An Anomaly Detection System for the Protection of Relational Database Systems against Data Leakage by Application Programs

Daren Fadolalkarim
Computer Science Department
Purdue University
West Lafayette, IN, US
dfadolal@purdue.edu

Elisa Bertino
Computer Science Department
Purdue University
W Lafayette, IN, US
bertino@purdue.edu

Asmaa Sallam
Computer Science Department
Purdue University
West Lafayette, IN, US
asallam@purdue.edu

***Abstract*—Application programs are a possible source of attacks to databases as attackers might exploit vulnerabilities in a privileged database application. They can perform code injection or code-reuse attack in order to steal sensitive data. However, as such attacks very often result in changes in the program's behavior, program monitoring techniques represent an effective defense to detect on-going attacks. One such technique is monitoring the library/system calls that the application program issues while running. In this paper, we propose AD-PROM, an Anomaly Detection system that aims at protecting relational database systems against malicious/compromised applications PROgraMs aiming at stealing data. AD-PROM tracks calls executed by application programs on data extracted from a database. The system operates in two phases. The first phase statically and dynamically analyzes the behavior of the application in order to build profiles representing the application's normal behavior. AD-PROM analyzes the control and data flow of the application program (i.e., static analysis), and builds a *hidden Markov model* trained by the program traces (i.e., dynamic analysis). During the second phase, the program execution is monitored in order to detect anomalies that may represent data leakage attempts. We have implemented AD-PROM and carried experimental activities to assess its performance. The results showed that our system is highly accurate in detecting changes in the application programs' behaviors and has very low false positive rates.**

***Index Terms*—Database, insider attacks, anomaly detection, application profile, data leakage**

I. Introduction

Motivation. Securing databases (DBs) not only from malicious outsiders but also from insiders who have privileges to access them is critical. The data might be leaked purposely or accidentally through different channels (e.g., through the Internet via email or social media, through hardware via printers, etc.) [1]. According to recent reports [2], [3], [4], data leakage (DL) has been one of the top five attack vectors in 2017-2018, and one of the main causes of such leakage is inadvertent or malicious insiders. As reported by [3], insider thefts accounted for about 41% in North America, and almost 50% of data losses caused by insiders were reported in the Asia-Pacific region. Thus, it is critical that mechanisms for DL detection be deployed. One such mechanism is based on monitoring users' and applications' behavior with respect to data use to detect anomalies that can be indicative of possible DL.

Problem and scope. The problem of tracking activities performed on data can be viewed as a multilevel problem in terms of data state. Data can be in different states: at rest (at the server), in transit (over a network), or in use (at endpoints) [1]. Detecting DL attempts requires in-depth analysis for each data state. Different techniques can be deployed depending on the state of the data to be protected. For example, one approach for monitoring data at rest is to detect anomalous DB queries. This can be implemented by tracking submitted queries to the DB. Then the queries' arguments or their frequencies are compared to profiles that represent normal activities. Monitoring data in use requires an anomaly detection (AD) system on each endpoint. Such system should be able to audit confidential data transfer, e.g., via printers, emails, or portable media, and monitor data transferred by users directly or through application programs. Systems were proposed for tracking data in use [5], [6]. However, those systems are unable track a user's activities on the data after the user gains access to the data through application programs.

Relevant approaches. Several approaches have been proposed [5], [6], [7], [8], [9], [10], [11], [12] to detect anomalies in DB queries or application programs. Such approaches are effective in detecting anomalous queries that are issued by users or application programs [5], [8], detecting code changes in application programs [11], [12], and tracking users' actions on data stored in relational DB or in files by tracking file accesses [6], [7]. Although such systems have significant advantages, they suffer from at least one of the following limitations: (1) inability to monitor data in use [8], [9], [10]; (2) inability to link activities to data sources [11], [12]; (3) inability to track users' or applications' actions on data after gaining access to the data [8], [5], [9], [10]; (4) inability to track sequences of actions on data [5], [6], [7], [8], [9], [10]. This paper thus address the following problem: *After an application program retrieves some data from a DB, is it possible to detect anomalous activities by the application program on such data that could be a sign of DL attempts?*

The design of such anomaly detection (AD) system is **challenging** as it needs to fulfill the following requirements: (1) It should require the least possible modifications to the application program code; (2) It should not degrade the system performance by introducing high overhead; (3) It should have high accuracy in the detection of anomalies.

Proposed approach. We address the above challenges by designing and implementing AD-PROM, a system to monitor data in use with the goal of detecting DL attempts by application programs; the system considers the attempts that result in

2375-026X/20/$31.00 ©2020 IEEE
DOI 10.1109/ICDE48307.2020.00030

changes to the call sequences issued by the application. The targeted data (TD) to be monitored is data whose source is a relational DB. Our system works in two phases: a training phase that collects normal program traces and builds a profile of the program behavior, and a detection phase that uses this profile to monitor the use of the TD by the program at runtime.

AD-PROM relies on program analysis to create the profile of the TD usage by the application program. There are two main approaches for analyzing a program behavior. The first approach is to use a *static program analysis* technique to identify all possible paths in the program. The second approach is to use a learning method that relies on program traces (e.g., execution-graph model [13], automaton model [14], and hidden Markov model (HMM) [15]). The static program analysis method can recognize all statically feasible paths in the program. However, it cannot distinguish the likelihood of occurrence of each possible path. Unlike static analysis, a model built using a *learning-based (dynamic) approach* can provide an estimation of the probability of occurrences for new traces. Nonetheless, a learning-based model usually produces a lot of false positives if the traces used to train the model are not long enough. Unseen paths in the training data are considered anomalous even if they are feasible paths in the program [16].

AD-PROM uses both static and dynamic program analysis to build accurate profiles of the programs to be monitored. The purpose of the static analysis is to extract control and data flow of the program. AD-PROM uses the HMM to learn the dynamic behavior of the program. The reason for choosing the HMM is to enhance the accuracy of the system since it has shown high accuracy at detecting anomalous sequential activities, and predicting the likelihood of new ones [17]. However, a straightforward integration of those two methods would suffer from limitations inherited from both methods [11].

To address such issue, AD-PROM follows the approach by Xu et al. [12]. The approach integrates static and dynamic analysis by initializing the HMM using a new technique, that is, using probability forecast based on the program's data and control flow instead of using a random initialization as followed by most HMM-AD systems [17], [18]. Notice however that the system by Xu et al. [12] has a different goal, that is, detecting code-reuse attacks and is unable to connect program's activities on the TD to its source as it does not consider data flow analysis. Our system analyzes the data dependency between the statements that transfer the TD to the program (e.g., PQexec) and output statements (e.g., printf and fprintf). We then label such statements to distinguish them when we build the model. We also instrument the program while running to track the data flow dynamically to detect code changes that may lead to data leakage.

DL detection systems can be categorized into two main methods [1]: content-based that detect the presence of the sensitive data at relevant endpoints, and context-based methods that focus on the behavior of users or applications that might leak the data. Our system utilizes both methods by performing control and data flow analysis to track the actions performed on the TD by application programs at the endpoints. It also analyzes the likelihood of actions by the application programs on the TD by using a statistical method. Statistical methods usually provide high AD accuracy and one such technique is the HMM [19]. Different systems were proposed that use HMM as an AD system; most of them show high accuracy with very low false positive (FP) [19]; thus, we use this method in AD-PROM.

The main **novel features** of AD-PROM are:

- It can link activities performed by application programs on the TD *in use* to its source, that is, the DB from which the TD has been retrieved.
- It analyzes both content and context of the data to detect signs of potential DL attempts.
- It applies an HMM technique to achieve high detection accuracy.
- It is able to predict the likelihood of new activities that may lead to DL.

The paper is organized as follows. Section II introduces preliminary notions. Section III discusses the adversary model. Section IV introduces the proposed system's architecture and the system phases. Section V reports results of the experiments. Section VI surveys related work. Section VII discusses the limitations of the system and how they can be addressed. Section VIII outlines concluding remarks.

II. PRELIMINARIES

In this section, we introduce preliminary notions needed for the subsequent discussion in the paper.

Hidden Markov Model (HMM): HMMs are statistical tools used to represent probability distributions of sequences of observations [20]. The models assume that the observations are generated by an unobserved (i.e., hidden) process that satisfies the Markov property, that is, given the value of previous state(s) (S_{i-1} or $\{S_{i-1}, ..S_{i-n}\}$, where $n >= 1$), the present state (S_i) is independent of all states that occur prior to it. An HMM is characterized by $\lambda = (A, B, \pi)$, where A is a matrix of the transition probabilities between hidden states; B is a vector of the emission probabilities of the hidden states; π is a vector of initial state distributions; N is the number of hidden states; and M is the number of observation symbols [21].

After doing the necessary initialization to the model's parameters, an observed sequence is built (e.g., $O = \{O_1, ..., O_T\}$, where T is the length of the sequence). Given O and an HMM model $\lambda = (A, B, \pi)$, there are three basic problems for the HMM [21]:

- The *evaluation problem*: the problem of finding the probability that O was generated by the model ($Pr(O|\lambda)$).
- The *decoding problem*: the problem of identifying the most likely state sequence in λ that produced O.
- The *learning problem:* the problem of modifying the model parameters (A, B, π) to maximize the value of $Pr(O|\lambda)$.

Original code:

```
query = "SELECT * FROM items WHERE ID = 10";
result = PQexec(query);
rows = PQntuples(result);
for (r=0; r<rows; r++)
    printf  ("%s", PQgetvalue(result, r, 0));
```

Resulting library call sequence would be:

```
PQexec, PQntuples, PQgetvalue, printf
```

Modified code:

```
query = "SELECT * FROM items WHERE ID >= 10";
result = PQexec(query);
rows = PQntuples(result);
for (r=0; r<rows; r++)
    printf  ("%s", PQgetvalue(result, r, 0));
```

Resulting library call sequence would be:

```
PQexec, PQntuples, PQgetvalue, printf, PQgetvalue,
printf, PQgetvalue, printf, PQgetvalue, printf …
```

Fig. 1: The attacker modifies the application code to exfiltrate more data by increasing query selectivity

Principal Component Analysis (PCA): PCA is the most widely used dimension-reduction tool. It can reduce large amounts of data (e.g., variable set or large matrix) while preserving most of the information in the original data.

K-Means Clustering: It is an unsupervised learning algorithm used to solve clustering problems. It is a greedy algorithm that classifies a given data set into K clusters, where K is a number set in advance.

Software Instrumentation: It is a technique for adding extra code to an application program for monitoring its behavior. It is used in application profiling for testing, performance evaluation, and code optimization. The program can be instrumented while executing (i.e., dynamically) or after it has been linked (i.e., statically) without the need to re-compile, re-link, or re-execute the program to change its binaries [22].

III. Adversary Model

There are different methods that enable the attacker to change an application program: (1) having a direct access to the code, (2) having access to the application's binary, (3) exploiting program's vulnerabilities without having access to neither the source code nor the binary. In what follows, we discuss example attacks that can be perpetrated in each case.

Case 1: The attacker has access to the source code (i.e., a developer); he/she can change the source code of the program by adding new commands leaking data (e.g., printing or redirecting data to a file) or reusing existing ones to output query results. One of the following attacks could be performed:

- *Attack 1.1*: The attacker partially modifies a command inside a program to retrieve more data and then prints the results on the screen (see Figure 1). The attacker can then take a screenshot of the data or take a picture of the screen using his/her phone camera. It is hard to track these actions. However, we can at least trigger an alarm (or for example a surveillance camera) when the user is inspecting data on his/her screen if it is the case that usually he/she does not do so.

Snippet of a vulnerable program (no prepared statements):

```
scanf("%s",accNo);
char query[200];
char *ts = "SELECT * FROM clients where id='";
char *tr = "';";
int i;
strcpy(query, ts);
strcat(query, accNo);
strcat(query, tr);
if (mysql_query(conn, query))
        finish_with_error(conn);
MYSQL_RES *result = mysql_store_result(conn);
...
while ((row = mysql_fetch_row(result)))
{
   for(i = 0; i < num_fields; i++)
       printf("%s ", row[i]? row[i]:"NULL");
}
```

Normal input: When the user inputs a **normal client id value** (i.e., 105), the resulted query will be:

```
SELECT * FROM clients where id='105';
```

Result: One record will be retrieved from the DB and the resulted call sequence of such query will be:

```
scanf, strcpy, strcat, strcat, mysql_query,
mysql_store_result, mysql_fetch_row, printf
```

Malicious input: When the attacker performs an **SQL injection** by passing `(1' OR'1'='1)`, the resulted query will be:

```
SELECT * FROM clients where id='1' OR '1'='1';
```

Result: The condition in where clause is evaluated to be true; hence, all client records will be retrieved from the DB and the resulted call sequence of such query will be:

```
scanf, strcpy, strcat, strcat, mysql_query,
mysql_store_result, mysql_fetch_row, printf,
mysql_fetch_row, printf, mysql_fetch_row, printf,
mysql_fetch_row, printf ...
```

The calls `(mysql_fetch_row, printf)` will be called the number of the retrieved records.

Fig. 2: Tautology-based SQL injection attack that results in anomalous call sequences

- *Attack 1.2*: The attacker inserts a new command into the code to store data to a file.
- *Attack 1.3*: The attacker extracts data via an existing command that stores some random data into a file by replacing the data with a query result.

Case 2: The attacker has access only to the program binaries; he/she still can make changes to the program:

- *Attack 2.1*: If the binary is not defended against analysis, the attacker can inject code patches into the binaries using an instrumentation tool (e.g., Dyninst [23]).
- *Attack 2.2*: The attacker can follow a technique called Return-oriented programming (ROP). In ROP, the attacker subverts the program's control flow, and then reuses short snippets of existing code in the program image, called gadgets. Each gadget ends with a ret instruction which enables the attacker to chain the snippets together to create a payload. To find the gadgets, the attacker can perform an automated search of the targeted application's binary [24].

Case 3: The attacker does not have access to neither the source

code nor the binary; he/she still can exploit program's vulnerabilities that could result in changing the program behavior. Some common vulnerabilities are as follows.

- *Attack 3.1*: If the program is not using prepared statements to submit queries, it is vulnerable to SQL injection attacks. SQL injection attacks have many types and targets; one of these attacks, which our system can detect, is the one that aims at exfiltrating (harvesting) data using a tautology. In tautology-based SQL injection attack, the attacker injects malicious input into conditional statements so that they are always evaluated to be true [25]. The injected SQL command could retrieve more data than usual, and hence the program issues more print or write to a file commands. Such change in the behavior can be detected by AD-PROM. Figure 2 shows an example on how such attack can change the behavior of the program. In the example, we injected the tautology attack (1' OR'1'='1) [25].
- *Attack 3.2*: If the program does not use encrypted connections to the DB engine, it is vulnerable to man-in-the-middle (MITM) attacks. One of the MITM attacks that can be detected by our system is when the attacker updates the query before sending it to the server to retrieve more data. When the application program retrieves a higher rate of result records than the usual rate, the program behavior will change like in attack 3.1.
- *Attack 3.3*: If address space layout randomization (ASLR) is not utilized to randomize the code location, the attacker can perform a blind return-oriented programming (BROP) attack. In BROP, the attacker can read the stack to leak canaries[1] and return addresses. The attacker hence can search for enough gadgets to invoke write call and control its arguments. He/she then can dump enough of the binary to find enough gadgets to build a payload [27]. The payload could leak the TD and change the behavior of the application, and consequently the application would issue different call sequences.

A modified program may exhibit two types of anomalous behaviors: (1) Execute a new action that the program was not supposed to execute and thus was not in the original program. This behavior can be observed from the attacks 1.1, 2.1, 2.2, and 3.3. (2) Change the behavior of an action that is part of a possible path in the program. This behavior can be observed from the attacks 1.2, 1.3, 3.1, and 3.2.

IV. AD-PROM Design

In this section, we first define some terms we use when introducing our system. We then describe the AD-PROM components, and the operation phases, that is, profile (model) creation phase and detection phase.

A. Definitions

- Call Graph (CG): This graph represents the relationship between the functions in the program.

[1]A canary is a small value placed on the stack before return instructions. The value is checked before the function returns to detect if the return address is tampered with because of a buffer overflow [26].

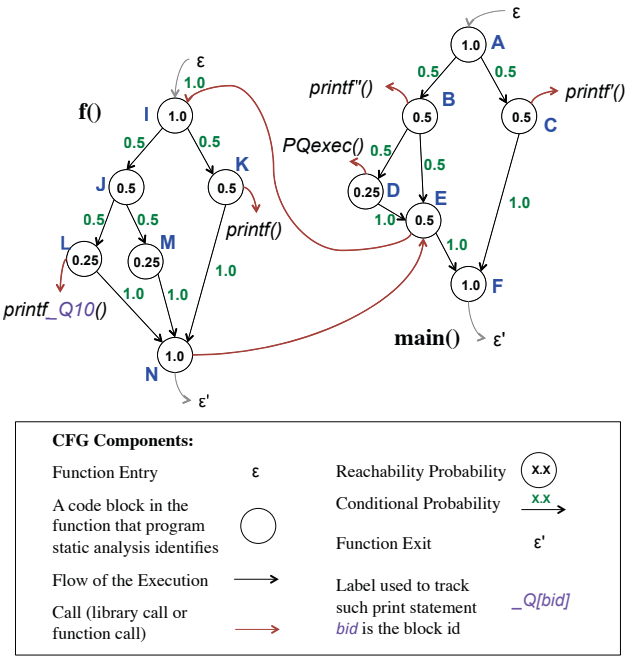


Fig. 3: Control Flow Graphs (CFGs) of two functions in a small program

- Control Flow Graph (CFG): Each function in the program is represented by a CFG. The graph is a directed graph that is built using static analysis of the program. A *node* in the graph represents a code block in a function. The *edges* represent the control flow of the execution, such as conditional branches and calls. An edge that represents a conditional branch between two nodes, say i and j represents an execution path in which i is executed before j. Calls include system calls, library calls, or user-defined function calls. Figure 3 shows an example CFG of a program.
- Data Dependency Graph (DDG): A data dependency is a situation in which an output statement (e.g., printf, fprintf, sprintf, snprintf, fputc, fputs, write, and fwrite), in an application program, refers to data retrieved from the DB by input statements (e.g., PQexec, mysql_query, ... etc.).
- Conditional Probability ($P_{xy}^{c_f}$): It is the occurrence probability of node y in function f given that node x has been executed, i.e., $P(n_y|n_x)$.
- Reachability Probability for node x in function f ($P_x^{r_f}$): Its value represents the likelihood that the execution of function f reaches node x.
- Transition Probability ($P_{xy}^{t_f}$): Its value is the likelihood that the call pair ($c_x \rightarrow c_y$) occurs in function f.
- Call Transition Matrix (CTM): This matrix records the transition probability $P_{xy}^{t_f}$ for each call pair ($c_x \rightarrow c_y$).

B. System Architecture

In what follows, we introduce the components of our system. Figure 4 shows the flow of information between the different components during the training and detection phases.

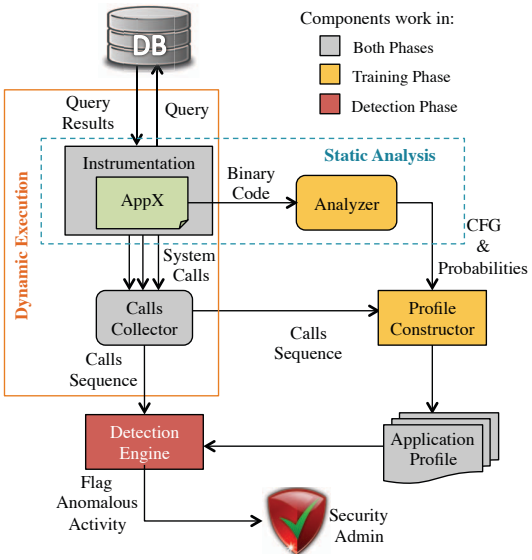


Fig. 4: System architecture

*1) **Analyzer**:* It takes as input an executable program authorized to access the DB. It statically builds the program's DDG. AD-PROM considers the data dependencies between output statements and input statements that retrieve the TD from the DB (e.g., PQexec, mysql_query, ... etc.). The *Analyzer* labels such output statements based on the program's DDG. The *Analyzer* then extracts the program's CFG and passes the CFG to the *Profile Constructor* (see IV-B3).

*2) **Calls Collector**:* It collects the calls that the application program issues at run-time. During training phase, the *Calls Collector* sends the program traces to the *Profile Constructor*. During the detection phase, it interacts with the *Detection Engine* and sends it n-length call sequences.

*3) **Profile Constructor**:* It receives as input the library calls logs and the CFG from the *Calls Collector* and the *Analyzer*, respectively. It then builds the CTM used to initialize the HMM model. If the application program has a large number of hidden states (i.e., more than 900), the *Profile Constructor* applies a data reduction technique (see Section IV-C4) to cluster similar system calls in order to reduce the training time. The clustered result is then used to initialize the HMM model. It trains the model with the program traces to create the application profile.

*4) **Detection Engine**:* It receives n-length call sequences. It uses the trained HMM model to check the probability of this call sequence and inspects the profiles to compare the resulting probability value with a pre-defined *threshold*. If the probability value is *less* than the *threshold*, the *Detection Engine* flags the activity to the *Security Admin*.

C. Training Phase

AD-PROM executes the following steps to build the model of an application program used later to detect anomalies. We use the graphs in Figure 3 to explain the steps.

*1) **Program Analysis**:* At this step, the *Analyzer* analyzes the program to build the program DDG. The purpose of such step is to label all output commands that access data from the DB. This technique enables the system to track the presence of the sensitive data from the DB. For example, if the application writes query results to a file using the library function *fprintf*, the system will label this call to be *fprintf*$\_Q[bid]$, where $[bid]$ is the block id where the call is invoked. This label is used when the *Analyzer* performs program analysis, and when the *Calls Collector* captures the library call. Thus, they can recognize the output statements that redirect query results to a file/screen and their locations in the application program.

The *Analyzer* also extracts the control flow graph (CFG) of each function in the program. At the static analysis stage, AD-PROM does not handle loops and recursions as each node is visited once. However, loops and recursions are learned from program traces by the HMM model at the end of the dynamic analysis phase (training phase).

*2) **Probability Forecast**:* After extracting the CFG, AD-PROM executes the following steps for each function $f()$:

- Approximate the **conditional probability** P_{xy}^{cf} for each pair of adjacent nodes $(n_x \rightarrow n_y)$ in function $f()$.
- Compute the **reachability probability** P_y^{rf} for each node j. The computation is performed as follows:
  a. A top-down traversal is executed from the entry point of the function ε to the exit point ε'.
  b. A topological sorting is performed on all nodes.
  c. The computation follows the topological order and uses the equations below:

$$P_{xy}^{cf} = 1/ \text{ \#out going edges from parent node } x \quad (1)$$

$$P_y^{rf} = \sum_{\forall n_x \in \text{ parent set of } n_y} P_x^{rf} * P_{xy}^{cf} \quad (2)$$

The following examples, from Figure 3, illustrate the above steps.

Node B in function $main$ has one parent. The reachability probability of node B is computed as follows:

$$\begin{aligned} P_B^{r_m} &= P_A^{r_m} * P_{AB}^{c_m} \\ &= 1.0 * 0.5 = 0.5 \end{aligned}$$

Node E has 2 parents, and hence its $P_E^{r_m}$ is computed as follows:

$$\begin{aligned} P_E^{r_m} &= (P_B^{r_m} * P_{EB}^{c_m}) + (P_D^{r_m} * P_{ED}^{c_m}) \\ &= (0.25 * 1.0) * (0.5 * 0.5) = 0.5 \end{aligned}$$

- Compute the **CTM** for each function. To build the CTM, AD-PROM computes the transition probability P_{ij}^{tf} for each call pair (c_i, c_j) in function $f()$. AD-PROM finds a set L which is a set of nodes between nodes n_x, where c_i is called, and node n_y, where c_j is called. L has the following properties (see Figure 5):
  a. n_x is the first element in L and makes the call c_i.
  b. n_y is the last element in L and makes the call c_j.
  c. A directed path $(n_x, n_{x+1}, n_{x+2}, ..., n_{y-1}, n_y)$ exists such that all nodes in the path between n_x and n_y do not make

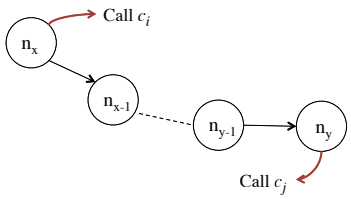


Fig. 5: Set of nodes between a pair of calls

any call. For example, in the CFG in Figure 3, the call pair $(\varepsilon, c_{printf''})$ in function $main$ has the set $L = \{A, B\}$. The call pair $(c_{printf''}, \varepsilon')$ has the set $L = \{B, E, F\}$. Also, the call pair $(\varepsilon, c_{PQexec})$ does not have such a path, and hence the set $L = \phi$.

After finding L for each call pair (c_i, c_j), the *Analyzer* computes the $P^{t_f}_{i_x j_y}$ from call c_i (in node n_x) to call c_j (in node n_y) of the call pair by equation 3:

$$P^{t_f}_{i_x j_y} = P^{r_f}_{x} \prod_{k=x}^{y-1} P^{c_f}_{k(k+1)} \tag{3}$$

TABLE I: CTM of function main() (mCTM)

main()	ε'	$printf'$	$printf''$	$PQexec$	$f()$
ε	0	0.5	0.5	0	0
$printf'$	0.5	0	0	0	0
$printf''$	0.25	0	0	0.25	0
$PQexec$	0	0	0	0	0.25
$f()$	0.25	0	0	0	0

TABLE II: CTM of function f() (fCTM)

f()	ε'	$printf$	$printf\_Q10$
ε	0.25	0.5	0.25
$printf$	0.5	0	0
$printf\_Q10$	0.25	0	0

Table I shows the CTM of function $main()$, and Table II shows the CTM of function $f()$, from Figure 3. As an example, the transitional probability of the call pair $(c_{printf''}, \varepsilon')$, in $main()$, is computed as follows. We first find the set of nodes between the pair of calls $L = \{B, E, F\}$. Then, we calculate the value using equation 3.

$$\begin{aligned} P^{t_m}_{B_{p''} F_{\varepsilon'}} &= P^{r_m}_{B} * (P^{c_m}_{BE} * P^{c_m}_{EF}) \\ &= 0.5 * (0.5 * 1.0) = 0.25 \end{aligned}$$

where p'' stands for $printf''$, and m stands for $main$.

As another example, the transitional probability of the call pair $(\varepsilon, c_{PQexec})$ is 0. The reason is that the only path between ε and $PQexec$ has another system call, that is, $printf''$ in node B.

*3) **Aggregation**:* The *Profile Constructor* uses the CG of the program and the CTMs of all functions to build one large transition matrix that we refer to as program's transition matrix (pCTM). The order of aggregation is a reverse topological order; $f_i()'s$ matrix is aggregated in $f_{i-1}()'s$ for $i = n, n-1, ...3, 2$. The method is similar to the aggregation method used by Xu et al. [11]. We discuss how the aggregation is performed using the example in Figure 3. First AD-PROM traverses the CG and aggregates the callee's matrix (e.g., function f() and its CTM, referred to as fCTM into the caller's matrix (e.g., function $main()$ and its CTM (referred as mCTM). This operation is called *in-lining*. The cases that AD-PROM has to cover are shown in Figure 6.

- *Case 1:* the call that precedes the call to $f()$; we refer to it as (c_{m_i}, f).
- *Case 2:* the call pair that follows the call to $f()$; (f, c_{m_j}).
- *Case 3:* the call pair within $f()$; $(c_{f_i}, c_{f_{i+1}})$.
- *Case 4:* the callee function that does not have any call.

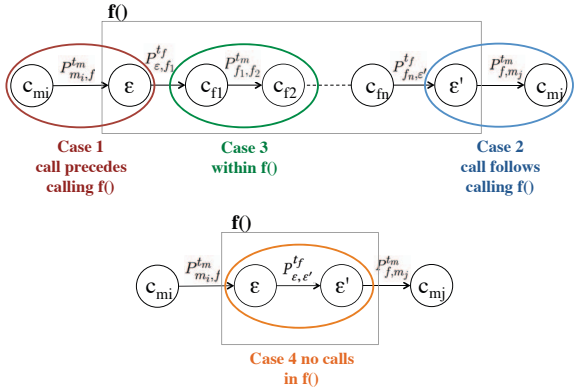


Fig. 6: Aggregation cases

The *aggregation algorithm* works as follows:

- **Case 1 ($f()$ is the child):** For each call c_{f_k} in the first row in fCTM, if fCTM$[\varepsilon][c_{f_k}] \neq 0$, then this call has a transition value with a system call from $main()$. We have two cases.
  a. If $(c_{m_i}, c_{f_k}) \notin m$CTM, **add a column** for c_{f_k} to mCTM, and update $P^{t_m}_{m_i, f_k}$ as follows.
  $$P^{t_m}_{m_i, f_k} = P^{t_m}_{m_i, f} * P^{t_f}_{f_\varepsilon, f_k} \tag{4}$$
  b. If $(c_{m_i}, c_{f_k}) \in m$CTM, update $P^{t_m}_{m_i, f_k}$ as follows.
  $$P^{t_m}_{m_i, f_k} = P^{t_m}_{m_i, f_k} + P^{t_m}_{m_i, f} * P^{t_f}_{f_\varepsilon, f_k} \tag{5}$$
- **Case 2 ($f()$ is the parent):** $\forall$ call c_{f_k} in the first column in fCTM, if fCTM $[c_{f_k}][\varepsilon'] \neq 0$ then this call has a transition value with a system call from $main()$. We have two cases:
  a. If $(c_{f_k}, c_{m_i}) \notin m$CTM, add a **row** for c_{f_k} in mCTM, and update $P^{t_m}_{f_k, m_i}$ as follows.
  $$P^{t_m}_{f_k, m_i} = P^{t_f}_{f_k, f_{\varepsilon'}} * P^{t_m}_{f, m_i} \tag{6}$$
  b. If $(c_{f_k}, c_{m_i}) \in m$CTM, update $P^{t_m}_{f_k, m_i}$ as follows.
  $$P^{t_m}_{f_k, m_i} = P^{t_m}_{f_k, m_i} + P^{t_f}_{f_k, f_{\varepsilon'}} * P^{t_m}_{f, m_i} \tag{7}$$
- **Case 3 (the call pair within $f()$):** $\forall$ call pair (c_{f_k}, c_{f_l}) in $f()$, there are two cases.
  a. If $(c_{f_k}, c_{f_l}) \notin m$CTM, **add a column and row** for c_{f_k} and c_{f_l} in mCTM, and update $P^{t_m}_{f_k, f_l}$ as follows.
  $$P^{t_m}_{f_k, f_l} = (\sum_i P^{t_m}_{m_i, f}) * P^{t_f}_{f_k, f_l} P^{t_m}_{f, m_i} \tag{8}$$

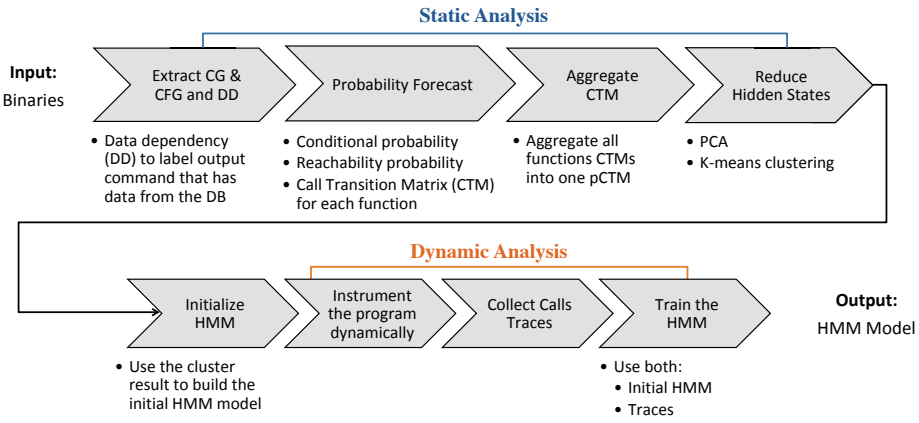


Fig. 7: Training phase workflow

b. If $(c_{f_k}, c_{f_l}) \in m$CTM, update $P^{t_m}_{f_k,f_l}$ as follows.

$$P^{t_m}_{f_k,f_l} = P^{t_m}_{f_k,f_l} + (\sum_i P^{t_m}_{m_i,f}) * P^{t_f}_{f_k,f_l} P^{t_m}_{f,m_i} \quad (9)$$

- **Case 4 ($f()$ does not make any call):** we simply remove all columns and rows corresponding to the function from mCTM. Also, we update the transitional probability $P^{t_m}_{m_i,m_j}$ of the call pair (c_{m_i}, c_{m_j}) in the mCTM. The new value is calculated by equation 10.

$$P^{t_m}_{m_i,m_j} = P^{t_m}_{m_i,m_j} + (P^{t_m}_{m_i,f} * P^{t_f}_{\varepsilon,\varepsilon'} * P^{t_m}_{f,m_j}) \quad (10)$$

The *output* of the *aggregation* is a matrix that represents the whole program; we refer to it as program call transition matrix (pCTM). The matrix has the following properties: (1) the sum of all the elements of the first row is equal to 1, (2) the sum of all the element of the first column is equal to 1, (3) for each call in CTM, the sum of the incoming probabilities is equal to the sum of outgoing probabilities.

*4) **HMM Initialization and Training**:* In AD-PROM, we use the pCTM to initialize the parameters of the HMM. However, using a one-to-one correlation between system calls and hidden states would not be efficient due to space overhead. We thus use the reduction method by Xu et al. [12] since it has shown high accuracy and efficiency in the experiments. For reduction purposes, AD-PROM uses a *many-to-one mapping* technique, in which a hidden state in an HMM may be initialized to represent multiple system calls that are similar in terms of incoming and outgoing system calls. To implement this mapping, AD-PROM performs the following **reduction** steps. First, for each system call C, the system creates a call-transition vector (CTV), that is, a vector consisting of both the transition-from (i.e., column) probabilities of C and the transition-to (i.e., row) probabilities of C. For a CTM of dimension n, a CTV of a call C is of size $2n$. For example, consider the fCTM in Table II, the CTV of call *printf_Q*10 is $< 0.25, 0, 0, 0.25, 0, 0 >$. Then, AD-PROM builds a new matrix where each row i represents a call c_i and contains the corresponding CTV. We refer to this matrix as program call-transition vectors (pCTV).

AD-PROM then uses PCA to reduce the vectors dimension since the vectors are sparse; hence reducing the training time. The pCTV is given as input to PCA, and the output of this step is PCA-pCTV. Subsequently, AD-PROM uses the K-clustering algorithm to reduce the hidden states in the model. AD-PROM passes the PCA-CTVs as input to the clustering algorithm. The similarity between each two system calls vectors is measured by the sets of incoming and outgoing system calls, and the probabilities distribution from incoming calls and to outgoing calls.

The results of the clustering step are then used to **initialize** the HMM model (λ). Namely, N will be the number of the clusters. System calls that have similar CTVs belonging to the same cluster are associated with the same hidden state (i.e., S_i). Hence, the corresponding emission probability vector (i.e., B_i) has the *averaged* vector. The transition probabilities vector (i.e., A_i) is averaged as well.

AD-PROM further **trains** the model with the program traces. This step is considered a learning step, in which the model adjusts the parameters of the HMM model. Thus it can handle dynamic program behaviors that cannot be learnt statically (e.g., loop and recursions). After this step, the model λ can be used to classify a given sequence of observations (i.e., a sequence of library/system calls) as normal or anomalous behavior of the program. The method improves the sensitivity of the detection, and has high likelihood of recognizing new legitimate activities (i.e., call sequences) [11], [12]. Figure 7 presents an overview of the training phase workflow.

D. *Anomaly Detection*

Detection Workflow. After building the HMM model λ, AD-PROM dynamically instruments the monitored application to label the output commands that display/redirect data retrieved from the DB. The system also collects call sequences while the monitored application is running. The *Calls Collector* sends n-length call sequences (cs) to the *Detection Engine*. The sequence includes last call and the $n-1$ past calls. The *Detection Engine* computes the probability that the observed call sequence is generated by the model $P(cs|\lambda)$.

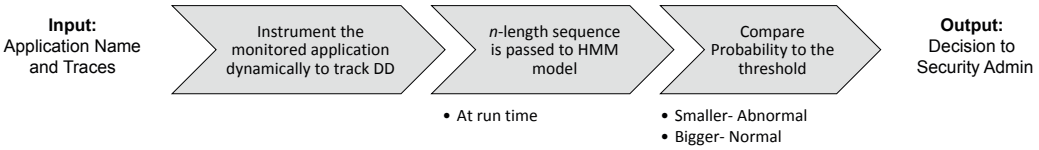


Fig. 8: Detection phase workflow

After computing $P(cs|\lambda)$, the system compares it to a *threshold* to determine whether the cs is anomalous or not. If the probability is lower than the *threshold*, then the *Detection Engine* flags the action as an anomalous one to the security administrator. Figure 8 shows an overview of the detection phase workflow.

Figure 9 shows how labeling output commands that belong to the DDG enables the system to detect the changes in the code. In the figure, at line 9, the program prints data retrieved from the DB. The attacker could modify the code by adding new commands (lines 10 and 11) that issue a call sequence similar to the original one. However, recording the position of the blocks where the original print call is supposed to be called helps AD-PROM to recognize the difference between line 9 and line 11 in the example.

Original code:

```
1   query1 = "SELECT COUNT(*) FROM employees";
2   query2 = "SELECT COUNT(*) FROM employees
              WHERE yearlyIncome < 30000";
3   result1  = PQexec(query1);
4   result2  = PQexec(query2);
5   allEmps  = PQgetvalue(result1,0,0);
6   empLowIn = PQgetvalue(result2,0,0);
7   percentage = (float) empLowIn/allEmps;
8   if (percentage > 0.6)
9      printf("%f Majority of the employees
        have low income.", percentage);
10  printf("Tax for such income is less than
        18%% in IN state.");
```

Resulting call sequence when line 9 is executed , *without* labels:
`PQexec, PQexec, PQgetvalue, PQgetvalue, printf, printf`

AD-PROM finds the DDG and label output statements that access queries' results (i.e., `printf` in line 9):
`printf_Q[block_id]`, e.g., `printf_Q6`

How AD-PROM captures the call sequence when line 9 is executed:
`PQexec, PQexec, PQgetvalue, PQgetvalue, printf_Q6, printf`

Modified code:

```
    //lines 1-9 are the same
6   empLowIn = PQgetvalue(result2,0,0);
7   percentage = (float) empLowIn/allEmps;
8   if (percentage > 0.6)
9      printf("%f Majority of the employees
        have low income.", percentage);
10  else
11     printf ("Number of the employees who
        have low income is %d.", empLowIn );
12  printf("Tax for such income is less than
        18%% in IN state.");
```

Call sequence is *similar* when line 9 or 11 is executed , *without* labels:
`PQexec, PQexec, PQgetvalue, PQgetvalue, printf, printf`

During detection phase, AD-PROM finds the DDG and label output statements that access queries' results (i.e., `printf` in lines 9 and 11):
`printf_Q6` and `printf_Q7`

How AD-PROM captures the call sequence when line 9 is executed:
`PQexec, PQexec, PQgetvalue, PQgetvalue, printf_Q6, printf`

How AD-PROM captures the call sequence when line 11 is executed:
`PQexec, PQexec, PQgetvalue, PQgetvalue, printf_Q7, printf`

Fig. 9: Dynamic instrumentation during detection phase

Threshold Selection. Choosing the probability threshold is an important step in any AD system. Small thresholds could reduce false positive (FP) alerts but increase false negative (FN) rate. A high threshold leads to high FP rate and low FN rate. Thus, the threshold should be carefully chosen. Selecting the threshold value for AD systems has been widely investigated. One simple method is to perform cross validation during the training phase using a set of predefined thresholds. Then, the value that achieves the best validation result is set to be the detector's threshold [28]. Another method is to use an adaptive threshold in which the security administrator can change the detector's threshold over time to reduce the false positive rate when there are legitimate changes in the program behavior over time [29].

V. Evaluation

In this section, we describe the implementation of each component of AD-PROM. We then discuss the results of the experiments we conducted to evaluate the performance of AD-PROM in the detection of anomalies and overhead.

A. Implementation

The *Analyzer* uses the *Dyninst* library [23] to perform the static analysis. Before starting the analysis, the application programs are compiled with static linking. The *Calls Collector* also uses the *Dyninst* library to instrument the application programs for intercepting library calls along with the caller function. The *Profile Constructor* and the *Detection Engine* are written in Java and use the *Jahmm* library [30] to train the HMM model and evaluate the call sequences, respectively.

B. Preparing the Dataset

We used two datasets to evaluate AD-PROM. The first one consists of three client applications (CA-dataset). We use this dataset for evaluating the system accuracy at detecting DL attacks. The other dataset consists of four of the program applications provided by the Software artifact Infrastructure Repository (SIR) [31] (SIR-dataset). We use this dataset for testing the analysis technique of AD-PROM on real programs with many test cases and call sequences.

CA-Dataset. We searched GitHub repositories for open source client applications suitable for our experiment. We selected three applications that implement real database client

applications. The first is a mini hospital client application. The second is a small banking system. The third is a supermarket management system program. We refer to the applications as App_h, App_b, and App_s, respectively. They execute different types of transactions containing DML queries. Table III shows statistics about the applications. We created test cases with as high coverage as possible of the applications. We derived the required traces by running the test cases and collecting their library calls traces.

TABLE III: Statistics about the CA-dataset

Client App	App$_h$	App$_b$	App$_s$
#states	59	139	229
DBMS	PostgreSQL	MySQL	MySQL
#test cases	63	73	36
#sequences	3810	10286	4053

SIR-Dataset. We used four applications provided by the Software artifact Infrastructure Repository (SIR) [31], namely, *grep, gzip, sed,* and *bash*. We refer to them as App1, App2, App3, and App4, respectively. SIR provides a large set of test cases that helped us to train the HMM model. Table IV shows statistics of the dataset. We derived the required traces by running the test cases and collecting their library calls traces.

TABLE IV: Statistics about the SIR-dataset

	App1	App2	App3	App4
#Test Cases	809	214	370	1061
Branch Coverage	58.70%	68.50%	72.30%	66.30%
Line Coverage	63.30%	66.90%	65.60%	59.40%
Traces	34770	69866	14514	6628647

Datasets Partitioning. For both datasets, we set the length of the sequences to 15 as previous work has shown that using sequences of length $10-30$ for classification produces more accurate results compared to shorter sequences (i.e., less than 6) [32]. We considered all sequences obtained by running the test cases as *Normal-sequences*. Only *Normal-sequences* were used to train the HMM model, and the model gave these sequences *high* probabilities (i.e., 1). To avoid overfitting, we kept about $1/5$ of the normal data aside, and used it to determine when data training should terminate. We refer to this data as the converge sub-dataset (CSDS). We run multiple training rounds; after each round, the intermediate resulting model is tested on the CSDS. The system stops the training with a converged model (λ) once it does not notice any improvement on the CSDS. To further enhance the training and maximize the use of the available data, we perform *k-folds cross validation* on the rest $(4/5)$ of the data, where *k* here is equal to 10.

C. AD-PROM vs. CMarkov

The goal of this experiment is to evaluate the system accuracy at detecting the following attacks: modifications to a query in a program to retrieve more records than intended; and insertion of an output command that might leak the data, i.e., printing a query result on the screen. We used CA-Dataset for this experiment.

Attack Detection. We created the following attacks to evaluate the systems. In the first three attacks we assume that the attacker has access to the source code. In the fourth attack, the attacker has access only to the program's binaries. In the last attack, the attacker does not have access neither to the source code nor to the binaries:

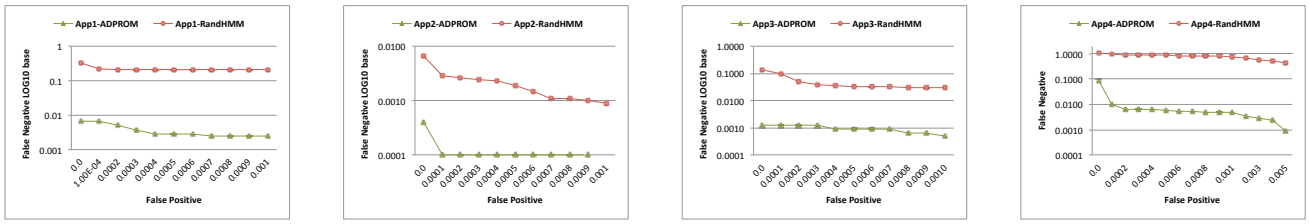
- Attack 1: The attacker inserts a new printing command in the program that is similar to another command in another branch of the program.
- Attack 2: The attacker changes the source code by inserting a new call in a different function to print query results.
- Attack 3: The attacker reuses an existing print command to print the TD. The attacker changes the arguments to a printf call to print a field from the query result.
- Attack 4: The attacker has access to the program binaries only and makes changes to the program by adding code patches to the binaries using an instrumentation tool. To carry out this attack, we used Dyninst [23]. The tool allows one to rewrite the binaries and insert commands to display the TD or redirect it to a file.
- Attack 5: We found a vulnerability in App_b. A query is sent to the DB without using prepared statements. The query should retrieve only a client record by his/her account number. The attacker however could retrieve all records by injecting the malicious value (1'OR'1'='1).

TABLE V: AD-PROM vs. CMarkov

	CMarkov	AD-PROM
Attack 1	undetected	detected & connected to source
Attack 2	detected	detected & connected to source
Attack 3	undetected	detected & connected to source
Attack 4	detected	detected & connected to source
Attack 5	detected	detected & connected to source

We compared AD-PROMwith CMarkov [12] as CMarkov is the closest to our system. It uses a similar program analysis technique to initialize the HMM model. However, CMarkov does not perform data flow analysis, and it cannot distinguish between different paths in the program. Table V shows what each system can detect and its ability to connect activities to the source of the data. In Attack 1, recording the block ids enabled AD-PROM to detect the anomalous call sequence that is similar to another call sequence. In Attack 3, analyzing data flow and labeling the calls enabled the system to detect the attacks.

To confirm that the attacks are detected correctly, we ran each test case and collected the traces; once with the original program and once with the malicious copy. We then sent the traces to the models and compared the flags issued for the traces by both systems. In case of AD-PROM, the system issues different flags to help the security admin in better understanding the attack. It has three flags: (1) Out of context flag for inserting a new call in a function. (2) DL flag for anomalous sequences that have output calls with the TD. (3) Anomalous flag if the abnormal sequences do not have output call with the TD. (4) Normal flag for the rest of the sequences. On the other hand, CMarkov cannot distinguish anomalous actions on the TD from other activities.



(a) App1 (b) App2 (c) App3 (d) App4

Fig. 10: Comparison of AD-PROM and Rand-HMM FN rates (in Y-axis, log10 base) for the application programs (App1, App2, App3, and App4) under the same FP rates (in X-axis)

Performance. In terms of **time** performance due to calls collection, CMarkov uses *ltrace* to collect the library calls and the instruction pointers. It then uses *addr2line* to translate the pointers to the caller functions. However, in AD-PROM we built our own collector using the instrumentation tool Dyninst [23]. Our collector collects both calls and callers while the program is running. The instrumented applications showed a short delay for few seconds before they start executing the program due to the execution of the data flow analysis. However, once the analysis is completed, the program runs normally. The overhead due to intercepting library calls is low compared to *ltrace*. Table VI shows the difference between the performance of AD-PROM's *Calls Collector* and *ltrace*. To measure the overhead we ran test cases that perform many printing calls (test cases 1 and 2), and executing multiple queries (test cases 3 and 4). On the average the *Calls Collector* of AD-PROM decreases 78.29% of the overhead that *ltrace* introduces. The reason of this performance enhancement is that we reduce the amount of information that is collected. Unlike *ltrace*, we only collect the names of the library calls without their arguments. In terms of space, the averaged size of an application's profile is about $\sim 31k$.

TABLE VI: *Calls Collector* vs. *ltrace* performance

Test case	*ltrace*	*Calls Collector*	Overhead Decrease
1	0.52101	0.01403	97.30%
2	3.0546	0.17736	94.19%
3	0.003956	0.001518	61.63%
4	0.00345	0.00138	60.04%

D. *Scalability Experiment (Rand-HMM vs. AD-PROM)*

In this experiment we: (1) test the analysis technique of AD-PROM using real programs, (2) compare the performance of AD-PROM and the regular HMM model that initializes the model randomly (Rand-HMM), and (3) assess AD-PROM ability of detecting different types of synthetic anomalous sequences.

We used synthetic *Anomalous-sequences* to test the model. We generated three types of *Anomalous-sequences* by: (1) replacing part of a *Normal-sequence* (i.e., last 5 calls) with random calls from the set of legitimate calls[2] (sequence A-S_1); (2) adding new library calls that do not belong to the legitimate calls (sequence A-S_2); and (3) increasing the frequency of legitimate calls (sequence A-S_3).

Accuracy Evaluation. For accuracy evaluation, we computed the false negative and false positive rates. Let TP be the number of true positives, TN be the number of true negatives, FP be the number of false positives, and FN be the number of false negatives. In our experiments, a correct detection of an anomalous sequence is a TP, whereas missing such sequence is a FN. Our goal is to maximize the number of detected anomalies (high TP), and to minimize the number of normal sequences that are classified as anomalies, that is, to minimize FP. We evaluated the system accuracy using the following metrics:

- FN $= \frac{|A:P_A>T|}{|A|}$, where A is the set of anomalous sequences, and P_A is the probability of an anomalous sequence.
- FP $= \frac{|N:P_N<T|}{|N|}$ where N is the set of normal sequences, and P_N is the probability of a normal sequence.
- FP Rate $= \frac{FP}{(FP+TN)}$
- FN Rate $= \frac{FN}{(FN+TP)}$
- Precision $= \frac{TP}{(TP+FP)}$
- Recall $= \frac{TP}{(TP+FN)}$
- Accuracy $= \frac{(TP+TN)}{(TP+TN+FP+FN)}$

The experiment has two parts. The first compares the performance of AD-PROM and the regular HMM model that initializes the model randomly (Rand-HMM) [33]. We compared the ability of both models to recognize new sequences including the normal sequences (through the cross validation's folds) and A-S_1 sequences. Figures 10a, 10b, 10c and 10d present accuracy results for each model for application programs App1, App2, App3 and App4, respectively. The results show that AD-PROM outperforms the Rand-HMM in all the cases.

The second part evaluates AD-PROM ability to detect the anomalous sequences A-S_2, and A-S_3. For this experiment, we computed the confusion matrix[3] of each model. We conducted the experiments on thousands of programs' traces. AD-PROM shows high accuracy in detecting new sequences. It was also very accurate in detecting legitimate library calls that are out of context (i.e., a library call issued from a function that usually does not issue such a call). Overall the system performed very well at detecting anomalous sequences with an

[2]Legitimate calls refer to the calls that were observed during the training phase when the system analyzed the application and built the model.

[3]A confusion matrix is a table often used in machine learning to indicate the performance of a classification model on a dataset where the true values are known.

averaged accuracy of 0.997%. Table VII presents the results about the accuracy of the models λ_{App1}, λ_{App2}, λ_{App3}, and λ_{App4}. Analyzing the program to initialize the HMM model beside having a large dataset played an important role at increasing the detection accuracy which was also shown by other researchers [11], [12].

TABLE VII: Confusion matrix of the programs' models

	#seq.	TP	TN	FP	FN	Rec.	Prec.	Acc.
App1	1245	91	1148	6	0	1	0.94	0.9952
App2	65885	82	65793	4	6	0.93	0.95	0.9998
App3	3131	153	2971	7	0	1	0.96	0.9978
App4	67626	92	67525	8	1	0.99	0.92	0.9999

Performance (Time overhead). We measured the elapsed time to perform each step to build the program's models. Table VIII shows the results we obtained for the pre-training steps: the time to build the CFG including parsing the binaries, the time to estimate the probabilities (e.g., computing conditional probability, reachability probability and transition probability) for each function in the program, and the time to aggregate all functions' CTMs to build the program's pCTM. Each step requires few seconds even for large programs, like App4. Training the models is also fast for small programs (e.g., App1, App2, and App3). Unlike small programs, training large programs (i.e., App4) is time consuming. However, we noticed that using clustering to speedup the training works very well with programs with large numbers of hidden states (i.e., greater than 900). We ran the K-means clustering algorithm on *bash* with K is 0.3. The number of hidden states before reduction was 1366 and after the clustering became 455. The training time was reduced by about $\%70$.

TABLE VIII: Elapsed time to perform training steps

Time (sec)	App1	App2	App3	App4
Build CFG	0.42	0.12	0.23	1.65
Probabilities Est.	1.99	0.40	1.14	7.18
Aggregation	58.83	46.84	53.94	237.31

VI. Related Work

Related work includes: (a) AD systems that only monitor queries and commands issued to the DB, but do not monitor the use of data by application programs - they are complementary to our system; (b) Approaches that use an HMM-based AD approach.

Query Monitoring Systems. Several systems were proposed working **at the DB level** to monitor activities on data at rest [34], [8], [9], [10]. DEMIDS [34] is a system to detect misuses in relational DB by building profiles that characterize the normal access patterns based on audit logs. Also, Sallam et al. [8] proposed an AD system to detect anomalies in submitted queries. Their system uses multiple classifiers (i.e., binary) to flag queries as anomalous when different attributes or selectivity appear compared to training queries. Other AD systems were introduced that work **at the application level**. Bossi et al. [5] proposed a technique using concolic testing technique to assure that transactions preserve the correct order of SQL commands and that no query in the program is deleted or altered, or additional queries added. The approach tracks the control and data flow of the application and compares with the expected flows. However, under such an approach profiles can be incomplete as the approach relies only on program flow and does not learn the history of the program behavior. If a program executes a path that is not frequently executed, the system might decide that is normal. DIDAFIT [35] is a system to detect abnormal accesses to the DB by checking statements against a set of known legitimate DB transactions fingerprints. DIDAFIT does not take into account the control and data flow of the program, i.e., the system will always flag an existing (rare) path in the program as anomalous if it is not in the training data.

Data Use Monitoring Systems. AD-PROM is analogous to the STILO [11], CMarkov [12], and [33] systems. The main difference between our system and STILO [11] and CMarkov [12] is the goal, that is, STILO and CMarkov aim at detecting code-reuse attacks, whereas AD-PROM aims at detecting anomalies in the program behavior. STILO and CMarkov were proposed as HMM-based AD systems to analyze application programs behavior statically and dynamically. However, they are unable to track the source of the data on which activities are performed. Guevara et al. [33] proposed a system with the same goal of AD-PROM. The system uses an HMM-based AD system. However, it initializes the HMM model randomly which might affect the accuracy of the model. Furthermore, it relies only on program traces without considering the program control flow.

VII. Limitations

One of the system's limitations is that if the attacker knows that the training dataset consists of call traces without recording queries or queries' metadata (i.e., query signature), the attacker can issue new queries with similar selectivity to avoid changing the call sequences that the program issues. This case is not detected by our system; however, recording queries signatures along with library calls can mitigate this case. Another way to solve this problem is by deploying complementary systems like detAnom [5] to detect anomalous queries.

Also our system relies on training dataset to handle loops. It thus might fail at handling loops and produce false positive alerts if the training dataset is not sufficient. This can be mitigated by adding an intermediate stage between training and detection phases to collect more data to learn a better threshold that reduces the FP rate.

There are other methods to leak data beside printing and storing the TD to a file, such as sending the TD through the network. The attacker could inject code in the application program to send the TD via email using, for example, $system$ command. Such change can be detected since the injected code will issue a new call sequence. Another way to send the data through the network is by storing the TD to a file and then send the file over a network. This case is not handled currently by AD-PROM; however, it can be mitigated as follows. When a call like fprintf, write, or fwrite is issued and the data flow

analysis indicates that the call stores TD, the file is labeled. Then, actions on such files are monitored using auditing tools.

VIII. Conclusion and Future Work

We have designed, implemented and experimentally evaluated AD-PROM, a system for creating application programs behavior profiles used to detect anomalies in the program behaviors. Such anomalies are often indicative of attacks aiming at leaking data. AD-PROM uses both static and dynamic program analysis to build the program's profile, and considers both context and content of the data to track programs activities. AD-PROM uses an HMM to predict the likelihood of the occurrence of sequences of activities. We have evaluated the overhead of the system and its accuracy in detecting DL. As part of future work, we plan to consider types of applications other than desktop ones, i.e., web applications.

IX. Acknowledgement

We thank Dr. Kui Xu, Prof. Danfeng Yao and the rest of their research group for sharing the code of CMarkov [12].

References

[1] S. Alneyadi, E. Sithirasenan, and V. Muthukkumarasamy, "A survey on data leakage prevention systems," *J. Netw. Comput. Appl.*, vol. 62, no. C, pp. 137–152, Feb. 2016.

[2] Verizon, "2018 data breach investigations report," Verizon, Tech. Rep., April 2018.

[3] McAfee, "Data exfiltration study: Actors, tactics, and detection," McAfee, Tech. Rep., 2017.

[4] McAfee, "Mcafee labs threats report," McAfee, Tech. Rep. McAfee Labs Threats Report: September 2018, Sep 2018.

[5] 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 (CODASPY'15)*. New York, NY, USA: ACM, 2015, pp. 25–35.

[6] D. Fadolalkarim, A. Sallam, and E. Bertino, "Pandde: Provenance-based anomaly detection of data exfiltration," in *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*. New York, NY, USA: ACM, 2016, pp. 267–276.

[7] D. Fadolalkarim and E. Bertino, "A-pandde: Advanced provenance-based anomaly detection of data exfiltration," *Computers & Security*, vol. 84, pp. 276–287, 2019.

[8] A. Sallam, D. Fadolalkarim, E. Bertino, and Q. Xiao, "Data and syntax centric anomaly detection for relational databases," *Wiley Int. Rev. Data Min. and Knowl. Disc.*, vol. 6, no. 6, pp. 231–239, Nov. 2016.

[9] A. Kamra, E. Terzi, and E. Bertino, "Detecting anomalous access patterns in relational databases," *The VLDB Journal*, vol. 17, no. 5, pp. 1063–1077, Aug. 2008.

[10] S. Mathew, M. Petropoulos, H. Q. Ngo, and S. Upadhyaya, "A data-centric approach to insider attack detection in database systems," in *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection (RAID'10)*, 2010, pp. 382–401.

[11] K. Xu, D. D. Yao, B. G. Ryder, and K. Tian, "Probabilistic program modeling for high-precision anomaly classification," in *Proceedings of the 28th IEEE Computer Security Foundations Symposium (CSF'15)*, 2015, pp. 497–511.

[12] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*, 2016, pp. 467–478.

[13] 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)*. New York, NY, USA: ACM, 2004, pp. 318–329.

[14] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy (SP'01)*, 2001, pp. 144–155.

[15] D. Gao, M. K. Reiter, and D. Song, "Beyond output voting: Detecting compromised replicas using hmm-based behavioral distance," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 96–110, April 2009.

[16] D. Yao, X. Shu, L. Cheng, and S. Stolfo, *Anomaly Detection as a Service: Challenges, Advances, and Opportunities*. Synthesis Lectures on Information Security, Privacy, and Trust. Morgan & Claypool Publishers, 2017.

[17] P. Wang, L. Shi, B. Wang, Y. Wu, and Y. Liu, "Survey on hmm based anomaly intrusion detection using system calls," in *Proceedings of the 5th International Conference on Computer Science Education (ICCSE'10)*, 2010, pp. 102–105.

[18] N. Goernitz, M. Braun, and M. Kloft, "Hidden markov anomaly detection," in *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*, vol. 37. Lille, France: PMLR, 2015, pp. 1833–1842.

[19] A. Ahmadian Ramaki, A. Rasoolzadegan, and A. Javan Jafari, "A systematic review on intrusion detection based on the hidden markov model," *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 11, no. 3, pp. 111–134, 2018.

[20] Z. Ghahramani, "An introduction to hidden markov models and bayesian networks," *International journal of pattern recognition and artificial intelligence*, vol. 15, no. 01, pp. 9–42, 2001.

[21] L. R. Rabiner and B.-H. Juang, "An introduction to hidden markov models," *ieee assp magazine*, vol. 3, no. 1, pp. 4–16, 1986.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[23] *DYNINST binary instrumentation technology*. [Online]. Available: https://www.dyninst.org/

[24] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, Mar. 2012.

[25] W. G. Halfond, J. Viegas, A. Orso *et al.*, "A classification of sql-injection attacks and countermeasures," in *Proceedings of the International Symposium on Secure Software Engineering (ISSSE'06)*, Washington D.C., USA, 2006.

[26] A. Baratloo, N. Singh, T. K. Tsai *et al.*, "Transparent run-time defense against stack-smashing attacks," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2000, pp. 251–262.

[27] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proceedings of the IEEE Symposium on Security and Privacy (SP'14)*. IEEE, 2014, pp. 227–242.

[28] R.-E. Fan and C.-J. Lin, "A study on threshold selection for multi-label classification," *Department of Computer Science, National Taiwan University*, pp. 1–23, 2007.

[29] A. Ghafouri, W. Abbas, A. Laszka, Y. Vorobeychik, and X. Koutsoukos, "Optimal thresholds for anomaly-based intrusion detection in dynamical environments," in *Proceedings of the International Conference on Decision and Game Theory for Security (GameSec'16)*. Springer, 2016, pp. 415–434.

[30] *JAHMM: An implementation of hidden Markov models in Java*, 2009. [Online]. Available: https://code.google.com/archive/p/jahmm/

[31] *Software-artifact infrastructure repository*. [Online]. Available: http://sir.csc.ncsu.edu/portal/

[32] D.-Y. Yeung and Y. Ding, "Host-based intrusion detection using dynamic and static behavioral models," *Pattern Recognition*, vol. 36, no. 1, pp. 229–243, 2003.

[33] C. Guevara, M. Santos, and V. López, "Data leakage detection algorithm based on task sequences and probabilities," *Know.-Based Syst.*, vol. 120, no. C, pp. 236–246, Mar. 2017.

[34] C. Y. Chung, M. Gertz, and K. Levitt, *DEMIDS: A Misuse Detection System for Database Systems*. USA: Springer, 2000, p. 159–178.

[35] 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)*. Zurich, Switzerland: Springer, 2002, pp. 264–280.