Codebase-Adaptive Detection of Security-Relevant Methods

Goran Piskachev
Fraunhofer IEM
Germany
goran.piskachev@iem.fraunhofer.de

Lisa Nguyen Quang Do
Paderborn University
Germany
lisa.nguyen@upb.de

Eric Bodden
Paderborn University and Fraunhofer IEM
Germany
eric.bodden@upb.de

ABSTRACT

More and more companies use static analysis to perform regular code reviews to detect security vulnerabilities in their code, configuring them to detect various types of bugs and vulnerabilities such as the SANS top 25 or the OWASP top 10. For such analyses to be as precise as possible, they must be adapted to the code base they scan. The particular challenge we address in this paper is to provide analyses with the correct *security-relevant methods* (Srm): sources, sinks, etc. We present SWAN, a fully-automated machine-learning approach to detect sources, sinks, validators, and authentication methods for Java programs. SWAN further classifies the Srm into specific vulnerability classes of the SANS top 25. To further adapt the lists detected by SWAN to the code base and to improve its precision, we also introduce $\text{SWAN}_{\text{Assist}}$, an extension to SWAN that allows analysis users to refine the classifications. On twelve popular Java frameworks, SWAN achieves an average precision of 0.826, which is better or comparable to existing approaches. Our experiments show that $\text{SWAN}_{\text{Assist}}$ requires a relatively low effort from the developer to significantly improve its precision.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Computing methodologies** → *Machine learning approaches*; • **Software and its engineering** → *Software verification and validation*; • **Human-centered computing** → Interaction design process and methods.

KEYWORDS

Program Analysis, Machine-learning, Java Security

ACM Reference Format:
Goran Piskachev, Lisa Nguyen Quang Do, and Eric Bodden. 2019. Codebase-Adaptive Detection of Security-Relevant Methods. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330556

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330556

1 INTRODUCTION

Nearly two thirds of security vulnerabilities are caused by simple repetitive programming errors [29]. To ensure the security of their software, more and more companies enforce regular code reviews. Because of the large scale of the code bases produced every day, purely manual reviews are expensive—often prohibitively so. To assist in detecting vulnerabilities, more and more companies use static analysis, a method of reasoning about a program's runtime behaviour from the source code, without running it [27, 43, 48]. A large range of static analysis tools supports the detection of security vulnerabilities such as the SANS top 25 [18] or the OWASP top 10 [42]. Taint analysis in particular, can detect code injections [21] or privacy leaks [20]. Typestate analysis can be used to detect misauthentications [19], API misuses, etc.

In general, such analyses can find a vast array of security vulnerabilities. But to do so effectively, the analysis tool must be correctly configured. In particular, analysis users must configure the tools with lists of *security-relevant methods* (Srm) that are relevant to their development context. Srm are methods of the analyzed program or from application interfaces (APIs) they use, that change the state of the analysis. For example, a call to `getParameter()` of class `javax.ServletRequest` notifies a taint analysis that a potentially user-controlled piece of information enters the program. Likewise, a call to the Java Spring AuthenticationProvider's `authenticate()` can signal a new authentication status to a typestate analysis.

Lists of Srm are generally created manually by the analysis writers, and in larger companies, are often refined by dedicated security teams through manual work. This is important, as even lists used in commercial tools can be incomplete and thus, cause the analyses to miss vulnerabilities or to signal false positives. For instance, a previous study by Arzt et al. showed that, in the past, static analysis tools have frequently missed *a large majority* of relevant findings due to insufficient configurations [9]. In their work, Arzt et al. presented Susi, an automated machine-learning approach for the detection of two types of Srm (sources and sinks) in the Android framework. While the resulting list allows taint analyses to detect more privacy leaks more accurately, it can be used in one case only: the detection of injections and privacy leaks in Android applications. In later work, Sas et al. [44] extend Susi to detect sources and sinks to general Java programs. Both approaches are run ahead of time, before the analysis is deployed.

This paper presents SWAN (Security methods for WeAkNess detection), an approach that directly aids analysis users in detecting Srm in both their code and the libraries they use. Compared to earlier work, SWAN detects two additional types of Srm: validators and authentication methods. This allows analyses to detect more types of vulnerabilities with a higher precision. In addition, SWAN

provides more granularity in the SRM lists, as it is able to differentiate between different vulnerability types in terms of CWEs (Common Weakness Enumerations) [22].

We also extend SWAN to take user feedback into account, allowing developers to adapt SWAN to the code base under analysis at debug time. This allows developers to train the analysis on their own code, thereby significantly improving the list of SRM, and therefore, optimizing the signal-to-noise ratio of the analysis of their code base. Our extension $\text{SWAN}_{\text{ASSIST}}$ is implemented as an IntelliJ plugin that allows code developers to mark SRM in their own code.

We used SWAN to generate SRM lists for twelve popular Java frameworks used for web and mobile development, and home automation. In a 10-fold cross-validation, SWAN achieves a median precision of 0.826 over all frameworks when classifying SRM. Our experiments show that the lists generated by SWAN are sometimes similarly and sometimes more precise than lists from current approaches such as SUSI [9], JoanAudit [49], and Sas et al's [44]. Moreover, we show that $\text{SWAN}_{\text{ASSIST}}$ can significantly improve SWAN's precision by asking the user to manually label only small portion of the codebase's methods.

The main contributions of this paper are:

- SWAN, an approach for the detection of SRM in Java projects: sources, sinks, validators and authentication methods, and their refinement in CWE classes.
- $\text{SWAN}_{\text{ASSIST}}$, an IntelliJ plugin that allows code developers to customize the lists of SRM using SWAN.
- SRM lists generated for twelve popular Java frameworks.
- The manually refined training set used to detect SRM on those frameworks.
- An evaluation of SWAN's precision on the twelve Java frameworks, and a comparison against existing approaches.
- An evaluation method for measuring the effort required by the developer to use $\text{SWAN}_{\text{ASSIST}}$ effectively.

Our implementation and datasets are available online [51] and will be subjected to artifact evaluation. We next explain the requirements for building a machine-learning based detection approach for SRM. Section 3 explains the design of our automated classification approach SWAN, and Section 4 the developer-assisted extension $\text{SWAN}_{\text{ASSIST}}$. We discuss our evaluation and its results in Section 5, followed by a discussion of limitations and future work, related work and our conclusions.

2 REQUIREMENTS FOR THE MACHINE-LEARNING BASED DISCOVERY OF SRM

Table 1 presents the list of the SANS top 25 most dangerous and widespread software errors that can lead to security vulnerabilities. The detection of most of those issues can be done via static *data-flow analysis*. For example, in Listing 1, the variable `userId` at line 3 is assigned a user-controlled (and thus potentially attacker-controlled) value and should therefore be marked as dangerous. This value flows through the program and is used to execute an SQL query (line 7), which can result in an SQL injection (*CWE-89*). This same value is used to create a URL to which the user is redirected (line 9), potentially causing an Open Redirect (*CWE-601*). To avoid those two vulnerabilities, one possibility is to introduce a

```
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).

validator (line 4), which ensures that the input is valid and in the correct format.

To detect SANS 25 problems, data-flow analyses need to be aware of critical points in the program that influence the computation of the analysis: the SRM. In our case, those are *sources*, i.e., methods which create the data that should be tracked (e.g., `getParameter()` line 3), *sinks*, i.e., methods at which the analysis should raise an alarm (e.g., `executeQuery()` at line 7 and `sendRedirect()` at line 9), *validators*, i.e., methods at which the data becomes safe and should no longer be tracked (e.g., `encodeForSQL()` at line 4). In addition, *authentication methods* change the state of the program from safe to unsafe or vice-versa (needed for *CWE-306* and *CWE-862*). Supporting the SANS top 25 thus yields the following requirement for SRM lists.

> **R1** SRM should differentiate between sources, sinks, validators and authentication methods.

When analyzing a program, the choice of SRM can heavily influence the outcomes of the analysis. For example, configuring an analysis with `executeQuery()` as a sink would make it detect SQL injections. Configuring the same analysis with `sendRedirect()` would make it detect Open Redirects. This is further illustrated in Table 1 where we detail the types of methods considered as sources, sinks and validation methods for each of the CWEs. For example in *CWE-306* (Missing Authentication for Critical Function), methods requiring prior authentication are considered sinks, since as soon as they are reached without proper authentication, the analysis should return an error. Authentication methods are thus considered validators, unlike in *CWE-89* (SQL Injection) where validators are typically String sanitizers. This shows that SRM are vulnerability-type specific. An analysis configured with the wrong sets of SRM can easily cause false positives and negatives, yielding what practitioners tend to call a "bad signal". To improve the signal-to-noise ratio, and to aid in the categorization of the analysis warnings, it is therefore important to relate the SRM to each CWE.

> **R2** SRM lists should be specific to each CWE.

In past work, SRM have been extracted from particular libraries and frameworks. SUSI for example, lists all sources and sinks from the Android framework. However, this overlooks SRM in other

Table 1: List of the SANS top 25 CWE [18], and a description of the SRM required to detect them using data-flow analysis. *CWE-120* and *CWE-131* cannot happen in Java: exceptions are triggered before the issues are exploitable. *CWE-863* and *CWE-307* happen on the conceptual level, and should be detected at design time instead.

CWE	Description	Source	Validator	Sink
CWE-89	SQL Injection	External	Code sanitization	SQL execution
CWE-78	OS Command Injection	External	Code sanitization	Execution
CWE-120	Classic Buffer Overflow	N/A	N/A	N/A
CWE-79	Cross-site Scripting	External	Code sanitization	Save / Execution / Print
CWE-306	Missing Authentication for Critical Function	Entry point	Authentication	Critical function
CWE-862	Missing Authorization	Entry point	Authentication function	Critical function
CWE-798	Use of Hard-coded Credentials	New string	Anonymization	Send / Save / Execute
CWE-311	Missing Encryption of Sensitive Data	External	Encryption	Execution
CWE-434	Unrestricted Upload of File with Dangerous Type	External / New string	Type check	Load
CWE-807	Reliance on Untrusted Inputs in a Security Decision	External	Code sanitization / Sanity check	Security function
CWE-250	Execution with Unnecessary Privileges	External / New object	Object sanitization	Execution function
CWE-352	Cross-Site Request Forgery	External	Code sanitization	Save / Execution / Print
CWE-22	Path Traversal	External	Path sanitization	File operation
CWE-494	Download of Code Without Integrity Check	External / New string	Integrity check	Load
CWE-863	Incorrect Authorization	Entry point	Incorrect authorization method	Critical function
CWE-829	Inclusion of Functionality from Untrusted Control Sphere	External / New string	Input sanitization	Load
CWE-732	Incorrect Permission Assignment for Critical Resource	External / New object	Object sanitization	Execution function
CWE-676	Use of Potentially Dangerous Function	External / New object	Object sanitization	Dangerous function
CWE-327	Use of a Broken or Risky Cryptographic Algorithm	External / New object	Object sanitization	Cryptographic API
CWE-131	Incorrect Calculation of Buffer Size	N/A	N/A	N/A
CWE-307	Improper Restriction of Excessive Authentication Attempts	N/A	N/A	N/A
CWE-601	Open Redirect	External	URL sanitization	URL access
CWE-134	Uncontrolled Format String	External	Execution with format	Execution without format
CWE-190	Integer Overflow or Wraparound	External / New integer	Value test	Operation on integer
CWE-759	Use of a One-Way Hash without a Salt	Hashing function	Hashing function with salt	Hashing function

third-party libraries and in the source code itself. External SRM like `encodeForSQL()` in Listing 1 or custom methods defined in the source code will be overlooked by an off-the-shelf SRM list. When analyzing a program, it is thus important to consider all libraries and frameworks it uses, but also its methods as potential SRM.

R3 SRM lists should be specific to the code base.

The Java Spring framework contains more than 30 000 methods [45]. Considering that a reasonably-sized program uses multiple such libraries, it is infeasible to create a complete list of SRM manually. Therefore, it is necessary to compute SRM automatically. Figure 1 presents a high-level workflow of SWAN, our automated SRM detection approach, and $\text{SWAN}_{\text{ASSIST}}$, our extension that queries the analysis user to increase the precision of the generated lists of SRM. SWAN is a classical supervised machine-learning approach, initialized with a training set of classified methods from general Java libraries. Compared to SUSI, SWAN supports Java libraries, and therefore contains a broader set of features and initial training inputs. We expand more on this choice in Section 3. To compensate for the high number of SRM yielded by such a general approach, SWAN additionally classifies the SRM into CWEs, and further allows the analysis user to tune SWAN with $\text{SWAN}_{\text{ASSIST}}$ (dashed and dotted lines), in an active learning approach.

R4 The detection of SRM should be automated but **R5** it should also involve the code developer.

Of the five requirements, SUSI and Sas et al.'s approach meet **R4** and part of **R1**. In the following sections, we extend SUSI into SWAN and $\text{SWAN}_{\text{ASSIST}}$, and answer all requirements.

3 SWAN: SECURITY METHODS FOR WEAKNESS DETECTION

In this section, we present SWAN, a machine-learning classification approach for detecting SRM in Java programs and their libraries.

3.1 General Architecture

SWAN runs the automated classification shown in the lower part of Figure 1 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), one of the three types of *authentication methods* detailed below, or *none* of the above. In the second iteration, it discards the methods marked with *none*, and classifies the remaining SRM into the individual CWEs (**R2**). This is done to avoid classifying non-SRM methods as CWE-relevant.

In the first iteration, SWAN runs a set of four classifications, one for each type of SRM. Since those four sets are not disjoint (e.g., `getContent()` can be both a source and a sink), the classifications are run independently. For sources, sinks, and sanitizers, the classifications are binary, e.g., for the sources, each method is classified in one of the two classes: *source* or *not source*. In the case of authentication methods, SRM are typically distributed between four disjoint classes: *auth-safe-state* (Ass), *auth-unsafe-state* (Aus),

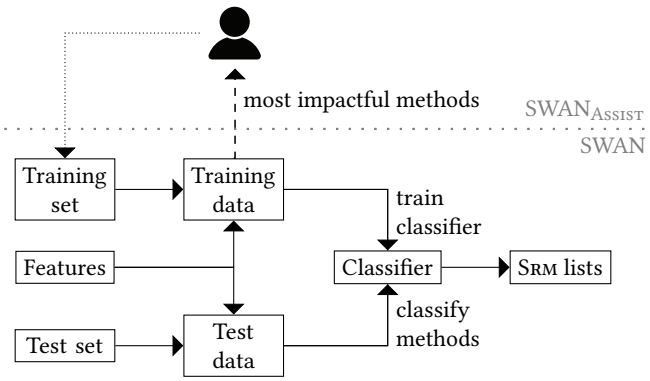

Figure 1: Machine-Learning Approach used in SWAN (solid edges), including the developer's feedback (angled dotted edge - manual action), and the assisted detection of impactful methods (straight dashed edge - automatic action).

auth-no-change (Anc), and *none*. The first one refers to authentication methods that elevate the privileges of the user, e.g., login methods. The second contains methods that lower those privileges (e.g., logout methods). The third category marks methods that do not change the state of the program (e.g., `isAuthenticated()`). Although exceptions are not rare, in most cases seen in our data sets, *Ass* and *Aus* tend to be disjoint. In addition, the two types of authentication methods are semantically very similar. As a result, running three different binary classifications yields a significantly lower precision and recall than a single classification with both classes. *Auc* was thus introduced to reduce the number of such methods being classified unto *Ass* and *Aus*.

In the second iteration of the classification, the training set is kept the same, but the methods that were not classified in any of the SRM classes are removed from the test set. A binary classification is run for each CWE. Currently, SWAN supports seven CWEs: *CWE-78*, *CWE-79*, *CWE-89*, *CWE-306*, *CWE-601*, *CWE-862*, and *CWE-863*.

After running SWAN on the code shown in Listing 1, `getParameter()` is classified as a source for injection vulnerabilities (*CWE-78* and *CWE-89*) and open redirect (*CWE-601*), `executeQuery()` is classified as a sink for *CWE-89*, `sendRedirect()` is classified as a sink for *CWE-601*, and `encodeForSQL()` is found to be a sanitizer for *CWE-89*.

3.2 Features and Training Data

To help the machine-learning algorithm classify the methods into the different classes, SWAN uses a set of binary features that evaluate certain properties of the methods. For example, the feature instance `methodClassContainsOAuth` is more likely to indicate an authentication method than any other type of SRM. As a first phase to the learning, SWAN constructs a feature matrix by computing a true/false result for each feature instance on each method of the training set. This matrix is then used to learn which combination of features best characterizes the classes, and uses this knowledge to classify the methods of the testing set, after creating the feature matrix for that set.

We have identified 25 feature types, instantiated as 206 concrete features, to be relevant for SWAN. We call *feature types* generic features such as `methodClassContains` and *feature instance* their concrete instances (e.g., `methodClassContainsOAuth`). Table 2 shows the list of feature types in SWAN and their number of concrete instances. Overall, 15 feature types, and only 18 feature instances of SWAN, are derived from SUSI, where 10 feature types and 188 feature instances have been added to complete the approach, and make it compliant with **R1**–**R5**. To ensure a good selection of the new feature instances, we manually selected SRM methods from the Spring framework and created feature instances that comply the methods' characteristics.

Compared to SUSI, SWAN contains more general features. For instance, SWAN does not contain Android-specific features such as `Required Permission`. On the other hand, SWAN contains more features based on *method and class names* such as **F03**, **F04**, **F10**, **F14**, **F15**, or **F16**. This is due to the Java naming conventions followed in major libraries, which make functionalities explanatory through naming. Those features are particularly useful for the classification in CWEs, as both method/class names and CWEs are human-defined concepts and match in their descriptions. For example, a call to a database library is made, or when the method is called "query", this can likely denote an SQL injection (*CWE-89*).

SWAN features also support *access control to methods*: SRM are more likely to be publicly accessible, so whether the method is public, private, protected, contained in an anonymous class, or an inner class are covered in **F01**, **F02**, and **F08**, and are used to differentiate between potential SRM and other methods. SWAN also dedicates features to *parameter and return types* like **F21**, **F23**, or **F25**, which can help differentiate between different types of SRM (e.g., void methods are less likely to be sources), and between different types of CWE (e.g., Open redirects (*CWE-601*) most likely take Strings or URLs as inputs.

Other features in SWAN aim at *removing false positives*, e.g., **F11** which helps distinguish constructors from sources, since they both return potentially sensitive data. *Data-flow specific features* (e.g., **F19**, **F20**, **F24**) also serve this purpose, refining the classifications with more information such as whether a parameter flows to the return value (potentially indicating a sanitizer), or if a parameter flows to a method call (denoting a potential sink).

SWAN's features further aim to recognize sanitizers and authentication methods. For example, some instances of **F14** are dedicated to sanitizer detection: `MethodNameContainsSanit`, or `MethodNameContainsReplac`. Similarly, **F19** finds methods that transform a parameter into a return value. In combination with the instance of **F18** applied to Strings (`ParameterContainsTypeString`), this covers the most typical type of sanitizer which replaces sensitive data, or strips dangerous characters in a String. Feature instances have also been created with the three types of authentication methods in mind. Authentication methods are mainly determined through their names or the names of their declaring classes, so they are targeted through instances of **F03** and **F10**, and **F14** such as `methodNameContainsLogin`, or `methodClassContainsOAuth`.

The training set in SWAN contains 235 Java methods collected from 10 popular Java frameworks: Spring [45], jsoup [34], Google Auth [25], Pebble [47], jguard [33], WebGoat [41], and four Apache

Table 2: Feature types of SWAN, and their total number of instances (#I) used within all classifications in SWAN.

Feature	#I	Feature	#I
F01 IsImplicitMethod	1	**F14** MethodNameStartsWith	9
F02 AnonymousClass	1	**F15** MethodNameEquals	3
F03 ClassContainsName	36	**F16** MethodNameContains	46
F04 ClassEndsWithName	3	**F17** ReturnsConstant	1
F05 ClassModifier	3	**F18** ParamContainsTypeOrName	11
F06 HasParameters	1	**F19** ParaFlowsToReturn	1
F07 HasReturnType	1	**F20** ParamToSink	13
F08 InnerClass	1	**F21** ParamTypeMatchesReturnType	1
F09 InvocationClassName	10	**F22** ReturnTypeContainsName	6
F10 InvocationName	39	**F23** ReturnType	5
F11 IsConstructor	1	**F24** SourceToReturn	7
F12 IsRealSetter	1	**F25** VoidOnMethod	1
F13 MethodModifier	4		
Total			*206*

frameworks [1, 2, 7, 8]. We put particular care in ensuring that the methods were chosen so that each of the 206 feature instances of SWAN had at least a positive and a negative example, making each example relevant for the learning algorithm.

3.3 Classifiers

To obtain the feature matrix, SWAN uses the Soot [10, 35, 52] program analysis framework. As its machine-learning module, it uses the SVM learner from the WEKA [53] library. The training set is defined as a JSON file that contains the 235 Java methods mentioned above, annotated with SRM types and CWEs. SWAN accepts a Java program or library as its test set, and runs the two-phase classification, yielding lists of classified test SRM.

WEKA contains different types of classifiers: linear, probabilistic, tree-based, rule-based, etc. We have evaluated seven of them to determine which one would be most appropriate to use in SWAN: Support Vector Machine (SVM), Bayes Net, Naive Bayes, Logistic Regression, C4.5, Decision Stump, and Ripper. We have run a set of ten 10-cross fold validations [46] for each of the classifiers on the training set. The median precision and recall are shown in Table 3. We see that SVM yields the best precision and recall in all cases, classifying on average with 90% of the methods correctly. Naive Bayes also yield good results, and Decision Stump has the lowest precision, with 62,5% for the *CWE-79*. As a result, we chose SVM as the default classifier for SWAN.

4 $\text{SWAN}_{\text{ASSIST}}$: INTEGRATING USER FEEDBACK

We introduce $\text{SWAN}_{\text{ASSIST}}$, an extension of SWAN for integrating developer feedback in the training set to improve the precision of the SRM detection by adapting SWAN to the codebase.

4.1 Active Learning

Because SWAN is designed for general Java applications, when run on one particular program, it may not be precise enough to correctly classify all methods in the code base. In order to improve its precision, we have extended SWAN to query the code developer for their knowledge of the code base (**R3**).

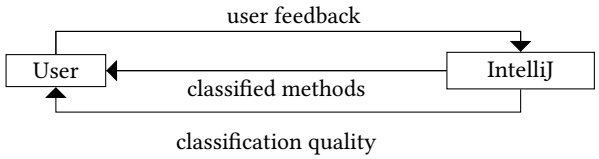

Figure 2: Active learning in $\text{SWAN}_{\text{ASSIST}}$

SWAN is extended with the component $\text{SWAN}_{\text{ASSIST}}$ (as shown in the upper part of Figure 1) that allows developers to edit SWAN' training set directly in their Integrated Development Environment (IDE). The developer can add or remove methods of the training set, or change the classification of a method. The new training set is then fed to SWAN for another classification iteration. To continuously refine the list of SRM, $\text{SWAN}_{\text{ASSIST}}$ uses an active learning approach, integrating the user[1] feedback 2.

In this particular instance of active learning, $\text{SWAN}_{\text{ASSIST}}$ integrates the developer in the loop. It runs the machine learning at each iteration, by changing the training set.This system allows developers to further adapt the classification of the methods in their code base after the original run of SWAN, by improving the training set. Since the user is involved in the process, $\text{SWAN}_{\text{ASSIST}}$ is a semi-automatic approach.

To help the developer identify methods that are most useful to the classification, $\text{SWAN}_{\text{ASSIST}}$ 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 generated for the training set. This is detailed in Section 4.3. Overall, $\text{SWAN}_{\text{ASSIST}}$ uses the automated mechanism of SWAN (**R4**) to detect SRM, and enhances it with developer-based information to improve the precision of the SRM detection (**R5**).

4.2 The IntelliJ Plugin

We have implemented $\text{SWAN}_{\text{ASSIST}}$ as a plug-in component for the IntelliJ IDEA IDE [32]. $\text{SWAN}_{\text{ASSIST}}$ provides an interface for editing the SRM lists and for executing SWAN, updating the SRM classification on demand. Figure 3 presents $\text{SWAN}_{\text{ASSIST}}$'s Graphical User Interface (GUI), which we detail below.

SWAN's training set is shown on the rightmost view of the GUI, called the SWAN_Assist view. Methods in this view can be filtered by classification class or by file. The pop-up dialog in the center 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.

$\text{SWAN}_{\text{ASSIST}}$ also allows the developer to re-run the classification by clicking on the ↻ icon in the toolbar of the SWAN_Assist view. This update configures the inputs for SWAN, runs it in the background, and updates the list of SRM. This is shown as the dotted edge in the upper part of Figure 1. The methods that were just removed are displayed in gray, at the bottom of the list, and can

[1] In this context, typical users would be software developers that seek to configure a static analyzer.

Table 3: Precision (P) and recall (R) of the 10-cross fold validation for all classifiers averaged over 10 iterations in %.

	So		Si		Sa		Auth		*CWE-78*		*CWE-79*		*CWE-89*		*CWE-306*		*CWE-601*		*CWE-862*		*CWE-863*		Average	
	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
SVM	93.8	94	87.8	87.9	97.9	97.9	94.7	94.8	86.5	89.6	77.9	79.9	86.3	90.2	93.4	93.1	85.3	86.3	85.4	85.7	88.8	90	**90**	**90**
BayesNet	94.5	94.6	87.8	88	97.5	97.2	92.9	92.4	89.2	90.4	78.2	78.5	89.4	91	93.6	93.2	82.4	85.3	87.1	87.3	88	88.3	89	90
NaiveBayes	94.5	93.5	86.9	86.8	96.6	96.4	89.4	90.5	88.1	89.9	78	78.6	88.5	90.6	93.1	92.6	82.5	85.5	86.9	87.1	89.1	89	88.5	89.1
LogReg	94.3	94.1	78	78.3	95.2	95.3	94.4	94.2	87.7	89.9	63	79.1	86.5	89.6	93.6	93.4	84	86.2	84.4	85.4	88	88.2	87.5	88.5
C4.5	94.5	94.6	82.4	83	97.4	97.5	93.4	93.8	85.3	89.1	81.6	80.3	86.9	90.2	93.6	93.2	85.6	86.1	86.5	86.6	87.5	87.7	88.6	89.3
Stump	90.4	89.8	66.4	71.5	88.9	89.6	78	86.8	87.2	90.2	62.5	77	82.6	90.3	87.5	84.7	86.3	85.7	84.6	80.4	86.6	83.8	81.9	84.5
Ripper	92.8	93	82.9	83.3	97.4	97.5	89.2	90	86.7	90	70	76.4	85.4	90.4	92.3	91.6	77.5	85	84.5	84.2	87	86.8	86	88

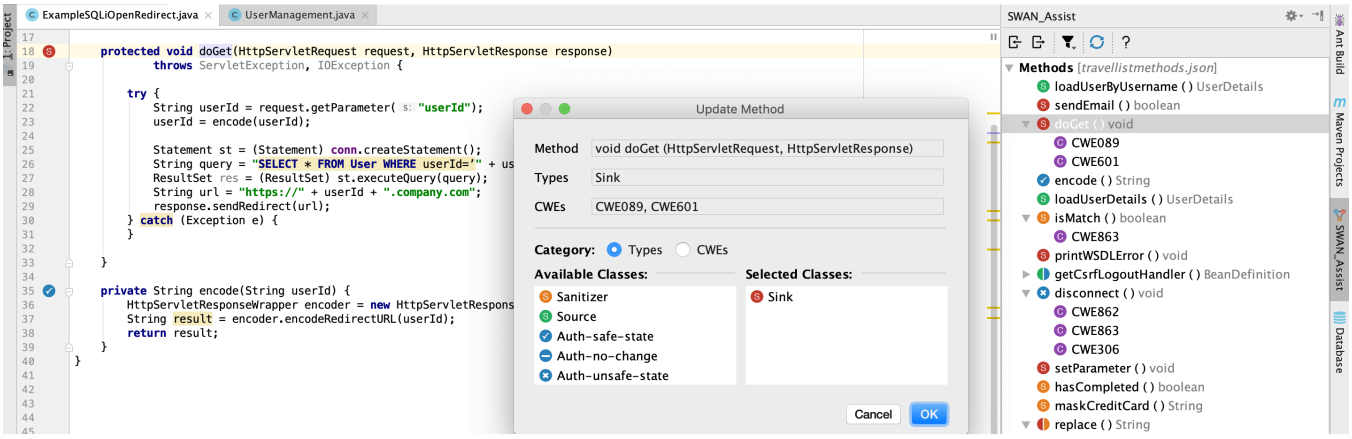

Figure 3: Graphical User Interface (GUI) of SWAN$_{\text{ASSIST}}$. Its components are detailed in Section 4.2

be returned into the training set by using the *restore* functionality from the context menu. Otherwise, they are removed from the list on the next run. We have made the re-running of SWAN manual because of its running times. In general, re-computing the SRM for smaller libraries containing about a thousand methods (e.g., Eclipse Smarthome) takes under one minute. Larger libraries with thousands of methods (e.g., Android) can take up to a few minutes.

4.3 Detecting Impactful Methods

To help developers classify methods more efficiently, the *SuggestSWAN* module provides them with suggestions of methods from the codebase that are likely to have the most impact on the classification (dashed edge in Figure 1). Algorithm 1 presents the suggestion strategy: it computes the method pair that will be the most impactful in the next classification round. This is calculated by iterating over all method pairs. For all features of SWAN, if the method pairs have different values in the feature matrix (i.e., they are a pair of example/counter-example for that feature), the weight of that pair increases. In the end, the pair with the best weight is returned for evaluation. This is repeated until all features are covered, which is monitored by the global $features$ set. When this point is reached, $features$ is emptied and the loop starts again until all methods are classified or until the developer decides to stop.

The weight added to a pair for a particular feature depends on the feature: some features in SWAN are more likely to be impactful than others. We have determined this by evaluating the impact of the individual SWAN features through a One-At-a-Time (OAT) analysis.

Algorithm 1 Choosing the most impactful pair of methods

```
1:  alreadySuggested ← ∅
2:  features ← ∅                          ▷ Keep covered features globally.
3:  function SUGGEST(Boolean[Methods][Features] testSet)
4:      if features = swanFeatures() then
5:          features ← ∅                  ▷ Reinitialize coverage.
6:      (Method m1, Method m2) ← ∅        ▷ Most impactful pair.
7:      for Method m1' in methods do
8:          for Method m2' in methods do
9:              if alreadySuggested.contains(m1', m2') then
10:                 continue
11:             for Feature f in swanFeatures() do   ▷ Add to the
    pair's impact if they have opposite evaluations.
12:                 if testSet[m1'][f] ≠ testSet[m2'][f] then
13:                     updateFeaturesAndWeight(m1', m2')
14:             (m1, m2) ← max((m1, m2), (m1', m2'))
15:     features ← features \ (m1, m2).features
16:     alreadySuggested = alreadySuggested ∪ {m1, m2}
        return (m1, m2)
```

In this analysis, we ran ten 10-fold cross validations on SWAN's training set per class (four SRM and seven CWE classes), disabling one feature instance at a time. For each run of the SVM classifier, we marked the F-measure (harmonic mean of precision and recall) averaged over all repetitions with randomly distributed folds. We

used the F-measure to rank the offsets to obtain the feature weights with which we initialize *SuggestSWAN*.

SuggestSWAN has a quadratic complexity. More complex strategies could be used to suggest methods with a better impact, taking into account several iterations at once. The ideal solution can be reduced to a knapsack problem over all combinations of features, running in an exponential complexity. Since $\text{SWAN}_{\text{Assist}}$ is designed to run in the IDE, we privileged the faster running method to satisfy the need for responsiveness.

5 EVALUATION

We answer the following research questions:

- **RQ1** What is SWAN's precision on real-world Java libraries?
- **RQ2** How does SWAN compare to existing approaches?
- **RQ3** How much manual training does $\text{SWAN}_{\text{Assist}}$ require to obtain optimal precision?

5.1 RQ1: Precision on Real-World Applications

We ran SWAN on a benchmark of twelve popular Java libraries. The benchmark applications were selected to be real-world, open-source Java programs that contain at least 500 methods, and that have evidence of maintenance and development over a recent period of time (i.e., at least 2 years and 5 contributors). They are composed of two frameworks from the mobile domain (Android v4.2 [24] and Apache Cordova v2.4 [3]), eight web frameworks (Apache Lucene v6.6.5 [4], Apache Stratos v4.0 [5], Apache Struts v1.2.4 [6], Dropwizard v1.3 [13], Eclipse Jetty v9.2 [15], GWT v2.8.2 [26], Spark v2.7.2 [31], and Spring v4.3.9 [45]), one framework from the home automation domain (Eclipse SmartHome v0.9 [16]), and one utility framework (Apache Commons[2] v19 [2]).

Table 4 presents the number of Srm detected by SWAN in each Java library. Over the 290,791 methods of the twelve frameworks, SWAN classified 74,603 of them as Srm. We see that a large number of methods are classified as sources and sinks. This is due to the broad definition of sources and sinks, as they should allow an analysis to detect any type of SANS top 25 sources and sinks. However, restricting the Srm to particular CWEs significantly reduces the number of methods to consider (e.g., from 20.39% source Srm to under 1% source/sink/validator/authentication methods for all CWE-specific Srm), and therefore decreases the complexity of the analyses that use them.

Because of the high number of reported Srm, we did not manually verify the complete classification. For each framework, we randomly selected 50 methods from each category of Srm and CWE (or fewer, if the number of methods detected by SWAN was lower), and manually verified their classification. The verification was done be two of the authors and one external researcher. Each person verified one third of the selected methods. The resulting precision of SWAN for each category is presented in Table 5.

Over the different classes, SWAN yields a precision of 0.826 for the Srm classes and of 0.677 for the CWE classes. SWAN is most precise (0.91) when detecting sources. Misclassifications for this category are mostly due to the presence of getter methods in plain old Java objects, which share similarities with source methods (e.g., returning a String). This can be improved by training the model with more counter-examples in the training set. SWAN is least precise for *CWE-862* (0.574), in particular on Spark (0), which is based only on three methods detected making the value an outlier to the dataset. Even though *CWE-862* and *CWE-863* are similar, making their Srm overlapping, the precision of *CWE-863* is better as there are more examples available, and it is more specific. Any authorization information available such as credentials and tokens, are considered in *CWE-863*, but not in *CWE-862* and the frameworks have generally more methods that are related to this.

Other misclassifications cannot be improved by modifying the training set. For example, the Spring method `Connection.getConnection()` has a different behaviour when overwritten in its subclasses: in `SingleConnectionFactory`, it is an authentication method of type *Ass*, and in `ConnectionHolder`, it does not perform an authentication behavior. This information cannot be inferred from the source code, as those two methods are too similar. The differentiation information can be found from the API documentation of the methods. We conclude that SWAN could be improved by adding features that go beyond the code.

Over the twelve libraries, SWAN yields a precision of 0.76. While it is naturally lower than the 0.9 found with the 10-fold cross validation (Table 3), it shows that the generalization of the approach to Java projects is still able to classify Srm with a good precision. The low standard deviation (σ = 0.075) denotes the stability of SWAN' precision over the different Java projects.

SWAN is most precise on *Eclipse Smarthome*, which is explained by the fact that the library is aimed at home automation, and does not contain the web CWEs that SWAN currently supports. Therefore, SWAN could only detect sources and sinks, for which it is strongest. One of the libraries for which SWAN performs the weakest is *Android*, with low precision for authentication methods and methods for *CWE-306*, *CWE-862*, and *CWE-863*. This is due to the keywords used in SWAN' features (e.g., dis/connect), which overlap with domain-specific methods (e.g., connection to Wifi, Bluetooth, or NFC adapter). On our training set, such methods are typically used for authentication, which is not the case for Android. We can conclude that despite its good precision, SWAN still needs more domain-specific information, motivating the need for user input and of $\text{SWAN}_{\text{Assist}}$.

> Over 12 Java libraries, SWAN yields a precision of 0.76. It is more precise for detecting Srm types (0.826) than for CWEs (0.677). SWAN can be improved by adding non source code-specific features, a more complete training set, and domain-specific information. The latter two can be provided by the code developer, using $\text{SWAN}_{\text{Assist}}$.

5.2 RQ2: Comparison to Existing Approaches

We know of the following three approaches to have open-sourced their Srm: Susi [9], Sas et al.'s approach [44], and JoanAudit [49].

Susi and Sas et al. We compare the lists of sources and sinks from Susi [9] and its extension by Sas et al. [44] to the lists of sources and sinks generated by SWAN on the Android framework (version 4.2). The number of sources and sinks detected by the three approaches is shown in Figure 4. SWAN reports a total of 25,085 sources and 13,798 sinks, Susi 18,044 sources and 8,278 sinks, and the tool by

[2]This also includes Apache XML-Xalan, XML-Xerces, XMl-Rcp, HttpComponents, and Oltu-OAuth2

Table 4: Total number of methods (#M), and number of Srm detected by SWAN per category.

	#M	#Srm	So	Si	Sa	Ass	Aus	Anc	*CWE-78*	*CWE-79*	*CWE-89*	*CWE-306*	*CWE-601*	*CWE-862*	*CWE-863*
Android	128,783	39,165	25,085	13,798	503	503	136	288	188	158	334	1,151	229	1,016	109
Apache Commons	24,654	9,129	6,200	2,905	39	81	24	22	126	557	9	110	72	104	35
Apache Cordova	717	273	147	123	3	9	0	1	7	2	0	10	0	2	0
Apache Lucene	3,240	717	491	221	5	0	0	0	0	0	0	0	0	0	0
Apache Stratos	50,724	19,596	12,774	6,602	214	191	44	92	145	26	24	327	107	334	131
Apache Struts	2,670	1,315	752	560	6	0	0	0	360	4	0	0	3	0	0
Dropwizard	659	280	139	137	0	5	0	2	0	0	0	7	0	6	5
Eclipse Jetty	1,157	650	371	255	4	22	4	25	0	20	24	49	27	45	28
Eclipse SmartHome	934	261	185	76	0	0	0	0	0	0	0	0	0	0	0
GWT	44,970	8,093	5,785	2,255	117	7	1	2	41	268	5	10	57	0	0
Spark	884	142	96	22	1	24	0	0	0	0	0	24	6	3	4
Spring	31,369	12,622	7,275	5,138	72	339	36	134	125	301	785	504	233	441	157
Median %	100	31.72	20.39	11.04	0.33	0.41	0.08	0.19	0.34	0.46	0.41	0.75	0.25	0.67	0.16

Table 5: Number of manually verified methods (#MV), and precision of SWAN for each category on twelve Java libraries. N/A marks categories for which SWAN detected no methods.

	#MV	So	Si	Sa	Ass	Aus	Anc	*CWE-78*	*CWE-79*	*CWE-89*	*CWE-306*	*CWE-601*	*CWE-862*	*CWE-863*	Median
Android	650	0.98	0.9	0.98	0.62	0.8	0.66	0.96	0.78	0.62	0.52	0.5	0.52	0.62	0.727
Apache Commons	529	0.88	0.78	0.9	0.74	0.792	0.727	0.9	0.54	0.694	0.75	0.56	0.7	0.743	0.747
Apache Cordova	134	0.88	0.9	1	0.556	N/A	1	1	1	N/A	0.52	N/A	1	N/A	0.888
Apache Lucene	105	0.94	0.68	0.8	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.81
Apache Stratos	594	0.9	0.82	0.78	0.64	1	0.64	0.68	0.769	0.792	0.68	0.56	0.5	0.7	0.721
Apache Struts	163	0.94	0.88	1	N/A	N/A	N/A	1	0.54	N/A	N/A	1	N/A	N/A	0.804
Dropwizard	125	0.88	0.66	N/A	0.8	N/A	0.5	N/A	N/A	N/A	0.8	N/A	0.667	0.8	0.76
Eclipse Jetty	348	0.88	0.68	1	0.773	0.75	0.8	N/A	0.75	0.708	0.714	0.704	0.533	0.714	0.724
Eclipse Smarthome	100	0.96	0.96	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.96
GWT	367	0.92	0.8	0.82	0.857	0	1	0.439	0.6	1	0.8	0.72	N/A	N/A	0.734
Spark	134	0.82	0.955	1	0.875	N/A	N/A	N/A	N/A	N/A	0.917	0.833	0	0.75	0.851
Spring	636	0.9	0.82	0.82	0.68	0.861	0.8	0.9	0.66	0.889	0.82	0.6	0.62	0.8	0.785
Median		0.91	0.82	0.864	0.7	0.862	0.718	0.798	0.648	0.71	0.723	0.608	0.574	0.716	

Sas et al., 3,035 sources and 7,311 sinks. SWAN reports more Srm than the other two approaches, which after a manual investigation we attribute to two reasons. First, SWAN's features target a broader range of vulnerabilities compared to Susi's and Sas et al's data privacy focus. Second, Susi reports methods from abstract classes and interfaces, SWAN reports their concrete implementations, which allows for a better precision. Sas et al. are stricter and only report warnings belonging to certain classes: *database, gui, file, web, xml*, and *io*. Unlike to SWAN and Susi, Sas et al. reports more sinks than sources. This is due to the larger number of sink features than source features contained in their approach. Both SWAN and Susi contain enough features and training instances to overcome this.

To compare the precision of the three approaches, we randomly selected 50 sources and 50 sinks in the lists produced by the three tools, and manually classified them. The selected methods of each tool were labeled by different researcher, two of the authors and one external researcher. SWAN shows a precision of 0.99 for sources and 0.92 for sinks (confirming our findings of **RQ1**), whereas Susi yields respective precisions of 0.96 and 0.88, and Sas et al.'s tool has 0.88 and 0.88 respectively.

JoanAudit. The authors of JoanAudit have manually created lists of 177 Srm classified in five injection vulnerabilities for taint analysis, including sources, sinks, and validators. JoanAudit's Srm are taken from various Java applications, two of which are in common with SWAN: Spring and Apache Commons. Applied to the Spring framework, SWAN is able to detect two of the three methods listed in JoanAudit, the third one being an interface method of which SWAN reports the concrete implementations. On Apache, SWAN detects seven of the ten JoanAudit methods. Two of the missing three are related to the XML injection vulnerability which is not yet included in the classification of SWAN. This indicates that SWAN can be used to create lists of Srm whose quality is comparable to hand-crafted lists such as JoanAudit's.

> For sources and sinks, SWAN yields a higher precision than Susi and Sas et al.'s approach. It can detect Srm with a quality comparable to hand-crafted lists.

5.3 RQ3: Manual Training

We next seek to evaluate how well $\text{SWAN}_{\text{Assist}}$ helps improve the classifiers precision, and in particular how much manual training these improvements require. To evaluate the precision of the active learning approach, we selected a (1) well-maintained, (2) open-source project that contains (3) a high number of Srm, and (4) fewer than 2,000 methods, so that we could classify all of them manually. We used the GitHub mining tool BOA [14] and selected the Gene

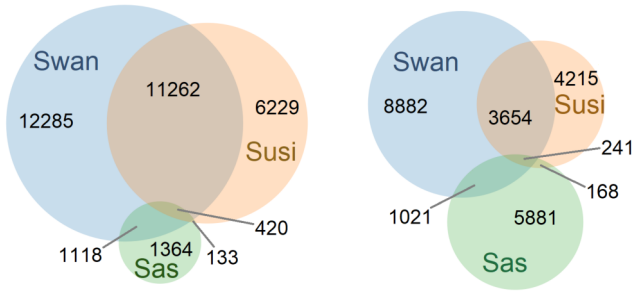

Figure 4: Number of sources (left) and sinks (right).

Expression Atlas (GXA) [30] application, a popular tool in the domain of bioinformatics maintained by the European Bioinformatics Institute (EMBL-EBI) [17]. We chose GXA since it showed lower precision with the base SWAN. This allows us to showcase the potential of the active learning approach in the worst case compared to an application that already has a good precision to begin with.

We manually classified and labeled the 1,663 methods of GXA with one or more of the following classes: sources, sinks, *CWE-89*, none. 286 methods were identified as sources, 183 methods as sinks, and 29 as relevant to *CWE-89*, and consider this our ground truth. The labeling of the methods was done twice, first by one of the authors, and then by one external researcher. The Cohen's Kappa value for sources is 0.605, 0.725 for sinks, and 0.919 for CWE089, which are all above the significant agreement threshold of 0.6 [36].

To evaluate how SWAN$_{\text{ASSIST}}$'s suggester algorithm (Section 4.3) helps improve the results of SWAN, we compare the resulting SRM lists when feeding SWAN$_{\text{ASSIST}}$ randomly selected method pairs, and when using *SuggestSWAN* to select those pairs. We first run SWAN with its initial training set and GXA as a testing set. Then, add a new method pair to the training set and continue until we run out of methods. For each of the 819 iterations, we report the classification's precision it in Figure 5. The precision shown for the random suggester is averaged over 10 runs.

We see that for sources and sinks, the evolution of the precision for the random suggester is linear. This shows that the suggester does not help the classification: the precision increases naturally as a new pair is added to the training set. On the other hand, SWAN shows a quick increase in precision at the beginning, showing that the suggester is efficient in selecting the methods with the most impact first. This maximizes the impact of the classification and minimizes the developer work to tune SWAN to their code base. In the case of sources, the precision reaches 0.8 at iteration 31 (from 0.75 at iteration 1), making 60 methods labeled (4% of the total number of methods in the application). Afterwards, the growth slows down, reaching a precision of 0.9 after 91 iterations. In the random case, the growth is much slower, reaching a precision of 0.8 at iteration 166 and 0.9 at iteration 414. For sinks, the precision using reaches 0.9 at iteration 20 (from 0.61 at iteration 1) with *SuggestSWAN*, requiring the developer to only label 1% of the total number of methods in the application. This precision is only reached at iteration 358 with the random suggester.

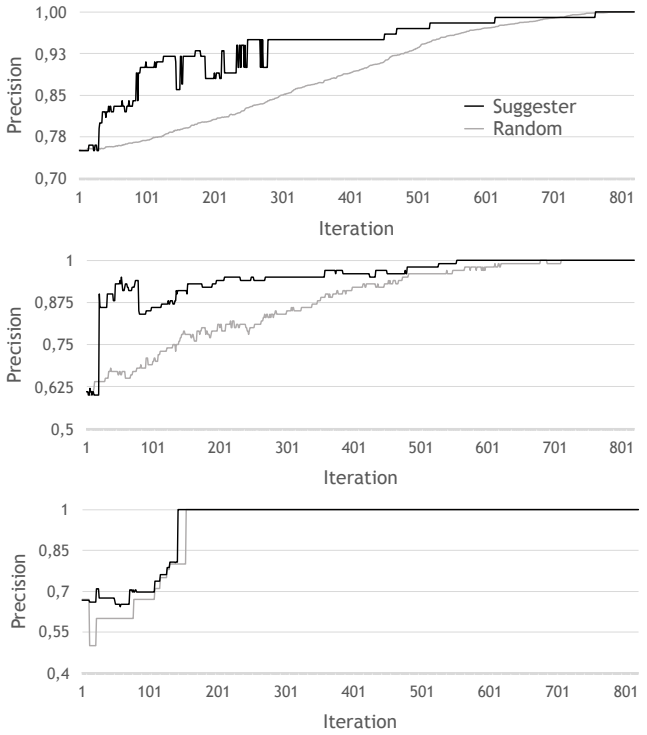

Figure 5: Precision of the sources (top), sinks (middle), and *CWE-89* (bottom) over 819 iterations of SWAN by adding methods with *SuggestSWAN* and with random selection

Although less visible, we see a similar trend for the case of *CWE-89* where the precision in the early iterations is better when using *SuggestSWAN*. The growth is less pronounced, only reaching a precision of 0.8 at the 130th iteration (from 0.67 at iteration 1), requiring 16% of the methods to be labeled. We attribute this to the lower number of SWAN instances targeting this class compared to the SRM classes, and to the low number of *CWE-89*-related methods in the test set, making the classifier less efficient in targeting it.

In all three cases, we note regular drops in the precision. Further investigation reveals that those drops occur when a problematic method is added to the training set. For example, method *void uk.ac.ebi.gxa.utils.EfvTree.put(uk.ac.ebi.gxa.utils.EfvTree)* is expected by the classifier to be a sink method, since it does not return anything, contains "put" in its name, and accepts an argument. However, the method is a simple accessor method, and does not constitute a sink. Such methods pollute the training set, and make the classification less precise until enough methods are added to compensate for the uncertainty. This issue can be mitigated by improving SWAN's features, or through a smarter handling of the problematic methods and when to add them to the training set to minimize pollution. The presence of such methods also shows one more reason for why a user-guided approach such as SWAN$_{\text{ASSIST}}$ is useful, particularly in the presence of imperfect training sets.

Using *SuggestSWAN* on GXA yields high precision significantly faster than with a random selection of methods.

6 LIMITATIONS AND THREATS TO VALIDITY

$\text{SWAN}_{\text{Assist}}$'s main usability issue lies in the manual trigger of the re-classification. This design choice was made because of SWAN's running time. While it takes a few seconds on small libraries with few hundred methods, it can take up to a few minutes for larger libraries with more than 100,000 methods such as Android. As a result, we reduce the number of times SWAN is run. In addition, on re-run, the tool starts a new background process for SWAN that does not block the GUI and let users continue working in the IDE.

For each RQ in Section 5 a manual classification of methods was performed which is difficult task that requires expertize in static analysis and security. In each case, the work was done by two or three persons. Some methods can be considered Srm in one context of use, but not in other, making the classification more difficult. In order to reduce any bias in **RQ2**, we did not reveal the names of the tools to each person. For **RQ3**, we reported the inter-rated reliability (Subsection 5.3).

7 RELATED WORK

7.1 Learning Srm

Many static analyses use Srm to configure their analyses. For example, in the domain of Android applications, the Srm are typically computed using Susi-like approaches. This makes those analyses [11, 12, 39, 40] susceptible to Susi's weaknesses. For instance, those approaches do not consider sanitizers. SWAN and $\text{SWAN}_{\text{Assist}}$ support sanitizers, and include user feedback in order to refine the list and reduce the number of false positives.

Susi [9] is a machine-learning approach for sources and sinks in the Android framework. It uses 26 feature types and runs two iterations of maching-learning to first classify methods as sources, sinks, or neither, and then, in different Android-specific classes such as bluetooth, browser, etc. SWAN extends Susi to detect sanitizers and authentication methods, on top of sources and sinks. It also allows for classifications into CWE sub-classes. Unlike Susi, which is specific to Android, SWAN generalizes Susi to Java applications. It thus loses in precision but makes up for it by introducing $\text{SWAN}_{\text{Assist}}$, which interleaves the code developer with the Srm detection task. This allows SWAN to generate Srm that are more specific to the analyzed code base.

Sas et al. [44] introduce the need for generalizing the detection of Srm for general Java libraries, and the classification in CWE classes. They extend Susi, modifying its features to achieve the former goal. But unlike SWAN, they do not address the latter. Similarly to Susi, Sas et al.'s approach detects sources and sinks offline. SWAN can additionally recognize sanitizers and authentication methods, and classify Srm by CWEs. In conjunction with $\text{SWAN}_{\text{Assist}}$, our approach provides more adapted functionalities to support code developers using static analysis in practice.

Like SWAN, Merlin [37] also detects Srm automatically. It uses probabilistic inference to detect specifications for taint-style analyses of string-based vulnerabilities. It models information flow paths in a propagation graph using probabilistic constraints. However, the resulting Srm are specific to the application of Merlin, i.e. string-based vulnerabilities, and Merlin does not provide support to classify them in sub-types such as CWEs.

7.2 Machine Learning and Developer Feedback in Static Analysis

Past approaches have included machine-learning and require developers' feedback to refine static analysis results. For example, Fry et al. [23] use machine learning to cluster analysis warnings into similarly actionable warning groups. Heckman et al. [28] apply the method to determine which warnings are more likely to be false positives. In such cases, machine-learning is used offline, after the analysis is run, and before the results are shown to the developer. They do not include developer feedback.

Other approaches include developer feedback. For example, Aletheia [50] filters the results that it displays by learning the needs of developers. Similarly to SWAN, Aletheia shows the developer a portion of the warnings and asks the developer to classify them, thus instantiating features for the machine learning algorithms. This classification is then used as a filter in the UI. The classification phase takes place earlier with SWAN, before the analysis runs. This reduces the analysis time and makes it usable in the IDE, similarly to Lucia et al.'s approach [38], which uses incremental machine learning to detect false positives in real time. SWAN achieves the same results, applied to the problem of Srm discovery.

8 CONCLUSION AND FUTURE WORK

Configuring static analyses with the correct set of Srm is an important factor in the precision of those analyses. In this paper, we presented SWAN, an automated approach for detection of Srm and their sub-classification in CWEs, and $\text{SWAN}_{\text{Assist}}$, an active learning approach that allows the adaption of the Srm lists to a particular codebase using feedback from the user. SWAN is able to detect sources, sinks, validators, and authentication methods with a high precision, and perform better or with a comparable precision to existing approaches. The sub-classification of those Srm into different CWE filters a large number of methods that can cause false warnings in the CWE detection. As a tool integrated in the developer's IDE, $\text{SWAN}_{\text{Assist}}$ requires little user feedback to significantly improve the Srm lists.

It is possible to improve the precision of static analyses by providing more granular Srm information. Not only the methods themselves are important, which objects they affect can also be important (i.e., which parameter, static variable, return variable, or base object). We leave the detection of such affected objects to future work. Additionally, we plan to extend SWAN and $\text{SWAN}_{\text{Assist}}$ to support a larger number of CWEs. We also plan to improve SWAN' training set in a more systematic manner, to ensure a better precision of the approach, and develop a better strategy to handle potentially problematic methods in *SuggestSWAN*.

ACKNOWLEDGMENTS

We thank Oshando Johnson for the work on the implementation of $\text{SWAN}_{\text{Assist}}$ and Parviz Nasiry for the work on the OAT analysis. This research was supported by a Fraunhofer Attract grant, the Heinz Nixdorf Foundation, the Software Campus Program of the German Ministry of Education and Research as well as the NRW Research Training Group on Human Centered Systems Security (nerd.nrw). It was partially funded by the DFG project RUNSECURE.

REFERENCES

[1] Apache. [n. d.]. Abdera. https://abdera.apache.org/.
[2] Apache. [n. d.]. Apache Commons. https://commons.apache.org/.
[3] Apache. [n. d.]. Apache Cordova. https://cordova.apache.org/.
[4] Apache. [n. d.]. Apache Lucene. http://lucene.apache.org/.
[5] Apache. [n. d.]. Apache Stratos. http://stratos.apache.org/.
[6] Apache. [n. d.]. Apache Struts. https://struts.apache.org/.
[7] Apache. [n. d.]. Roller. http://roller.apache.org/.
[8] Apache. [n. d.]. Tomcat. http://tomcat.apache.org/.
[9] S. Arzt, S. Rasthofer, and E. Bodden. 2013. SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks *(NDSS'13)*.
[10] S. Arzt, S. Rasthofer, and E. Bodden. 2017. The Soot-based Toolchain for Analyzing Android Apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*. IEEE Press, Piscataway, NJ, USA, 13–24. https://doi.org/10.1109/MOBILESoft.2017.2
[11] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299
[12] L. Nguyen Quang Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill. 2017. Cheetah: Just-in-time Taint Analysis for Android Apps. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17)*. IEEE Press, Piscataway, NJ, USA, 39–42. https://doi.org/10.1109/ICSE-C.2017.20
[13] Dropwizard. [n. d.]. Dropwizard. https://www.dropwizard.io/.
[14] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 422–431.
[15] Eclipse. [n. d.]. Jetty. https://www.eclipse.org/jetty/.
[16] Eclipse. [n. d.]. Smarthome. https://www.eclipse.org/smarthome/.
[17] European Bioinformatics Institute (EMBL-EBI). [n. d.]. EMBL-EBI home page. https://www.ebi.ac.uk/. Online; accessed 10 December 2018.
[18] Common Weakness Enumeration. [n. d.]. 2011 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.
[19] Common Weakness Enumeration. [n. d.]. CWE-287: Improper Authentication. http://cwe.mitre.org/data/definitions/287.html.
[20] Common Weakness Enumeration. [n. d.]. CWE-359: Exposure of Private Information ('Privacy Violation'). https://cwe.mitre.org/data/definitions/359.html.
[21] Common Weakness Enumeration. [n. d.]. CWE CATEGORY: OWASP Top Ten 2017 Category A1 - Injection. https://cwe.mitre.org/data/definitions/1027.html.
[22] Common Weakness Enumeration. [n. d.]. CWE home page. http://cwe.mitre.org/. Online; accessed 27 September 2018.
[23] Z. P. Fry and Westley. 2013. Clustering static analysis defect reports to reduce maintenance costs. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 282–291. https://doi.org/10.1109/WCRE.2013.6671303
[24] Google. [n. d.]. Android API 4.2. https://developer.android.com/about/versions/android-4.2.
[25] Google. [n. d.]. Google Auth Java. https://github.com/googleapis/google-auth-library-java.
[26] GWT. [n. d.]. GWT. http://www.gwtproject.org/.
[27] M. Harman and P. O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *Proceedings of the 2018 IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM) (SCAM '18)*. IEEE.
[28] S. Heckman and L. Williams. 2009. A Model Building Process for Identifying Actionable Static Analysis Alerts. In *2009 International Conference on Software Testing Verification and Validation*. 161–170. https://doi.org/10.1109/ICST.2009.45
[29] J. Heffley and P. Meunier. 2004. Can source code auditing software identify common vulnerabilities and be used to evaluate software security?. In *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the.* 10 pp.–. https://doi.org/10.1109/HICSS.2004.1265654
[30] European Bioinformatics Institute. [n. d.]. Gene Expression Atlas. https://github.com/gxa/gxa.
[31] Spark Java. [n. d.]. Spark. http://sparkjava.com/.
[32] JetBrains. [n. d.]. IntelliJ home page. https://www.jetbrains.com/idea/. Online; accessed 17 October 2018.
[33] JGuard. [n. d.]. JGuard. http://jguard.net/.
[34] jsoup. [n. d.]. jsoup. https://jsoup.org/.
[35] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*.
[36] J. R. Landis and G. G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (1977), 159–174. http://www.jstor.org/stable/2529310
[37] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. 2009. Merlin: Specification Inference for Explicit Information Flow Problems. *SIGPLAN Not.* 44, 6 (June 2009), 75–86. https://doi.org/10.1145/1543135.1542485
[38] Lucia, D. Lo, L. Jiang, and A. Budi. 2012. Active refinement of clone anomaly reports. In *2012 34th International Conference on Software Engineering (ICSE)*. 397–407. https://doi.org/10.1109/ICSE.2012.6227175
[39] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller. 2017. Detecting Information Flow by Mutating Input Data. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 263–273. http://dl.acm.org/citation.cfm?id=3155562.3155598
[40] A. Mendoza and G. Gu. 2018. Mobile Application Web API Reconnaissance: Web-to-Mobile Inconsistencies amp; Vulnerabilities. In *2018 IEEE Symposium on Security and Privacy (SP)*. 756–769. https://doi.org/10.1109/SP.2018.00039
[41] OWASP. [n. d.]. WebGoat. https://github.com/WebGoat/WebGoat.
[42] Open Web Application Security Project. [n. d.]. OWASP Top 10 Most Critical Web Application Security Risks. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.
[43] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720
[44] D. Sas, M. Bessi, and F. A. Fontana. 2018. [Research Paper] Automatic Detection of Sources and Sinks in Arbitrary Java Libraries. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 103–112. https://doi.org/10.1109/SCAM.2018.00019
[45] Java Spring. [n. d.]. Java Spring. https://spring.io/.
[46] M. Stone. 1974. Cross-validatory choice and assessment of statistical predictions. *Journal of the royal statistical society. Series B (Methodological)* (1974), 111–147.
[47] Pebble Templates. [n. d.]. Pebble. https://pebbletemplates.io/.
[48] T. W. Thomas, M. Tabassum, B. Chu, and H. Lipford. 2018. Security During Application Development: An Application Security Expert Perspective. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. ACM, New York, NY, USA, Article 262, 12 pages. https://doi.org/10.1145/3173574.3173836
[49] J. Thomé, L. K. Shar, D. Bianculli, and L. C. Briand. 2017. JoanAudit: A Tool for Auditing Common Injection Vulnerabilities. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 1004–1008. https://doi.org/10.1145/3106237.3122822
[50] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 762–774. https://doi.org/10.1145/2660267.2660339
[51] Paderborn University and Fraunhofer IEM. [n. d.]. SWAN and SWAN Assist github repository. https://github.com/secure-software-engineering/swan. Online; published 03 November 2018.
[52] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. 2000. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?. In *CC*. 18–34. https://doi.org/10.1007/3-540-46423-9_2
[53] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. 2016. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques* (4th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.