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. Unfortunately 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, E-bay, 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 [1] 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 [1], 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 [2], where a study was conducted to quantify the impact of information leaks in web application. 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 [2,3]. 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 the construction phase.

In a typical example, consider the case of an online Reservation System (for instance airplane seat reservation system), where for security reasons the user is not allowed to specifically choose his/her seat. The system should rather ask for the user’s preference then automatically reserve the seat accordingly without revealing the exact location of the seat. This set up is highly recommended in systems where group of users are not allowed to get a full picture of the current status of the system (airplane for example) and therefore prevent them from coordinating a pattern that could result in malicious act. Therefore, a confidentiality policy should be enforced by the system developer; this is exactly what our system aims to do. In the above example, the policy should say “no client can see all the available seats”.

---

**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. Moreover and to highlight the simplicity of the suggested approaches vs. existing approaches, two professional web developers assessed the annotation tasks needed in the presented case studies and provided a very positive feedback on the simplicity of the annotation task.

**Keywords**—Web Applications Security, Secure Information Flow, Program Dependence Graph, Database Annotation.
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. Contrarily to existing systems, where program variables are labeled for security, we associate security annotations with database attributes then propagate them through the application (program) code via the traversal of an extended from of the program’s dependence graph (PDG). We call the 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.

For testing, we consider 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 II, we present a classification of existing approaches for web application security analysis. In Section III, we discuss the details of the proposed framework. In Section IV, we present the results of running our system on two test cases for validation purposes. In Section V, we conclude the paper and present the future directions.

II. 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 based [1, 4, 5, 6, 7, 8, 24], and (2) Non-Information control based [9, 10, 11, 12]. The former can further be classified into two sub-classes: (1) Type-based system [1, 7], and (2) Program Dependence Graph (PDG) based systems [4]. The latter category of web application security (Non-Information control based) attempts to secure applications against attacks, such as cross-site scripting and SQL Injections [13, 14]. Our proposed approach falls under Program Dependence Graph based approach. We give a brief synopsis of prominent examples of each of these approaches.

A. Information Flow Control Based approach

The problem of secure information flow was first introduced by the Dennings in the mid-seventies [15, 16]. It is divided into Type-based systems and PDG-based systems. Typed based Systems works by having developers annotate application code and statements [1, 7, 17, 24]. Developers are required to append a security class for variables or code statements by marking the appropriate annotation at the beginning of each statement, which puts a lot of burden on the developers. Another drawback of this approach is the high rate of false alarms as reported in [18]. Swift [1], SIF [24], Flow Calm [25], JIF [26], and are yet other typed based flow control systems moving 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 [1] and SIF [24] were built using Jif (Java based typed programming language [26]) 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).

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 [19] and [20]. PDG is considered to be a powerful tool to perform application security analysis [4]. The approach we propose in this research work is a major improvement to the technique discussed in [4]. For comparison reasons, we have designed an application “hotel reservation system” which will be used later to validate our technique and eventually highlight the weaknesses of the approach proposed in [4] as well as the effectiveness of ours. In this application, we define the following confidentiality policy “only one available room in the hotel can be presented to the client” (this policy was defined by following the same thinking logic used when defining the policy of the “reservation system” discussed in the introduction). The fragment of code for the reservation system and its corresponding PDG are shown in Figures 1 and 2 respectively.

```
1 if ($query = "SELECT roomnumber,available FROM room WHERE (hotel = "hotel1" AND roomtype="roomtype")");
2 $result = mysql_query($query);
3 if (mysql_num_rows($result) == 0) { $available = array();
4 while ($row = mysql_fetch_array($result)) {
5     array_push($available, $row["roomnumber"]);
6 }
7 if (!$available) {
8     print "No room available or randomly selected room reserved";
9 } else {
10     print "No room available or randomly selected room reserved";
11 }
```

Figure 1. Hotel Reservation System Code Fragment

![Figure 2. Hotel Reservation System PDG](image)

In a PDG, a dotted arrow represents data dependence between two nodes, while a normal arrow represents control dependence. Following the approach described in [4], 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 *lenth()* or *empty()*, and others are only control or data dependent on an instance, such as *available[i]*. Following the rules presented in [4] a violation is detected in the hotel reservation system on lines 18 and 21.

Despite the fact that the method described in [4] 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 source of the information leak is not detected using the method presented in [4]. 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. Our proposed approach is different from that presented in [4] 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. In the next section, we briefly discuss other non-flow control based approaches.

**B. Non-Information Flow Control Based approach**

Under this class of approaches [9-12], 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, and consequently insert runtime guards to achieve a secure flow. We discuss next the details of our framework design for insecure information flow detection.

**III. FRAMEWORK DESIGN AND IMPLEMENTATION**

**A. Framework Architecture**

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 application code, and (3) it enforces the application confidentiality by propagating the security policies from the database attributes to the corresponding PDG. The proposed framework consists of 4 major blocks as depicted in Figure 3. Each block is discussed hereafter.

1) **Database (DB) Annotation**

Contrarily to existing approaches, where the focus was on protecting application code, we propose protecting application data 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) Secret, (3) Confidential, and (4) Public. Data in 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 is assigned to “Secret” class. It is worth mentioning that this classification is in-line with how organization (mainly in the defense side) internally classifies information. Next, we provide more details about each of the 4 proposed attribute classes.

- **Top Secret**: Data classified as *Top Secret* is 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.

- **Secret**: Data classified as *Secret* is data that is 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 *Secret* can return a result array that contains several items, among which is the data labeled as *Secret*. 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 a 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 by *Secret* ensures that only one tuple of the table (i.e. one available room) can flow to the output.

- **Confidential**: Data classified as *Confidential* is data that can be propagated to output with no restriction on the number of tuples, but rather with the restriction that this data has to be masked or encrypted before it is sent to output. A query to the database that involves data labeled as *Confidential* 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 confidential data is the name of bidders in auctions (E-buy 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.

2) **Program Dependence Graph (PDG) Construction**

PDG construction is a critical component in our system. It has to be correct, efficient, and should incorporate the database annotations and queries. PDG is known to be an intermediate representation of the program. In our work, it is 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 it also aids us in tracking what went through to output. A detailed description on how the PDG can be constructed for web applications is shown in [19]. We used the same method and applied some modifications to it in order to reflect the annotations placed on database attributes; we call the new representation, 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 [4], differentiates between an instance of the array and the array itself.

![Figure 4. Array Representation in PDG's Instance Dependence](image)

Figure 4. Array Representation in PDG's Instance Dependence

Figure 4 presents 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 Figure 5.

![Figure 5. Array Representation in PDG Array Dependence](image)

Figure 5. Array Representation in PDG Array Dependence

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. This will further be discussed in the next section as well as in the section III-B, where we provide more implementation details related to the EPDG construction.

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. The process is discussed next.

Following the request initiated by application clients, the web application constructs SQL queries which 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 that retrieves data from the database is as follows:

```sql
SELECT attribute1, attribute2, ... 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 be the least upper bound of the security class of all the nodes it is data dependent upon. 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 Figure 6.

![Figure 6. Query Representation in the EPDG](image)

4) **Search Engine**

The search engine is responsible for performing the application analysis by traversing the EPDG and ensuring that none of the nodes violate the security policy of its security class as per the database annotation. The identification of such violations is based on rules that are expected to govern the secure information flows. After a 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(), printf(), 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 \( y \) [4, 21, 22]. Figure 7 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.

![Figure 7. Insecure Information Flow based on Rule 3](image)

Rule 4. In the EPDG, if a node labeled Secret is involved in any loop structure, it is reported as a violation.  
Data labeled Secret is 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. Figure 8 illustrates how a violation according to Rule 4 can be caught.

![Figure 8. Insecure Information Flow based on Rule 4](image)

Rule 5. In the EPDG, if a node labeled Confidential has a direct, control or data dependent path to an output node labeled Public, it will be reported as a violation.  
Data labeled as Confidential can only be sent to output provided that it is masked or encrypted. A direct data or control edge from a node labeled Confidential to output means that the data hasn’t passed through any other statement, meaning that it is being sent to output without masking. Figure 9 illustrates how a violation according to Rule 5 can be caught.

![Figure 9. Insecure Information Flow based on Rule 5](image)

Rule 6. 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, Secret, and Confidential Rules 1, 2, and 3 apply respectively.

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.

B. Framework Implementation  
The implementation of our model consists of 4 major modules, each module feeding the other (Figure 10).

![Figure 10. System Modules](image)

B.1 Module for “Grammar construction and Parsing”  
Because of its popularity and flexibility, we decided to build, using ANTLR [23], the front-end compiler for PHP allowing us to parse PHP web applications. 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.

B.2 Module for “EPDG node extraction”  
This is the first step towards 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’s left is to
connect these nodes via edges representing data and control dependence.

B.3 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 [24], 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 dependencies 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. 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 literature.

B.4 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, and then checking whether an insecure flow does actually exist based on the rules listed in section III-A.4.

IV. 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.

A. Case Study 1: Hotel Reservation System

In a hotel reservation system, and for reasons we have mentioned earlier, 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 (Figure 11). “RoomNumber” attribute in the database is the only one annotated as Secret, hence allowing for only one room to propagate to output. The rest of the attributes are public. Figures 12 and 13 capture a snapshot of the code and its corresponding portion of the EPDG, respectively.

![Figure 11. Database tables in the Hotel Reservation System and the attribute annotations](image1)

![Figure 12. Code Fragment from the Hotel Reservation System application](image2)

The variable $available$ in line 11 of Figure 12 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 Figure 13, our tool can highlight the fact that the database attribute “RoomNumber” has a security class Secret, 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 Figure 14. 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 Secret data.

The detected cycle between nodes 10 and 12 from Figure 14 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 Figures 15 and 16, respectively.

Notice that the Row node is not involved in a loop structure anymore, and thus, the array Savailable 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 secret variable (RoomNumber) not being revealed.

B. 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 Confidential. Figure 17 sketches the database tables for the auction web application (Bids, Auctions, Sellers, and Products). Figures 18 and 19 show the security critical code section of the Auction application and the corresponding EPDG, respectively.

The path highlighted in red in the EPDG (Figure 19) is a path that causes the insecure flow as it violates Rule 5 of section III-A.4. This rule prevents data annotated as confidential 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.

We presented in this paper an initial prototype of a system that can secure web applications by only requiring the
developer to annotate database attributes based on the security policies of the concerned website. Two developers were introduced to the new system and asked to analyze the two case studies presented above. Their feedback was very positive in the sense that the annotation task was very straightforward and directly extracted from the security policy. Such feedback validates one of our contributions in this paper, which is simplicity. More work need to be done for this system to become deployed: (1) Rigorous testing should be performed; (2) the proposed approach should be compared to previous approaches; (3) the simplicity of using this approach should be compared to the simplicity of using other approaches; (4) and finally special code should be incorporated in PHP or the database to enable the developer to provide the appropriate annotations.

V. 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. Contrarily to existing systems, where program variables 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 (program) 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. Results show that the proposed model has a high success rate in terms of catching the security flaws at their source. Moreover, two developers assessed the annotation tasks needed in the presented case studies and provided a very positive feedback on the simplicity of the annotation task. For future work, we aim to rigorously test the proposed approach on big applications, compare to other works, and incorporate the database annotation in PHP grammar.

REFERENCES