SWAN$_{\text{ASSIST}}$: Semi-Automated Detection of Code-Specific, Security-Relevant Methods

Goran Piskachev*, Lisa Nguyen Quang Do†, Oshando Johnson‡ and Eric Bodden§
Fraunhofer IEM*‡§, Paderborn University†§
Email: *goran.piskachev@iem.fraunhofer.de, †lisa.nguyen@upb.de, ‡oshando.johnson@iem.fraunhofer.de, §eric.bodden@upb.de

***Abstract*—To detect specific types of bugs and vulnerabilities, static analysis tools must be correctly configured with *security-relevant methods* (SRM), e.g., sources, sinks, sanitizers and authentication methods—usually a very labour-intensive and error-prone process. This work presents the semi-automated tool SWAN$_{\text{ASSIST}}$, which aids the configuration with an IntelliJ plugin based on active machine learning. It integrates our novel automated machine-learning approach SWAN, which identifies and classifies Java SRM. SWAN$_{\text{ASSIST}}$ further integrates user feedback through iterative learning. SWAN$_{\text{ASSIST}}$ aids developers by asking them to classify at each point in time exactly those methods whose classification best impact the classification result. Our experiments show that SWAN$_{\text{ASSIST}}$ classifies SRM with a high precision, and requires a relatively low effort from the user. A video demo of SWAN$_{\text{ASSIST}}$ can be found at *https://youtu.be/fSyD3V6EQOY*. The source code is available at *https://github.com/secure-software-engineering/swan*.**

***Index Terms*—Program Analysis, Machine-learning, Security**

I. Introduction

More and more companies use static analysis to detect security vulnerabilities in their code, configuring them for various types of bugs and vulnerabilities such as the SANS top 25.[1] For such analyses to be as effective as possible, they must be adapted to the codebase. One particular challenge is to provide the analyses with the correct *security-relevant methods* (SRM): sources, sinks, etc.

Lists of SRM are generally created manually by security experts. As a result, they tend to be incomplete, causing an analysis to miss vulnerabilities, or to signal false positives. For instance, Arzt et al. show that, static analysis tools frequently miss *a large majority* of relevant findings due to insufficient configurations [1]. They present SUSI, an automated machine-learning approach for the detection of two types of SRM (sources and sinks) in the Android framework. In later work, Sas et al. [2] extend SUSI to detect sources and sinks to general Java programs. Both approaches are run ahead of time, before the analysis is deployed.

However, both approaches are specific to one framework: the Android framework for SUSI and Java runtime for Sas et al.'s approach. SRM contained in codebases that use specific libraries, code constructs, or custom SRM are likely to be missed. In this paper, we present SWAN$_{\text{ASSIST}}$, an IntelliJ plugin that works on top of SWAN (Security methods for WeAkNess detection), a machine-learning approach that classifies methods into SRM types (sources, sinks, sanitizers, or authenthication methods) and specific CWEs.[2] SWAN$_{\text{ASSIST}}$ allows users to actively feed new SRM from their code by labeling methods, thus adapting the classification to their own codebase. In addition, its recommender system, *SuggestSWAN*, proposes methods for classification by the user that are more likely to have the strongest impact on the classification. SWAN$_{\text{ASSIST}}$ can be used by developers at coding or debugging time, or by security teams to configure analysis tools before they are deployed.

We show that SWAN$_{\text{ASSIST}}$ can improve SWAN's base precision with minimal user effort in labeling methods.

```
1   protected void doGet(HttpServletRequest
        request, HttpServletResponse response)
        throws ServletException, IOException {
2     try {
3       String userId =
            request.getParameter('userId');
4       userId = ESAPI.encoder().encodeForSQL(new
            MySQLCodec(), userId);
5       Statement st = conn.createStatement();
6       String query = "SELECT * FROM  User WHERE
            userId='" + userId + "';";
7       ResultSet res = st.executeQuery(query);
8       String url = "https://" +userId+
            ".company.com";
9       response.sendRedirect(url);
10    } catch (Exception e) { ... }
11  }
```

Listing 1: Potential SQL injection (from l.3 to l.7) and open redirect (from l.3 to l.9).

II. Discovering SRM

Listing 1 contains a potential SQL injection (*CWE-89*) from line 3 to line 7, and an open redirect (line 9) (*CWE-601*). The SQL injection vulnerability is mitigated using the validator line 4, which sanitizes the user-controlled input.

To find the two potential vulnerabilities, a taint analysis can be used with specific SRM. Here, the *source* `getParameter()` creates the data to be tracked, the *sinks* `executeQuery()` and `sendRedirect()` raise the alarm, and the *validator* `encodeForSQL()`, marks the data as safe,

[1]http://cwe.mitre.org/top25/

[2]http://cwe.mitre.org/

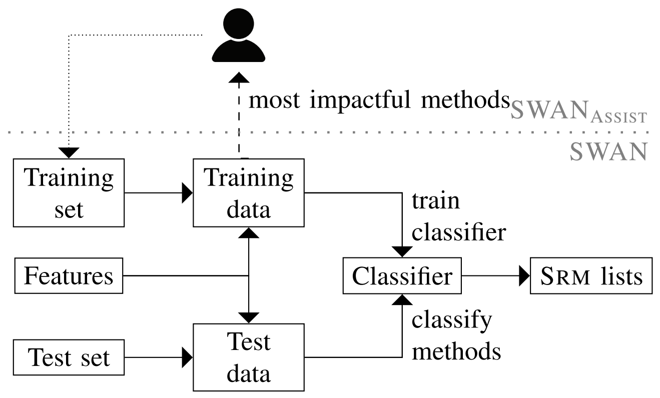


Figure 1: Machine-learning approach used in SWAN (solid edges), and SWAN$_{\text{Assist}}$ with developer feedback (angled dotted edge), and the recommender system *SuggestSWAN* (straight dashed edge).

but only for the SQL injection. To support the detection of the SANS top 25 vulnerabilities, we identify five requirements:

- ***R1**: SRM should differentiate between sources, sinks, validators and authentication methods*: In Java, the SANS 25 can be detected using sources, sinks, and sanitizers, that are required to configure data-flow analyses (e.g., in Listing 1). Authentication methods are required for specific CWEs such as *CWE-306* (Missing Authentication for Critical Function).
- ***R2**: SRM lists should be specific to each CWE*: As seen in Listing 1, different SRM should be used for different CWEs.
- ***R3**: SRM lists should be specific to the code base*: Since different applications can use different libraries, or define their own custom methods, not all SRM can be defined in advance. They should instead be derived from the code of the application or its libraries.
- ***R4**: The detection of SRM should be automated*: The Java Spring framework contains more than 30,000 methods. Real-life applications use many such dependencies. As a result, a fully manual definition of the SRM is not feasible.
- ***R5**: The detection of SRM should involve the user*: In different contexts, different methods can be used for different purposes. To best determine which methods are relevant SRM, it is important to allow the user to correct the set of SRM if the automated approach is insufficiently precise.

SUSI [1] and Sas et al. [2] only meet requirement **R4** and part of **R1**, because they report only sources and sinks.

III. SRM Classification With SWAN

Figure 1 illustrates the SWAN and SWAN$_{\text{Assist}}$ systems. SWAN runs the automated classification twice: in the first iteration, it classifies all methods of the analyzed program and libraries into general SRM classes (**R1**): *sources* (So), *sinks* (Si), *sanitizers* (Sa), *authentication methods*, or *none*. In the second iteration, it discards the methods marked with *none*, and classifies the remaining SRM into the individual CWEs (**R2**): *CWE-78*, *CWE-79*, *CWE-89*, *CWE-306*, *CWE-601*, *CWE-862*, and *CWE-863*, per SRM class.

SWAN uses 25 types of binary machine-learning features, instantiated into 206 concrete features. For example, the feature instance `methodClassContainsOAuth`, which is used to indicate an authentication method, is of type `methodClassContains`. Overall the features of SWAN are designed to address **R1**–**R2**, targeted in particular for the detection of sanitizers, authentication methods, and different types of CWEs according to the SANS 25 classification.

To obtain the feature matrix, SWAN uses the Soot [3] program analysis framework. As its machine-learning module, it uses WEKA's [4] SVM learner as it showed the best F-measure [5]. The training set contains 235 Java methods collected from 10 popular and diverse Java libraries (Spring, jsoup, Google Auth, Pebble, jguard, WebGoat, and four Apache frameworks), annotated with SRM types and CWEs. We selected methods to cover positive and negative examples for the features used for each SRM and CWE classification. SWAN accepts a Java program or library as its test set, and classifies its methods in the SRM types and CWEs.

SWAN is implemented as a standalone command line Java program with four parameters: a path to a directory containing the test dataset, a link to a Json file containing the signatures of the methods from the training sets, a path to a directory containing the source code implemenation of the methods listed in the Json file, and a path of a directory where the output files should be stored.

SWAN can be extended with new CWEs as follows:

- add a description of the CWE in the CWE index,
- create new feature instances specific to the CWE,
- match the feature instances to the classes (SRM types of CWEs), as shown in Listing 2,
- adjust the training set by (1) marking existing methods with the new CWE and (2) adding methods if necessary, to ensure that the training set contains at least one positive and one negative example per feature instance.

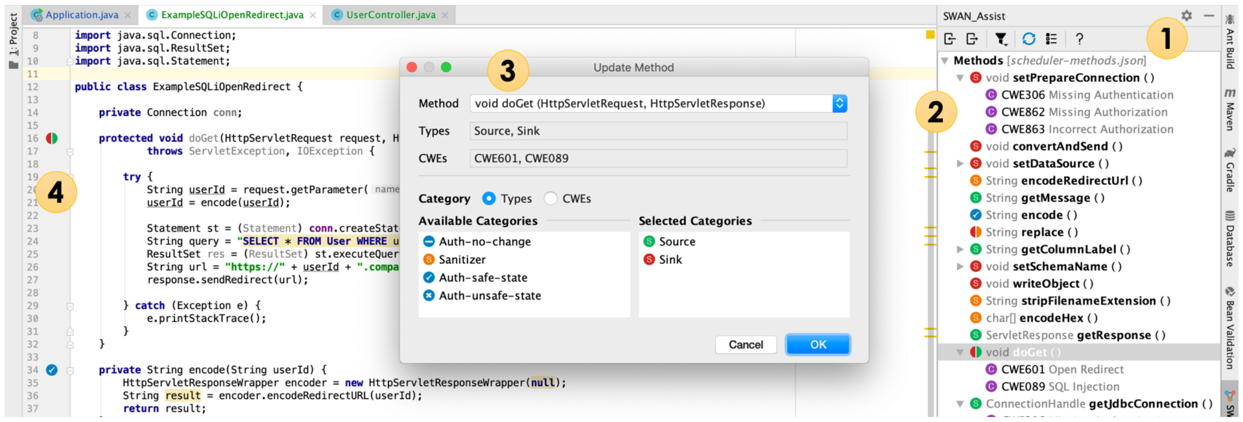
```
IFeature classNameContainsSql = new
    MethodClassContainsNameFeature("sql");
addFeature(classNameContainsSql, new
    HashSet<>(Arrays.asList(Category.SOURCE,
Category.SINK, Category.CWE089,
    Category.NONE)));
```

Listing 2: Matching a feature instance *classNameContainsSql* to the categories *Source*, *Sink*, *CWE-89*, and *None*.

Listing 2 shows a code example that creates the instance *MethodClassContainsSql*. The method *addFeature* matches the feature instance with the relevant classes *Source, Sink, CWE-89* and *None* in which the methods that match the feature instance can be classified.

IV. Active Learning With SWAN$_{\text{Assist}}$

To refine SWAN, SWAN$_{\text{Assist}}$ integrates developer feedback in order to adapt the learning algorithm to the code base under development (**R3** and **R5**). SWAN$_{\text{Assist}}$ allows the developer to edit the training set directly in their Integrated



```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;

public class ExampleSQLiOpenRedirect {

    private Connection conn;

    protected void doGet(HttpServletRequest request, H
            throws ServletException, IOException {

        try {
            String userId = request.getParameter( name
            userId = encode(userId);

            Statement st = (Statement) conn.createStat
            String query = "SELECT * FROM User WHERE u
            ResultSet res = (ResultSet) st.executeQuer
            String url = "https://" + userId + ".compa
            response.sendRedirect(url);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private String encode(String userId) {
        HttpServletResponseWrapper encoder = new HttpServletResponseWrapper(null);
        String result = encoder.encodeRedirectURL(userId);
        return result;
```

Update Method

Method: void doGet (HttpServletRequest, HttpServletResponse)

Types: Source, Sink

CWEs: CWE601, CWE089

Category: [✓ Types] CWEs

Available Categories	Selected Categories
Auth-no-change	Source
Sanitizer	Sink
Auth-safe-state	
Auth-unsafe-state	

Cancel OK

SWAN_Assist

- Methods [*scheduler-methods.json*]
  - void **setPrepareConnection** ( )
    - CWE306 Missing Authentication
    - CWE862 Missing Authorization
    - CWE863 Incorrect Authorization
  - void **convertAndSend** ( )
  - void **setDataSource** ( )
  - String **encodeRedirectUrl** ( )
  - String **getMessage** ( )
  - String **encode** ( )
  - String **replace** ( )
  - String **getColumnLabel** ( )
  - void **setSchemaName** ( )
  - void **writeObject** ( )
  - String **stripFilenameExtension** ( )
  - char[] **encodeHex** ( )
  - ServletResponse **getResponse** ( )
  - void **doGet** ( )
    - CWE601 Open Redirect
    - CWE089 SQL Injection
  - ConnectionHandle **getJdbcConnection** ( )

Figure 2: GUI of SWAN_{Assist}.

Development Environment (IDE), and includes this data in the training set for the next learning iteration, as shown in Figure 1. The developer can add or remove methods of the training set, or change the classes of a method.

In addition, SWAN_{Assist} generates a list of methods that—if classified differently—would yield the most impact on the next run of SWAN, based on the feature matrix, and proposes them to the user through a recommender system: *SuggestSWAN* (dashed edge in Figure 1). This identifies the most useful methods to the classification.

We have implemented SWAN_{Assist} as an IntelliJ plugin. It integrates into the development environment a GUI for editing the SRM lists and for executing SWAN, as shown in Figure 2.

SWAN's training set is shown on the rightmost view of the GUI ②, called the SWAN_Assist view. Methods in this view can be filtered by classification class or by file (button in ①). The pop-up dialog in the center ③ allows the developer to edit the training set. It is accessible through the SWAN_Assist view or through the context menu when a method in the code editor is selected. With this dialog, the developer can add or remove classes for the method. Methods can be added to the training set through the context menu, and removed through the context menu or using the SWAN_Assist view. SRM markers are also shown on the left side of the editor ④.

SWAN_{Assist} also allows the developer to re-run the classification by clicking on the icon in the toolbar of the SWAN_Assist view ①. This re-runs SWAN in the background, and updates the list of SRM (dotted edge in Figure 1). Methods that were just removed are shown in red, and can be returned to the training set by using the *restore* operation from the context menu. Otherwise, they are removed from the list on the next run.

V. Architecture

Figure 3 shows the components of SWAN_{Assist}, SWAN, and external components (shown in gray). There are three external components: *WEKA*, *Soot* and *JsonSimple*. *WEKA* is a machine-learning library used for the classification [4]. SWAN uses *Soot* to load the source code of the training and testing sets, and to evaluate the methods against the features [3]. *JsonSimple* is used by both SWAN and SWAN_{Assist} for manipulating the sets of methods serialized in Json format [6].

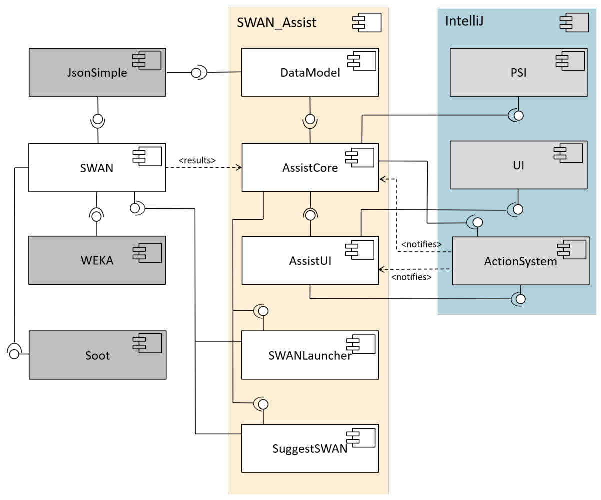


Figure 3: Architecture of the SWAN_{Assist} IntelliJ plugin.

MOISLauncher prepares the input parameters (training and testing sets) and calls SWAN in separated thread. Once the results are available, they are stored and *AssistCore* is notified. SWAN_{Assist} has its own model for methods, *DataModel*, which can be serialized in Json format using *JsonSimple*. This model contains the source code information for each methods and it maps the GUI elements, such as icons and labels.

The plugin SWAN_{Assist} uses three components of the IntelliJ framework: *ActionSystem*, *UI*, and *PSI*. *UI* provides the *Dialog* and *ToolWindow* user interface elements that *AssistUI* uses to display the SWAN results. These elements are populated by *AssistCore*. The main communication between the IntelliJ framework and the plugin is implemented through the *ActionSystem*, which is a listener/notifier design provided by IntelliJ. User actions on the GUI are handeled by this component. The *PSI* (Program Structure Interface) component provides information about the currently opened project. This is used by *AssistCore* to set the GUI filters, configure SWAN, and receive updates about changed classes or methods.

More details can be found in our publication [5].

VI. Evaluation

We evaluated the precision of SWAN on 12 open-source Java libraries: two frameworks from the mobile domain (Android and Apache Cordova), eight web frameworks (Apache Lucene, Apache Stratos, Apache Struts, Dropwizard, Eclipse Jetty, GWT, Spark, and Spring), one framework from the home automation domain (Eclipse SmartHome), and one utility framework (Apache Commons). We applied SWAN to the 12 libraries, and randomly selected 50 methods for each pair of library/class, whose classification we then manually verified.

SWAN yields an overall precision of 0.76. It is more precise for detecting SRM types (0.826) than for CWEs (0.677). Its best precision is of 0.99, for Android's sources, and worst, of 0.44, for GTW and *CWE-78*. SWAN's overall precision is consistent over different types of Java applications, but can be improved with SWAN$_{\text{ASSIST}}$.

To evaluate the efficiency of *SuggestSWAN* with respect to the manual work required by the user, we used the Gene Expression Atlas (GXA) [8] application, which yields a relatively low precision with the base SWAN. This allows us to showcase the potential of the active learning approach. As our ground truth, we manually classified all 1,638 methods of GXA, 286 of which were identified as sources. We compare the precision of the SRM lists obtained when adding to the training set randomly selected method pairs, against pairs obtained with *SuggestSWAN*. Figure 4 shows the precision, starting from the base SWAN until all 819 pairs are added to the training set. The random graph is averaged over 10 runs. We repeated the experiments also for sinks and methods related to *CWE-89*. Both show similar trend like the sources. Because of space limitation we omit these graphs.

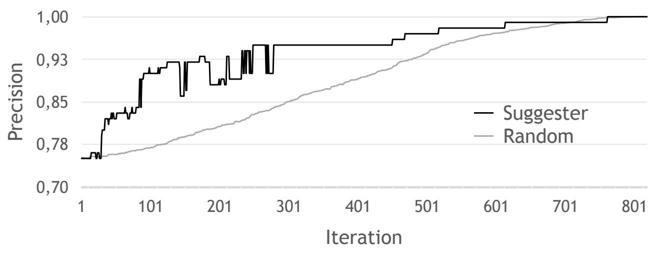


Figure 4: Precision of the sources over 819 iterations of SWAN on GXA by adding methods to the training set with *SuggestSWAN* and with a random selection of methods.

The evolution of the precision for the random recommender is linear, i.e. it does not help the classification. SWAN has a fast increase in precision at the beginning, showing that the recommender is efficient in selecting the methods with the most impact first. This maximizes the impact of the classification and minimizes the user's effort. The precision reaches 0.8 at iteration 31 (from 0.75 at iteration 1), with 60 labeled methods (4% of the total number of methods). Using *SuggestSWAN* on GXA, it yields a high precision significantly faster than with a random selection of methods.

VII. Related Work

SUSI is a machine-learning approach to detect sources and sinks in the Android framework. SWAN extends SUSI to be able to find sanitizers and authentication methods. SWAN sub-classifies the SRM into CWEs, unlike SUSI which sub-classifies its sources and sinks into Android-specific categories such as bluetooth, browser, etc. SWAN is applicable to general Java applications, including the Android framework, and its extension SWAN$_{\text{ASSIST}}$ is able to further adapt it to specific codebase. Sas et al. [2] generalize SUSI to Java applications, but do not further sub-classify in CWE classes, nor do they support sanitizers and authentication methods, or provide active learning functionalities.

JoanAudit [7] provides hand-crafted lists of sources, sinks, and validators specific to particular CWEs. However, the lists are not complete, and only contain SRM for targeted examples of specific libraries. SWAN is able to automatically detect SRM for any Java application and its libraries, with minimal manual work. The automated SRM extractor Merlin [9] uses probabilistic inference to detect string-based vulnerabilities for taint analyses. The approach used by SWAN can be extended to more types of vulnerabilities, on top of being able to adapt to the codebase with SWAN$_{\text{ASSIST}}$.

VIII. Conclusion

In this paper, we present the SRM detector SWAN and the SWAN$_{\text{ASSIST}}$ IntelliJ IDEA plugin, which help developers to create SRM lists specific to selected CWEs and specific codebases. We demonstrated tool features of SWAN and SWAN$_{\text{ASSIST}}$, and detailed their architecture, as well as the machine-learning model behind it. Our tool enables users to adapt the configuration of static analyses in semi-automatic way with relatively low effort. Using this tool, static analysis tools can adapt to individual projects, with minimal work and required knowledge from the developer.

References

[1] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks," ser. NDSS'13, 2013.

[2] D. Sas, M. Bessi, and F. A. Fontana, "Automatic detection of sources and sinks in arbitrary java libraries," ser. SCAM'18, 2018.

[3] S. Arzt, S. Rasthofer, and E. Bodden, "The soot-based toolchain for analyzing android apps," in *Proceedings of MOBILESoft '17*. Piscataway, NJ, USA: IEEE Press, 2017, pp. 13–24. [Online]. Available: https://doi.org/10.1109/MOBILESoft.2017.2

[4] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*, 2016.

[5] G. Piskachev, L. N. Q. Do, and E. Bodden, "Codebase-adaptive detection of security-relevant methods," ser. ISSTA 2019. NY, USA: ACM, 2019.

[6] "Jsonsimple," https://github.com/fangyidong/json-simple, Accessed in 2019.

[7] J. Thomé, L. K. Shar, D. Bianculli, and L. C. Briand, "Joanaudit: A tool for auditing common injection vulnerabilities," ser. ESEC/FSE'17, 2017.

[8] "Gene expression atlas," https://github.com/gxa/gxa, Accessed in 2019.

[9] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," *SIGPLAN Not.'09*, 2009.