satisfy this constraint, the program enters into the `if` branch. It then issues $query_1$ at line 7 along its current execution path. Here, the constraint for $query_1$ is the same as the condition for the `if` branch, i.e, c_1. Upon receiving this query, the *CEx* sub-module extracts this constraint from the *CE* module and stores for use in profile creation.

4.3 Profile Creation

In this section, we show how the application profile is created using *QR*s, which are composed of query signatures and corresponding constraints. The definition of application profile is as follows:

Application Profile (AP): The profile of an application program P is a directed graph $T(V_P, E_P)$. Each node $v_i \in V_P$ is a *QR* 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 4.1 and 4.2.

As $query_1$ is issued by the program, it is then passed to the *PB* where the *SG* sub-module generates the query signature as shown in Section 4.1. The *CEx* sub-module also extracts the constraint c_1 as described in Section 4.2. The *PB* module now generates the record of $query_1$ as $QR_1 = \langle sig(query_1), c_1 \rangle$ and inserts this record as the first child of the root of *AP* as shown in Fig. 4(a).

Afterwards, when the program reaches the `if` statement at line 9, the *CE* module identifies a branch condition that depends on the results returned by $query_1$. This is represented as a database constraint, c_2, as in Table 2. Now assume that the *test database* returns less than 100 rows of data to the application program for $query_1$. In this case, the program jumps to the `else` branch at line 13 and issues $query_2$. So,

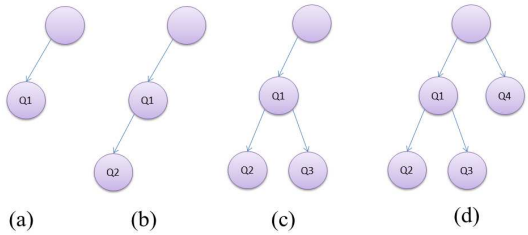

Figure 4: Steps of profile graph construction

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:` $1.0\ x_1 - 0.5\ x_2 <= -1.0$

the precondition for $query_2$ to be executed is same as the condition of the `else` branch at line 13. This constraint, c_2, is extracted by the *CEx* as shown in Table 2, where x_3 denotes for `resultSet1.getRow()`. The *PB* module then creates the record of $query_2$ as $QR_2 = \langle sig(query_2), c_2 \rangle$ and inserts it in the *AP* as a child of QR_1 as shown in Fig. 4(b). Table 4 presents the signature of $query_2$, i.e., $sig(query_2)$.

Since for the current execution there is no further paths to explore after line 15, the *CE* module backtracks the execution to the `if` statement at line 9 and negates the branch condition to explore the `if` branch. To do so, the *CE* module inserts some random records into the database tables `PersonalInfo` and `WorkInfo` so that execution of $query_1$ returns more than 100 rows, and thus explores the `if` branch. Along this path when the program issues $query_3$, the *CEx* sub-module captures the corresponding constraint which is same as the condition of the `if` branch at line 9. This constraint is represented as c_3 (shown in Table 2). The *AP* is also updated as in Fig. 4(c) by inserting the query record $QR_3 = \langle sig(query_3), c_3 \rangle$ as another child of QR_1. Note that, insertion or deletion of records to the *test database* in the *profile creation phase* do not have any impact on the *target database* as these operations are executed only for extracting the database constraints properly. Finally, the *CE* module backtracks the execution of the program to the `if` statement at line 5, negates the branch condition, and uses a constraint solver to find values of `profit` and `investment` variables so that the program execution can explore the other branch, i.e., the `else` branch at line 16. Assume that it sets `profit` and `investment` variables to 49999 and 100000, respectively and enters the `else` branch at line 16. Executing along this path when the program issues $query_4$, the *CEx* extracts of the constraint c_4, as shown in Table 2. The *PB* inserts $QR_4 = \langle sig(query_4), c_4 \rangle$ to the *AP* as a child of the root as shown in Fig. 4(d). At this point, since the *CE* module has completed exploring all execution paths, the *profile creation phase* ends.

5. 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.

Table 3: Query signatures

Query	Signature
$query_1$	{1, {{200, 1}, {200, 2}}, {200}, {{200, 2}}, 1}
$query_2$	{2, {{200, 3}}, {200}, {∅}, 0}
$query_3$	{1, {{200, 1}}, {200}, {{200, 2}, {200, 4}}, 2}
$query_4$	{1, {{100, 2}}, {100, 200}, {{200, 4}, {100, 1}, {200, 1}}, 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	$\langle sig(query_4), c_4 \rangle$

5.1 Detection of Anomalous Queries

In the *anomaly detection phase*, whenever the application program issues a query, the *QI* module intercepts the query and forwards it to the *ADE* module. The reason is that our system does not allow any query to reach the *target database* without verifying whether the query is anomalous or not.

When an application program starts executing in the *anomaly detection phase*, the *ADE* module sets the root node of the *AP* 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 *QR*. 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 *SG* sub-module generates the signature of the received query and the *SC* 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 *QI* 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 *QR* 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.

Now consider the case of an anomalous query. In this case, the signature for that query generated by the *SG* sub-module will not match that in the *QR* for which the constraints are satisfied by the application program. As a result, the *SC* sub-module raises a flag and the *ADE* takes next steps based on either the *'strict'* or the *'flexible'* policy discussed in what follows.

5.1.1 Strict policy

In the *profile creation phase*, we use *concolic execution* to explore all possible paths of an application program. This technique statically estimates the number of possible branches in a program. Then while executing the program, it sets different values to the inputs and thus tries to explore new paths. However, it uses a bounded depth-first strategy, i.e., bounded DFS. With this searching strategy, there is a trade-off between the exploration of other execution paths and termination of the current path if its length is significantly large. If the length of an execution path exceeds the bound of the DFS, it stops that particular execution, and searches for new paths. In this case, the *concolic execution* leaves

Algorithm 1 Anomaly Detection

```
1: Input: Application Profile (AP)
2: v_p = root of AP
3: while the program is executing do
4:   if a query q is issued then
5:     SG generates sig(q)
6:     for each child v_i of v_p do
7:       if c_i is satisfied then
8:         SC compares sig(q) to sig(query_i)
9:         if signatures match then
10:          /* q is a legitimate query */
11:          let the QI forward q to database
12:          v_p = v_i
13:        else
14:          /* q is an anomalous query */
15:          anomaly = 1
16:          break
17:        end if
18:      end if
19:    end for
20:    if anomaly == 1 || no c_i is satisfied then
21:      if policy is strict then
22:        raise an ALERT
23:      else if policy is flexible then
24:        if q is flagged more than k times then
25:          raise an ALERT
26:        else
27:          raise a flag
28:        end if
29:      end if
30:    end if
31:  end if
32: end while
```

some large execution paths unexplored that may contain queries. In the *strict* policy, we set the bound of the DFS high enough so that the *concolic execution* can explore almost all possible paths of the program and cover all the branches that are estimated statically. As a result, the profile of the application program gets close to be complete and the *ADE* module becomes strong enough to distinguish between legitimate and anomalous queries. So in this case, when the signature of an input query does not match, the *ADE* module identifies that query as anomalous with high confidence and raises an *alert* signal. This information is forwarded to the *QI* module.

5.1.2 Flexible policy

If the bound of the DFS for the *concolic execution* is not high enough, the *profile creation phase* may leave some large paths unexplored. For each query issued along an execution path that is within the DFS bound, if the *SC* does not find a match for its signature or the constraints are not satisfied, the *ADE* considers that query as anomalous. However, if the issued query is on an execution path that exceeds the DFS bound, the *SC* does not find a match for its signature. In this case, the *ADE* raises a flag for that query and asks the *QI* to drop it. If a particular query is flagged for more than k times (k is a threshold set in the *ADE* module), this module raises an *alert* signal, and requests the security officer (or some other trusted user) to check whether the query is actually anomalous or legitimate. If the query is assessed as anomalous, it is kept in the black-list of the *QI* so that future occurrences of such query are blocked automatically. If the query is assessed as legitimate, the *AP* is updated accordingly with its *QR*.

Also, consider the case in which the application program is compromised and its control-flow is hijacked. In this case, if some insider issues a query that belongs to the program but is not legal in the current execution path, the application inputs will fail to satisfy the constraints of any *candidate node*. In this case, the *ADE* will take an action according to the policy it adopts, as described above.

5.2 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.

5.2.1 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 *SG* sub-module generates the signature of the input query and the *SC* 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.

5.2.2 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 *SC* 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.

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

If a query is issued in the *anomaly detection phase* that is never encountered in the *profile creation phase*, the signature of that query does not match with any of the query records. In this case, the *ADE* module raises an *alert* or a *flag* based on the policy (*strict* or *flexible*) it adopts.

5.2.4 SQL injection attacks

Our approach is also able to detect SQL injection attacks. As these attacks typically modify the queries by adding new predicates, they can be easily detected by our anomaly detection mechanism because of its ability to profile the expected queries and compare them with the actual queries. 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:

```
1 username = readInputUser();
2 password = readInputPassword();
3 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 SQL injection attacks are based on re-structuring the SQL query, our mechanism by comparing the query structure to the query signatures saved in *AP* 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.

5.2.5 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.

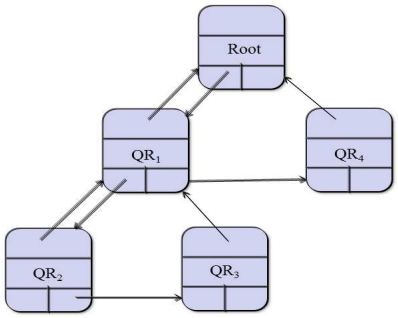

Figure 5: Profile graph

6. IMPLEMENTATION

In this section, we discuss the implementation details of the following components: *CEx*, *SG* and *AP*. In our implementation, we consider J2EE based application programs through which an attacker may illegally access the *target database*. However, our proposed *anomaly detection* mechanism can be used for other kinds of application programs. For simplicity of implementation, we use PostgreSQL-9.1.8 [14] for both *test database* and *target database*. These databases are logically separate but contains identical relation schema.

Constraint Extractor: Our implementation of *CEx* is built on top of the JCute concolic testing framework [18]. This framework uses Soot [21] for instrumenting Java class files and lpsolve for solving linear programs. We instrument the `executeQuery` and `executeUpdate` statements of the application program using Soot to insert the instructions for the invocation of the *CEx* module. So, whenever a query is encountered, the *CEx* first captures the constraints of the current path from root to the intercepted query. Since *CEx* 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 in the *AP* along with the signature of the intercepted query.

Signature Generator: We use PostgreSQL-9.1.8 [14] to implement the *SG* 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 *SG* imports necessary query information (command, target list, relation list, and qualifiers) from the parse tree and creates the query signature.

Application Profile: The *PB* module creates *query records* by combining the query signatures generated from *SG* and the constraints extracted by *CEx*. These records are stored in a hierarchical data structure at PostgreSQL as shown in Fig. 5.

7. EXPERIMENTAL EVALUATION

We have evaluated the performance of our proposed *DetAnom* mechanism for both the 'strict' and 'flexible' policies. Our experiments have been performed on an Intel(R) Core(TM) i7-3540M CPU machine (number of core = 2) with a CPU speed of 3.00 GHz, running Ubuntu-11.10 operating system with 4GB of memory. Since the complete-

Table 5: Program constructs

Program ID (PID)	Lines of code	Number of nested if-else	Number of for loops	Maximum depth of a path	Number of branches	Total # of queries in a program	Average # of queries in each if-else branch	Average # of queries in each loop
1	794	30	-	30	32	200	7	-
2	586	20	-	20	22	200	10	-
3	322	10	-	10	12	200	20	-
4	826	30	5	∞	32	150	4	7

ness and accuracy of our *application profile* depend mostly on the performance of concolic testing [18, 19], we have implemented four different database applications with different program constructs to evaluate the performance of *DetAnom*. A detailed overview of these applications is shown in Table 5.

7.1 Evaluation Metrics

We analyze the performance of our proposed anomaly detection mechanism using the following metrics:

(a) *Profile creation time*: It is the time required to create the profile of an application program. We use the JCute [18] tool to measure the time elapsed for the concolic execution of the application program.

(b) *Branch coverage*: It is defined as the ratio of the number of branches covered in run-time in *profile creation phase* to the total number of branches in the program. If there are total n branches in a program and c branches are covered among them while creating the profile, the branch coverage is computed as:

$$Branch\ Coverage\ =\ \frac{c}{n} * 100\%$$

(c) *Run-time overhead:* It is defined as the ratio of additional time required by *DetAnom* to the time required by the program to execute without any anomaly detection mechanism. We denote the execution time of a database application without *DetAnom* as t_1 and with *DetAnom* as t_2. We then compute the run-time overhead as follows:

$$Run\text{-}time\ Overhead\ =\ \frac{t_2 - t_1}{t_1} * 100\%$$

(d) *False positive and false negative:* A *false positive* is a case in which a legitimate query is evaluated as anomalous by the detection engine, whereas a *false negative* means that an anomalous query is evaluated as legitimate.

Table 6: Different scenarios

Scenario ID	Program ID	Depth limit of bounded DFS
1	1	30
2	1	20
3	1	10
4	2	20
5	3	10
6	4	30
7	4	20
8	4	10

7.2 Results

In order to analyze our *DetAnom* system more rigorously, we have set different values for the depth of the bounded DFS in concolic execution, which resulted in eight different scenarios shown in Table 6.

Table 7: Profile creation time for different scenarios

Scenario ID	Profile creation time (seconds)
1	324.441
2	130.727
3	55.448
4	137.885
5	67.9425
6	1441.940
7	1148.475
8	755.192

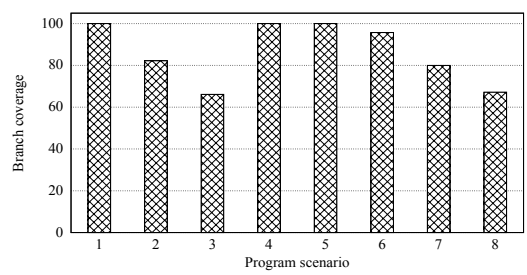

Figure 6: Branch coverage for different scenarios

(a) *Profile creation time*: The amount of time required for creating the application profile for the eight different scenarios is reported in Table 7. The results show that setting the depth of the bounded DFS to a higher value takes a longer time to profile the application. The time elapsed for scenarios 6-8 is comparatively high because these scenarios include loops in the application program.

(b) *Branch coverage*: The branch coverage of the profiles for the eight different scenarios is shown in Fig. 6. Note that increasing the depth of the bounded DFS also increases the completeness and accuracy of the profile. For the scenarios where the depth is set equal to the maximum depth of the program, the branches are covered 100%. Also for a single application program, decreasing the depth of bounded DFS results in a low branch coverage.

(c) *Run-time overhead*: Fig. 7 reports the run-time overhead for the eight different scenarios. The run-time overheads for these scenarios do not differ to a large extent. The reason is that the *anomaly detection phase* takes almost the

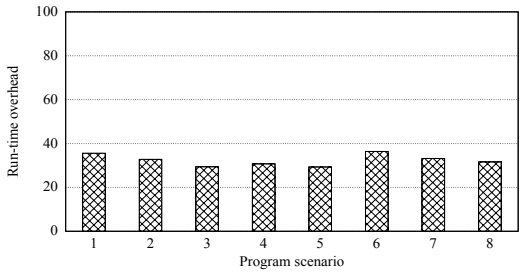

Figure 7: Run-time overhead for different scenarios

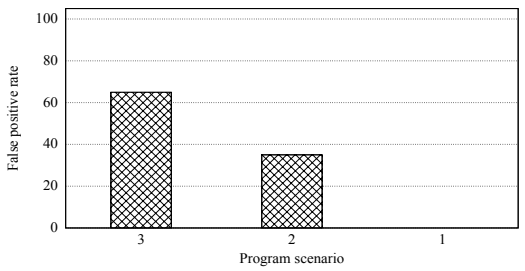

Figure 8: False positive rate for scenarios 1, 2, 3

same amount of the time in finding a match for each query issued.

(d) *False positive rate*: Fig. 8 shows the false positive rates for scenarios 1-3. In this experiment, false positive rates decrease with the increase of depth limits of bounded DFS set by the concolic execution in the *profile creation phase*. The higher the depth limit is set by the concolic execution, the more paths are covered by the application profile. Hence, scenario 1 covers all execution paths of the program and thus results in no false positives.

Fig. 9 also shows the false positive rates for scenarios 6-8. In this experiment, the termination condition of a `for` loop in the program is an input variable x, (e.g., `for (int i=0; i<x; i++)` {...}). One SQL query is issued from each iteration of this `for` loop. Hence, in the *profile creation phase* the `for` loop is iterated 30, 20, and 10 times in scenarios 6-8, respectively. So, in the *anomaly detection phase*, if the variable x is given the value 32, the number of false positives are 22, 12, and 2, respectively. Hence, it is evident that if the *profile creation phase* explores a large number of paths, the application profile becomes more complete and accurate.

Note that in our experiments we do not find any false negative as the *DetAnom* matches exact signature and constraints of a query.

8. RELATED WORK

Several approaches have been proposed to protect databases against malicious application programs. DIDAFIT [12] 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 ones 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 constants in the predicates. The approaches proposed by Bertino et al. [4] and Valeur et al. [20] 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.

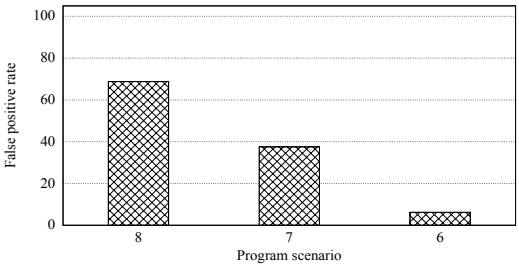

Figure 9: False positive rate for scenarios 6, 7, 8

Our previous poster paper [17] 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 *AP* 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. Also our current paper introduces the important notion of confidence for the profiles. According to the confidence obtained in the *profile creation phase*, our approach adopts either the *'strict'* or the *'flexible'* policy.

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

age statistics [16], monitoring system calls [10] [22] [9], and enhancing the performance of database applications. For example, the Pyxis system [5] 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. [7] propose static analysis of 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 [15], 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.

9. 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 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, e.g., SQL injection attacks. 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 [18] supports, i.e., arithmetic, pointer and thread constraints. However, Emmi et al. [8] propose a concolic testing approach for database applications which considers the database constraints as discussed in Section 4.3. We are incorporating these database constraints to our current prototype which will enhance the accuracy and completeness of our *anomaly detection* mechanism. We plan to improve our signature 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.

10. 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 under contract number HSHQDC-13-C-B0012. 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.

11. 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`.

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

[3] 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.

[4] 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.

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

[6] 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`.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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`.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.