Security-by-construction in web applications development via database annotations

Wassim El-Hajj a,*, Ghassen Ben Brahim b, Hazem Hajj c, Haidar Safa a, Ralph Adaimy a

a Computer Science Department, American University of Beirut, Lebanon
b Computer Science Department, Prince Mohammad Bin Fahd University, Saudi Arabia
c Electrical and Computer Engineering Department, American University of Beirut, Lebanon

ARTICLE INFO
Article history:
Received 9 October 2014
Received in revised form 23 December 2015
Accepted 27 December 2015
Available online 3 March 2016

Keywords:
Web applications security
Secure information flow
Program dependence graph
Database annotation
Security by construction

ABSTRACT
Huge amounts of data and personal information are being sent to and retrieved from web applications on daily basis. Every application has its own confidentiality and integrity policies. Violating these policies can have broad negative impact on the involved company’s financial status, while enforcing them is very hard even for the developers with good security background. In this paper, we propose a framework that enforces security-by-construction in web applications. Minimal developer effort is required, in a sense that the developer only needs to annotate database attributes by a security class. The web application code is then converted into an intermediary representation, called Extended Program Dependence Graph (EPDG). Using the EPDG, the provided annotations are propagated to the application code and run against generic security enforcement rules that were carefully designed to detect insecure information flows as early as they occur. As a result, any violation in the data’s confidentiality or integrity policies is reported. As a proof of concept, two PHP web applications, Hotel Reservation and Auction, were used for testing and validation. The proposed system was able to catch all the existing insecure information flows at their source. Apart from the proof of concept and to comprehensively test the performance of our system, we compared it to JLift, a state-of-the-art type-based system approach to detect information leaks. Both approaches were run against custom made PHP web applications and publicly available applications downloaded from SourceForge and GitHub. The results show that our approach outperforms JLift in terms of accuracy and the number of false alarms, and is able to catch the insecure flows at their source when they first occurred.

© 2016 Elsevier Ltd. All rights reserved.

1. Introduction

The past decade has witnessed a vast emergence in web applications’ technology. Web applications mainly have a three-tier model architecture composed of a client, server, and database. In this model, the server listens to requests coming from the client side, constructs database queries relevant to those requests, forwards these queries to the database, and finally retrieves the required information. However, the retrieved information, which will eventually be exchanged across the network, may have high confidentiality and integrity...
constraints and therefore should be handled with extra care. Facebook, Gmail, eBay, and Amazon are examples of such web applications that handle enormous amount of private and sensitive data. Facebook, for instance, stores more than 100 petabytes (100 quadrillion bytes) of photos and videos in a three-month timespan and has more than 901 million users, representing one-sixth of the world’s population. These numbers highlight the enormous amount of data stored on web application servers. Such data must be protected, an objective that mostly lies in the hands of web application developers. However, the authors in Chong et al. (2007a) state that even developers with good knowledge and expertise in security cannot manually ensure that the applications’ confidentiality and integrity policies are being enforced. This leads to our main motivation and contribution:

Motivation 1. The need for an accurate, robust, and dynamic system that is able to detect the release of sensitive data from web servers while relieving the developer from this burden.

Also in Chong et al. (2007a), the authors reported that web applications are not being implemented with sufficient built-in security assurance, jeopardizing the exposure of secret information to a third party. This was confirmed in Alessandro et al. (2006), where a study was conducted to quantify the impact of information leaks in web applications. The authors showed that high percentage of data breaches resulted from “Bad Security Practices”, part of which are the wrong choices or faulty implementation made during development time. These breaches lead to the leakage of sensitive information such as social security numbers, credit card numbers, and other personal information. The impact of such breaches is considered more dangerous than that caused by a Denial of Service type of attacks (Alessandro et al., 2006; Anat and D’Arcy, 2003). Consequently, more effort and priority should be given to breaches resulting from bad security practices and flaws in web development. This leads to our second motivation and contribution:

Motivation 2. The need to adopt a security-by-construction approach for web applications development ensuring that security is enforced during construction phase.

In this work, we propose a framework for securing, by design, the information flow in web applications. The system checks whether the confidentiality and integrity policies of web applications are violated. Contrary to existing systems where program variables or database tuples, fields, and queries are labeled for security, we associate security annotations with only database attributes then propagate them through the application (program) code via the traversal of an extended form of the program’s dependence graph (PDG). We call the new PDG representation “Extended PDG” (EPDG). The EPDG is then checked against customized predefined rules, which determine if a violation of the security policies has occurred. If a violation exists, the developer will be alerted and presented with the line number of the statement that caused the violation. A major advantage of our approach vs. existing approaches is its simplicity from the developer’s perspective, while achieving comparable or better results in terms of detecting information leaks. The developer needs only to annotate table attributes (columns) as opposed to annotating variables inside the code or annotating table tuples and queries that can be very problematic and prone to errors in large applications.

The proposed framework was validated and tested on PHP web applications. To efficiently generate the EPDG of PHP web applications, we created, from scratch, the front end of a PHP compiler using ANTLR. The compiler takes as input the PHP code and generates the code’s EPDG. In turn, the EPDG is traversed using simple graph traversal algorithms and checked against six pre-defined information flow rules that, if violated, a security alarm is triggered and the source of the information leak is reported.

The remainder of the paper is organized as follows. In Section 2, we present a classification of existing approaches for web application security analysis. In Section 3, we discuss the details of the proposed framework. In Sections 4 and 5, we validate our system on two test cases and present full performance results on larger scenarios, respectively. In Section 6, we include a discussion on the reasoning behind choosing 4 labels and 6 rules, and the possibility of increasing these labels/rules. In Section 7, we conclude the paper and present our future directions.

2. Approaches to securing web applications

There have been different approaches to the problem of securing web applications. These approaches can be classified into two broad classes: (1) Information flow control (IFC) based (Andrew, 1999; Benjamin and Lam, 2005; Chong et al., 2007a, 2007b; Christian and Snelting, 2009; Pasquier et al., 2014; Pistoia et al., 2005; Schoepe et al., 2014; Schultz and Liskov, 2013; Tuong et al., 2005), and (2) Non-Information control based (Chen et al., 2010; Huang et al., 2004; Krishnamurthy and Craig, 2009; Vikram et al., 2009). We focus in this section on the IFC-based approaches, which are more relevant to our work and just include a brief summary of non-IFC approaches.

IFC-based approaches were first introduced by the Dennings in the mid-seventies (Denning, 1976; Denning and Denning, 1977). They introduced the notation $x \to y$ to denote that an information flow from $x$ to $y$ is permissible. Information does flow from $x$ to $y$, denoted by $x \Rightarrow y$, if information that was stored in $x$ is transferred to $y$. Information flows are characterized as either explicit or implicit flows. An explicit flow $x \Rightarrow y$ can occur whenever an assignment statement such as $y = x$, or a method’s return value $y = \text{return\_value}(x)$ is executed. An implicit flow, on the other hand, can occur via a control channel, where the value of $y$ is dependent on an execution of the value of $x$. Based on these principles, IFC-based approaches branched out to include two sub-classes: (1) Typed-based systems and (2) Program Dependence Graph based systems.

2.1. Typed based systems

This approach works by having developers annotate application code and statements, or database tuples and queries (Andrew, 1999; Chong et al., 2007a, 2007b; Schultz and Liskov, 2013; Volpano et al., 1996) with security labels and then devise rules to decide on which flow is secure and which flow is not.
However, this approach requires the developers to be very well aware of the application’s security policies and needs a lot of effort and care while annotating program code and/or database queries/tuples. Moreover, this approach produces a high rate of false alarms as reported in Chong et al. (2007a) and King et al. (2008). The authors in Chong et al. (2007a) considered a simple “Guess-A-Number” web application to illustrate a major fault in typed based approach that causes false alarms. Consider, for instance, the following code fragment in Fig. 1.

In this piece of code, there is an implicit information flow from confidential to public. Hence, confidential is a variable that should remain confidential and public is a variable that can be publically accessible. However, note the re-definition of public at the end, which kills the implicit flow. Type-based systems cannot catch such re-definitions, and will report the code fragment above as an insecure information flow, although the value of public always remains 0 regardless of the value of confidential.

Swift (Chong et al., 2007a), SIF (Chong et al., 2007b), Flow Calm (Simonet and Rocquencourt, 2003), and JIF (Chong et al., 2006) are yet other typed based flow control systems that move the security assurance from the web application into the code itself. However, neither of these proposed languages provides integrated support with a database. For instance, Swift (Chong et al., 2007a) and SIF (Chong et al., 2007b) were built using JIF (Java based typed programming language (Chong et al., 2006)) to address various aspects of security when constructing multi-tier web applications. However, both languages essentially ignore the database tier (SIF focuses on servlet interactions and Swift considers client-server interactions). Our proposed approach is different from the aforementioned techniques in the sense that it attempts to directly protect the application data – database – instead of the application code. Moreover, the developer does not need to annotate every statement in the code; the developer will rather have very minimal annotations that target only the database attributes. We discuss next the PDG-based approaches (a sub-category of IFC-based approaches) for securing web applications.

2.2. Program dependence graph based systems

The Program Dependence Graph (PDG) approach consists of modeling the application code as a graph showing all control and data dependencies between statements within a procedure. A System Dependence Graph (SDG) is used to combine PDGs to model inter-procedural dependences. Details of constructing PDGs and SDGs are presented in Ricca and Paolo (2002) and Ferrante et al. (1987). PDG is considered to be a powerful tool to perform application security analysis. For instance, the authors in Christian and Snelting (2009) start by choosing some key nodes (code statements) within the graph, such as output nodes and nodes containing confidential data, and annotate them manually. Each selected node is then assigned two security annotations and one optional declassification annotation as follows:

- Provided Security Level \( P(x) \): A statement sends information with security level \( P(x) \).
- Required Security Level \( R(x) \): Only information with smaller security level than \( R(x) \) can reach this statement.
- Declassification Security Level \( x \in D \): Information reaching \( x \) with a maximal security level \( R(x) \) is lowered (declassified) down to \( P(x) \).
- Calculated Security Level \( S(x) \):

\[
S(x) = \begin{cases} 
P(x) & \text{if } x \in D \\
\bigcup_{y \in pred(x), y \in D} P(y) & \text{otherwise}
\end{cases}
\]

Finally, to enforce confidentiality, the following assertion must hold: \( \forall x \in \text{dom}(R): R(x) < S(x) \). Informally, this condition ensures that for every node with a provided and required security level, the required security level is not greater than the calculated security level. The approach we propose in this research work is a major improvement to the technique discussed in Christian and Snelting (2009). To illustrate a major weakness in the approach proposed in Christian and Snelting (2009), we have designed a hypothetical application, hotel reservation system, with the following confidentiality policy: “only one available room in the hotel can be presented to the client”. The fragment of code for the reservation system and its corresponding PDG are shown in Figs. 2 and 3 respectively. In a PDG, a dotted arrow
represents data dependence between two nodes, while a normal arrow represents control dependence. Following the approach described in Christian and Snelting (2009), an annotation of Low is given to output nodes 18 and 21, and an annotation of High is given to node 9, which is the result of the database call. Notice that nodes 10 and 11 in the graph are arrays. The authors differentiate between the actual array and an instance of the array by adding an extra [] node to arrays to represent array instances. Some methods are data or control dependent on the whole array, such as length() or empty(), and others are only control or data dependent on an instance, such as available[]. Following the rules presented in Christian and Snelting (2009) (Eq. 1), we populated Table 1, which illustrates that there exist violations in the hotel reservation system on lines 18 and 21.

Despite the fact that the method described in Christian and Snelting (2009) does catch insecure information flows, the insecure flow is only caught on output nodes, whereas in reality and as we shall see later, the insecure flow originates earlier than that from the time the database is queried. As a result, the exact source of the information leak is not detected using the method presented in Christian and Snelting (2009), causing a lot of confusion when the code is large. Another major drawback of this method is that, similar to the typed based approaches, it requires the developer to place annotations on graph nodes, which is just another variant of placing labels alongside program code. Developers who do not have enough knowledge in security might not be able to annotate nodes properly, especially when they encounter the notion of Required and Provided security levels, which might cause confusion.

Our proposed approach is different from the one presented in Christian and Snelting (2009) in the sense that it addresses the following two major limitations: (1) the annotation of graph nodes, which is equivalent to annotating code statements, and (2) the poor accuracy in reporting the violation source.

Apart from the IFC-based approaches for securing web applications, non-IFC based approaches exist. Under this class of approaches, some techniques attempt to analyze the code running on the server side to detect sections of code that might allow the flow of sensitive data such as user input forms. Consequently, runtime guards are inserted to achieve a secure flow (Huang et al., 2004). User input forms establish the main mechanism for allowing users to drive their private information into the web application servers and databases. Others attempt to analyze the code running on both client and server side (Huang et al., 2004; Vikram et al., 2009) then monitor data that flows throughout the different application tiers. Others discuss how personally identifiable information can be leaked in social networks and web search engines (Chen et al., 2010; Krishnamurthy and Craig, 2009). Unlike our approach that attempts to catch information leaks, this category of approaches (non-IFC-based approaches) protects against web attacks.

To recap, our approach provides the following contributions over both IFC-based and non-IFC-bases approaches:

- Very simple as opposed to approaches that annotate application code
- Very simple as opposed to approaches that annotate database fields, tuples, and queries
- Less prone to errors, given the few number of annotations required by the developers
- High accuracy in reporting the source of the violation
- Minimal developers’ knowledge in security and privacy

We discuss next the details of our framework design for insecure information flow detection.

### Framework design and implementation

Our proposed framework for analyzing web application code and checking for insecure information flows brings major improvements over existing solutions in the following areas: (1) it catches the violations at the source and reduces the false alarm rate, (2) it protects the application data itself by annotating database attributes into several levels of security classes, as opposed to the complex and error-prone approach of annotating database code or database queries/tuples, and (3) it enforces the application confidentiality by propagating the security policies from the database attributes to the corresponding EPDG. The proposed framework consists of 4 major blocks as depicted in Fig. 4. Each block is discussed hereafter.

#### 3.1 Database (DB) annotation

Contrary to existing approaches, where the focus was on protecting application code, we propose protecting application data

**Table 1 - Detecting an insecure information flow using Christian and Snelting (2009).**

<table>
<thead>
<tr>
<th>Node</th>
<th>R(x)</th>
<th>P(x)</th>
<th>S(x)</th>
<th>Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>L</td>
<td>L</td>
<td>$\cup({18}, 16, 17) \neq H$</td>
<td>$R(x) &lt; S(x)$ &amp; Yes</td>
</tr>
<tr>
<td>21</td>
<td>L</td>
<td>L</td>
<td>$\cup({21}, 16) = H$</td>
<td>$R(x) &lt; S(x)$ &amp; Yes</td>
</tr>
<tr>
<td>17</td>
<td>–</td>
<td>–</td>
<td>$\cup(11, 16) = H$</td>
<td>No</td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>–</td>
<td>$\cup(11, 12) = H$</td>
<td>No</td>
</tr>
<tr>
<td>11</td>
<td>–</td>
<td>–</td>
<td>$\cup(13) = H$</td>
<td>No</td>
</tr>
<tr>
<td>[]</td>
<td>–</td>
<td>–</td>
<td>$\cup(11) = H$</td>
<td>No</td>
</tr>
<tr>
<td>12</td>
<td>–</td>
<td>–</td>
<td>$\cup(10, 13) = H$</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td>–</td>
<td>–</td>
<td>$\cup(12) = H$</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>–</td>
<td>–</td>
<td>$\cup(9) = H$</td>
<td>No</td>
</tr>
<tr>
<td>[]</td>
<td>–</td>
<td>–</td>
<td>$\cup(10) = H$</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>H</td>
<td>H</td>
<td>$P(9) = H$</td>
<td>No</td>
</tr>
</tbody>
</table>
right at the source where they reside (i.e., database). The rationale behind this approach is the fact that private or secret information are usually stored in databases and are queried when needed. Under this scheme, database attributes will be annotated then propagated to the corresponding application dependence graph. After investigating the different types of data being manipulated by web applications and the way they are displayed to browsers, we propose the following 4 data categories: (1) Top Secret, (2) Single Disclosure, (3) Masking Needed, and (4) Public. Data in the database will be assigned to one of these categories based on its level of sensitivity. For instance, high sensitive data that should never be revealed or propagated across the network is classified as "Top Secret". Less sensitive data are assigned to "Single Disclosure" class. Next, we provide more description about each of the 4 proposed attribute classes.

**Top Secret**: Data classified as Top Secret are basically data that should never propagate to output. It can be used for internal computations, but it can never be sent to the client's browser as output. Examples of this type of data include passwords, credit-card numbers, reserved-price of an item in an auction, etc.

**Single Disclosure**: Data classified as Single Disclosure are data that are not allowed to flow to the output in groups. Specifically, only one instance of the data can propagate to output. A query to the database that includes data labeled as Single Disclosure can return a result array that contains several items, among which is the data labeled as Single Disclosure. However, only one tuple of the returned query can flow to output. For instance, consider a hotel reservation system that has the following confidentiality policy: the available rooms should not be known by a client; a client should be assigned a room based on his preferences. This confidentiality policy states that a query to the database asking for the available rooms is allowed, but transferring this information to the output is not; only one available room should be sent to output. Annotating the “Available Rooms” attribute in the database as Single Disclosure ensures that only one tuple of the table (i.e. one available room) can flow to the output.

**Masking Needed**: Data classified as Masking Needed are data that can be propagated to output with no restriction on the number of tuples, but rather with the restriction that these data have to be masked or encrypted before it is sent to output. A query to the database that involves data labeled Masking Needed might return a result array that contains several items, but any set of these items has to be masked before they are sent to output. An example of Masking Needed data is the names of bidders in auctions (eBay policy).

**Public**: This is the security class of all the other types of data. Public data can be sent to output without any constraints. Such data can be the list of available hotels in a certain city, list of users playing an online game, list of items available in an online shopping application or an auction, etc.

### 3.2. Program dependence graph (PDG) construction

PDG construction is a critical component in our system. PDG is known to be an intermediate representation of the program. In our work, it represents the data structure that will be used to propagate the security annotations we placed on the database table attributes to the program code and statements. Traversing the graph also helps track the flow of information, which has traveled to the output. Our approach to creating the PDG was based on that described in Ricca and Paolo (2002). The modifications we introduced consist of taking into consideration the following: (1) new DB annotations, and (2) DB queries. We refer to the new PDG as the Extended PDG (EPDG).

The PDG for a certain piece of code represents all data and control dependences among the code statements. The representation of arrays in the PDG, as shown in Christian and Snelting (2009), differentiates between an instance of the array and the array itself. Fig. 5 shows a piece of code and the corresponding PDG. Notice that the if statement is not dependent on the whole array, but rather on one instance of the array.
On the other hand, some statements will be dependent on the whole array rather than just an instance of it, as shown in Fig. 6. In order to accommodate the new database attribute annotations, the array representation in the PDG is further fragmented in order to handle the result of the database queries and eventually be reflected in the EPDG.

The implementation of the EPDG is composed of 4 major modules, each module feeding the other (Fig. 7).

- **Module for “Grammar construction and Parsing”**: Because of its popularity and flexibility, we decided to build, using ANTLR (Parr and Quong, 1995), the front-end compiler for PHP allowing us to parse PHP web applications. Hence, ANTLR is a parser generator that uses LL(*) parsing. We mainly used it as a tool to create a grammar for PHP and then attach actions to the grammar elements for the purpose of constructing the EPDG. In this grammar prototype, function calls and variables assigned through function calls are treated as assignment statements. These assignments will later be translated into a data dependence edge between variables belonging to the function parameters and the variable being assigned. Despite its limitation (lack of support of classes, objects, and function definitions), we believe that the initial prototype is sufficient to validate the proposed information leak analysis model. We plan to include the missing features in our next release.

- **Module for “EPDG node extraction”**: This is the first step toward the EPDG construction, which consists of extracting the nodes from the code under analysis. EPDG nodes are either assignment statements or predicate statements. In our implementation, assignment statements can assign variables to expressions, literals, function return values, or database query expressions. During the first parsing phase, whenever an assignment rule or predicate rule is encountered, a List of nodes, EPDGNodes, is updated to include the new node. This is done through actions inside the grammar. Consequently, after the first parsing phase, all the EPDG nodes are stored and ready to be input to the next module. What is left is to connect these nodes via edges representing data and control dependence.

- **Module for “EPDG construction”**: This module is responsible for constructing the control dependence graph (CDG) and the data dependence graph (DDG), resulting in the corresponding EPDG. Our approach was partially inspired by Chong et al. (2007b), where both graphs can be generated assuming that we already generated the Control Flow Graph (CFG) and using the following 2 rules for data and control dependence:

  **Rule 1**. Let D be a DDG with nodes n1 and n2. Node n2 is data dependent on n1 if
  
  a) Variable v is defined at n1 and used at n2
  
  b) There exists a path of non-zero length from n1 to n2 not containing any node that redefines v.

  **Rule 2**. Let C be a CDG with nodes n1 and n2, n1 being a predicate node. Node n2 is control dependent on n1 if there is at least one path from n1 to program exit that includes n2 and at least one path from n1 to program exit that excludes n2.

The needed CFG and the required CDG were built throughout the second parsing phase of the code; the required DDG was built, though, using the “after” action of the start rule of the grammar, which is basically, when parsing ends. So after parsing ends, a graph traversal mechanism (extended BFS) just reads the constructed CFW to determine data dependences based on the above Data Dependence rule (Rule 1). The extended BFS simply collects all paths from a certain node to the others instead of just determining whether a path exists; these paths are then traversed to check if any of the nodes redefines the targeted variable.

![Fig. 5 - Array representation in PDG’s instance dependence.](image1)

![Fig. 6 - Array representation in PDG array dependence.](image2)

![Fig. 7 - EPDG construction modules.](image3)
The time complexity of the EPDG construction is \( O(n^2) \), where \( n \) is the number of statements in the program. This is consistent and even better than some of the PDG algorithms discussed in the literature (Hammer and Snelting, 2009) that require \( O(n^3) \).

- Module for “EPDG traversal and report generator”: The implementation of this module is independent of any grammar and parsing mechanism, and can be used as a stand-alone module for any system whose EPDG is available. This module is responsible for traversing the EPDG from certain sensitive nodes using Extended BFS, and then checking whether an insecure flow does actually exist.

### 3.3. PDG and DB synchronization

This phase in the web application analysis consists of making sure that the database attributes are being propagated and captured by the PDG. Following the request initiated by application clients, the web application constructs SQL queries that will eventually be sent to the database for execution. The database then executes these queries, and returns a 2D array containing the results. In our work, we modify the construction of the PDG to handle these SQL queries along with their resulting 2D arrays (query results). The syntax of the SQL query retrieving data from the database is as follows:

```sql
SELECT attribute_{1}, attribute_{2}, ... FROM Tables WHERE condition
```

For each attribute in the query, we add to the PDG a node to represent it. The security class of this PDG node will be the same as that of the attribute. We also add a node entitled “Row”, that is data dependent on all the attributes involved in this query. The security class of the node entitled “Row” is set to the least upper bound of the security class of all nodes it is data dependent on. Finally, the resulting 2D array is set as the parent of the node “Row” and all the other attributes involved. In general, a query like the one above will be represented as shown in Fig. 8.

### 3.4. Search engine

The search engine is responsible for performing the application analysis by traversing the EPDG using Extended BFS and ensuring that none of the nodes violate the security policy of its security class per the database annotation. The identification of such violations is based on rules that are expected to govern the secure information flows. After thorough examination of web applications, we defined 6 different rules and attached each of the security classes to the appropriate rule. Under the proposed scheme, the rules are applied to the EPDG starting at the Row nodes of the resulting SQL queries.

**Rule 1.** A Row node is assigned the least upper bound of the security classes of the nodes it is data dependent on.

The Row node in the EPDG resulting from SQL queries is where our rules start to propagate in order to detect if any insecure flows actually exist. The Row node, as we have mentioned in our EPDG construction, is the parent of all the attributes involved in the query. Each of the attributes involved in the query might have a different security class, and the Row node is data dependent on all of these attributes. As a safety precaution, we assign the Row node the least upper bound of the security classes of the nodes it is data dependent on. Formally, the security class of the Row node is:

\[
S(\text{Row}) = \bigcup_{x \in \text{Attribute Nodes}} S(x)
\]

The way we have assigned the security class of the row node is a bit conservative favoring security over accuracy.

**Rule 2.** The security class of an output node is by default set to Public.

Output nodes are nodes, which are capable of sending server-side data to the client. They can vary (in PHP) from print(), print_r(), echo() and all other statements that print to the client’s browser. The security class of these nodes is automatically set to Public.

**Rule 3.** In the EPDG, if a path exists from a node labeled Top Secret to an output node labeled Public, it is reported as a violation.

Data with a security class Secret, as mentioned earlier, can never propagate to output. In the EPDG, a path \( x \rightarrow y \) means that information can flow from \( x \) to \( y \); the absence of this path is a guarantee that information cannot flow from \( x \) to \( y \) (Christian and Snelting, 2009; Reps and Yang, 1988; Wasserrab et al., 2009). Fig. 9 illustrates how a violation according to Rule 3 can be caught, i.e. a data path exists from node Row that is labeled Top Secret to an output node that is labeled Public.

**Rule 4.** In the EPDG, if a node labeled Single Disclosure is involved in any loop structure, it is reported as a violation.
Data labeled Single Disclosure are data that can only have one tuple sent to output. As a result, having it inside a loop structure, means that the Row is being fetched more than once, and therefore, more than one tuple might be sent to output. Fig. 10 illustrates how a violation according to Rule 4 can be caught.

Rule 5. In the EPDG, if a node labeled Masking Needed has a direct, control or data dependent path to an output node labeled Public, it will be reported as a violation.

Data labeled as Masking Needed can only be sent to output provided that it is masked or encrypted. A direct data or control edge from a node labeled Masking Needed to output means that the data has not passed through any other statement, meaning that it is being sent to output without masking. Fig. 11 illustrates how a violation according to Rule 5 can be caught.

Rule 6. If an aggregate function has more than one attribute, the same procedures and analysis of that Row node apply.

Although most aggregate functions return a single value, like MAX, MIN, AVERAGE, SUM and others, the same rules mentioned above apply to aggregate functions. If an aggregate function has more than one attribute involved, the same procedure as that of determining the security class of the Row node is performed, and the same security analysis follows. If the attributes involved are Top Secret, Single Disclosure, and Masking Needed, Rules 1, 2, and 3 apply respectively.

It is worth noting that the security labels can be appended with more security labels and the rules applied to the EPDG can be appended with more rules. Our approach is fully capable of accommodating these additions and with minimal changes. Assume that a new label called “Secret” is to be added to the existing security labels. Initially, certain database attributes will be annotated with the label Secret. As for the EPDG construction or the EPDG traversal, nothing at all will change. The only needed change is in the rules that need to be added to incorporate the violations that might be produced when the label Secret is involved in the information flow. The new rule(s) will be something very close to Rules 1 to 6 above. Consequently, our approach is easily extendable.

3.5. Report generator

This is the final component of our proposed model, which is responsible for generating a summary report about the line numbers of the code statements causing the security violations. In the following section, we give more insight about our implementation, then we present 2 simple case studies to validate the accuracy of the proposed model.

4. System validation

In order to validate our model, we run our proposed algorithm on 2 case studies we designed ourselves. This step is important to assess the accuracy and effectiveness of our model in capturing the security violations within web applications.

4.1. Case study 1: hotel reservation system

In a hypothetical hotel reservation system, the confidentiality policy states that only one available room can be disclosed to the client based on his/her preference. We developed this application with PHP being the server-side scripting language. The database tables used in this application are the Hotels and Rooms tables (Fig. 12). “RoomNumber” attribute in the database is the only one annotated as Single Disclosure, hence allowing for only one room to propagate to output. The rest of the attributes are public. Figs. 13 and 14 capture a snap-
shot of the code and its corresponding portion of the EPDG, respectively.

The variable \$available in line 11 of Fig. 13 is an array that is being filled with all the available rooms the query has returned. The while loop in line 12 repeatedly fetches data from the Row node of the query result (line 10), and pushes this data into the variable \$available in line 13. As a result, the variable \$available now contains all the available rooms based on the user’s preferences. Line 17 selects an entry from \$available at random, which is then displayed to the user. Now consider the case where the user refreshes the page that displays the available room. The code will typically run again when the refresh button is pressed. Running this application more than once may output more than one available room, thus violating the application’s confidentiality policy.

Now if we run our tool on the EPDG in Fig. 14, our tool can highlight the fact that the database attribute “RoomNumber” has a security class Single Disclosure, and thus, the Row node (line 10) which is data dependent upon this attribute, cannot be involved in a loop structure in the EPDG. The portion of the EPDG in which the insecure information flow occurred is highlighted in Fig. 15. Although the problem is that a random entry is being selected from the array, and then being sent to output, unfortunately, this is not the source of the problem. The source of the problem comes from the fact that the application code fetches more than one tuple of an attribute that contains Single Disclosure data.

The detected cycle between nodes 10 and 12 from Fig. 15 can be broken by simply inserting a break statement right after line 13 of the source code. The new code and corresponding EPDG are captured in Figs. 16 and 17, respectively. Notice that the Row node is not involved in a loop structure anymore, and thus, the array \$available at line 11 will only contain one entry, which will always be displayed after the call of the random function. This alternate code may not be the best implementation in terms of room assignment randomness, but it insures that the Single Disclosure variable (RoomNumber) is not being revealed.

4.2. Case study 2: auction web application

In most auction web applications, the confidentiality policy states that a reserved price of an item, which is the minimum price at which the seller is prepared to sell his item, can never be displayed. As a result, this attribute in the database should be annotated as Top Secret. Another confidentiality policy states that the user names of bidders cannot be sent to output as they are. Consequently, the database attribute containing user names in the bids table has to be labeled Masking Needed. Fig. 18 sketches the database tables for the auction web application (Bids, Auctions, Sellers, and Products). Figs. 19 and 20 show the security critical code section of the Auction application and the corresponding EPDG, respectively.

The path highlighted in red in the EPDG (Fig. 20) is a path that causes the insecure flow as it violates Rule 5 of Section
3.4. This rule prevents data annotated as Masking Needed to be outputted without being masked or encrypted. The two case studies just presented highlight the potential of the proposed system and its accuracy in detecting insecure flows. In the next section, we present our system performance in terms of its ability to catch illegal information flows as well as the false alarm rate on more complex and real PHP application.

Fig. 16 – Alternative secure implementation for the hotel reservation system application.

Fig. 17 – EPDG for the alternative secure implementation depicted in Fig. 16.

Fig. 18 – Database tables for the auction web application and the corresponding attribute annotations.

5. System performance

5.1. Experiment setup

We compare the results of our approach to JLift (Sabelfeld and Myers, 2003), which we consider to be the state-of-the-art in detecting insecure information flows. To make the testing as comprehensive as possible, we analyzed 9 applications (downloaded from SourceForge and GitHub) that vary in size and can be categorized as follows: 1 Medical Application, 1 banking application, 2 auction applications, 2 polling applications, 2 seat reservation applications, and 1 conference management system (CMS) application called HotCRP (Kohler, 2008); HotCRP is a very well known and open source CMS written in PHP, and is composed of 72 files with 40,339 lines of code. With the exception of the HotCRP application, we have manually counted the illegal information flows in the studied applications to get the ground truth data, and then ran our approach and JLift on these applications to detect how many insecure flows did each method capture. For each approach, we capture the ratio of false alarm with respect to the total number of alarms reported. Next, we present a detailed analysis of the Bank Controller and the medical applications, followed by a summary of all the applications.
5.2. Analysis of “bank controller”

The bank controller application simulates a banking system web application, where users can access their accounts, check expenses and other information about their bank account. The part of the application we have monitored covers the part of the application where the user wants to check his expenses. The database attributes related to this part of the application are mainly five attributes: uid, expenses_id, expenses_am, date_issued, closing_date, annotated as shown in Table 2. Under this annotation, expenses_id and expenses_am attributes are annotated with the annotation Single Disclosure (SD), meaning that only one tuple carrying any of these attributes can be released to output. The code uses an SQL query to extract the sum of all the expenses for a user and orders them by uid.

For testing using type-based systems, we have annotated the variable holding the query result with Confidential, and traced JLift’s algorithm to see how many flows it can catch. Fig. 21 shows the code segment being analyzed (commented code represents JLift annotations).

| Table 2 – Database skeleton and annotations for bank controller. |
|---|---|---|---|---|
| Uid | Expenses_id | Expenses_am | Date_issued | Closing_date |
| P | SD | SD | P | P |
Fig. 22 captures the EPDG corresponding to the bank controller code segment from Fig. 21. The results show that both JLift and our approach catch an insecure flow from Single Disclosure to public, but our approach catches the insecure flow as soon as the variable query (line 17) is assigned to the variable expenses (line 18), and this is where all the tuples from the query result are placed in the array "expenses". JLift, on the other hand, also catches an insecure flow. However, it assumes that the insecure flow is generated at line 28, which is the wrong position of the leak causing confusion from the developer’s perspective.

5.3. Analysis of “hospital patient records application – OpenEMR”

This is a patient record management application. In such an application, medical records are stored for each patient. Doctors can update patient information based on test results and check previous records. Patients, on the other hand, can check their status and some of their results. For safety and medical reasons, some results should not be seen by patients. The database attributes corresponding to this application along with the security annotations are shown in Table 3, where TS, P, SD, MN refer to classes Top Secret, Public, Single Disclosure, and Masking Needed, respectively.

The abnormal attribute cannot be at all seen by any client, and should only be seen by doctors, and thus for the client, it is annotated as Top Secret, meaning that it can never propagate to output. The result attribute can only be seen by the client himself, and thus it is marked as Single Disclosure, so the client can only check 1 result exactly (his own result). The result_status is related to the result so the same security class is appended to it (Single Disclosure). The range attribute is the range of normal results, which is usually public information that can be accessed by anyone. The client name is not security critical in this application, but it might be in other applications, so we can decide on it later. Fig. 23 shows the core code segment we analyzed in openEMR. The PDG for openEMR is presented in Fig. 24.

JLift annotations are within the source code. We can see that JLift raises an alarm on line 75, where an assignment from Top Secret to Public is detected. Our approach, on the other hand, assigns the query result the annotation “Top Secret” since it contains a Top Secret attribute, and then reports all the paths from node 61 to output as illegal. Some of these paths are false alarms, resulting from the fact that the developer is projecting the data to be sent to output over some of the attributes labeled public like range for example.

5.4. Results summary

Table 4 captures a summary of running all the gathered PHP applications including the ones discussed in the previous sections. Results clearly show that our approach outperforms JLift in terms of the count of false alarms being reported and its ability in catching the illegal flows. The table shows that at all times, our approach was able to catch all insecure flows (and at their origin), while JLift misses many illegal flows (highlighted in bold–italic font in the table). The biggest advantage of our approach over JLift is the number of reported false alarms (shaded in gray in the table). JLift produces a lot more false alarms than our approach, which rules it inefficient for detecting insecure information flows in web applications.

6. Discussion on the choice of labels and rules

We do realize that the suggested approach might not be comprehensive yet in the sense that a web application might emerge that requires more than 4 labels and more than 6 rules. However, we confirm that extending these labels and rules is an easy task as will be discussed below. We also assert that
the choice of the 4 labels and the corresponding 6 rules was the result of many experiments. We will proceed by discussing two points: (1) how did we come up with the 4 labels and 6 rules, and (2) how can the labels and rules be extended?

How did we come up with the 4 labels and 6 rules?

We started by gathering web applications from SourceForge and GitHub (100 to 1000 lines of code, mainly PHP). We then asked two web developers to manually analyze the code to identify the existing information leaks. We considered the manually annotated leaks to be our ground truth data, which we based our analysis and testing on. For data to be useful we knew that the confidentiality of the data should vary between public and top secret, i.e. you cannot release all data and at the same time you cannot hide it all. We started looking for these

![Fig. 23 – OpenEMR code segment.](image)

![Fig. 24 – PDG for openEMR.](image)
intermediate levels while inspecting the applications we collected. We discovered what we are now calling “single disclosure” and “masking needed”, bringing the total suggested security levels to four. We then acquired more web applications from SourceForge and GitHub, and analyzed the data just to confirm that no more than only 4 labels are needed. We did not base the selection of the 4 labels on any theory, as there is no one theory that governs data confidentiality classification. After defining the labels, and deciding that we will adopt the EPDG to check for the leaks, the formation of the rules followed. Creating the rules follows directly from the definition of the labels. For instance, a datum that is labeled “Masking Needed” should not be outputted without masking; the rule follows directly from the definition as follows: “In the EPDG, if a node labeled Masking Needed has a direct, control or data dependent path to an output node labeled Public, it will be reported as a violation.” Now what if more labels and more rules are needed. This brings us to the next question.

How can the labels and rules be extended?

Let us assume that our approach was only composed of three labels as follows: Top Secret, Single Disclosure, and Public. The corresponding rules will be (as indicated in Section 3.4):

- Rule 1. A Row node is assigned the least upper bound of the security classes of the nodes it is data dependent on.
- Rule 2. The security class of an output node is by default set to Public.
- Rule 3. In the EPDG, if a path exists from a node labeled Top Secret to an output node labeled Public, it is reported as a violation.
- Rule 4. In the EPDG, if a node labeled Single Disclosure is involved in any loop structure, it is reported as a violation.
- Rule 5. If an aggregate function has more than one attribute, the same procedures and analysis of that of the Row node applies.

Now let us say that the developer found the need to add another security label where secret information can be outputted, but should be masked before that. The developer creates a new label called ”Masking Needed” and then defines the rules on the EPDG as follows: “if a node labeled Masking Needed has a direct, control or data dependent path to an output node labeled Public, it will be reported as a violation.” Similarly any new label and the corresponding rule can be added.

<table>
<thead>
<tr>
<th>Application</th>
<th># of illegal flows</th>
<th>Our approach</th>
<th>JLift</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total alarms</td>
<td>Illegal flows</td>
</tr>
<tr>
<td>OpenEMR</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Bank controller</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Auction 1</td>
<td>2</td>
<td>2o</td>
<td>2</td>
</tr>
<tr>
<td>Auction 2</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>ITL poll – user login</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ITL poll – polling section</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Hotel reservation</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>mySeat reservation</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>HotCRP (Kohler, 2008)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

7. Conclusion and future work

In this work, we propose a framework for securing the information flow in web applications. The system checks whether the confidentiality and integrity policies of web applications are violated. Contrary to existing systems, where program variables or database tuples/queries are labeled for security, we associate security annotations with database attributes then propagate them through the program code via the traversal of an extended form of the application dependence graph (PDG). The paths in the extended PDG (EPDG) are then checked against six predefined rules to determine if a violation to the security policies has occurred. The proposed framework was validated using 2 customized test cases, a Hotel Reservation System and an Auction Web Application. We also ran our system on more complicated applications (downloaded from SourceForge and GitHub) and showed the high success rate of our system in terms of catching the security flaws at their source and in terms of reducing the reported false alarms, when compared to JLi, a state-of-the-art type-based system approach to detect information leaks. Currently, we are in the process of working on the next release of our system, with more improvement in the “Grammar Construction and Parsing” module to capture all PHP constructs.

References
