Automatic Detection of Sources and Sinks in Arbitrary Java Libraries

Darius Sas
Dipartimento di Informatica Sistemistica e Comunicazione
Universitá degli Studi di Milano-Bicocca
Milan, Italy
d.sas@campus.unimib.it

Marco Bessi
CAST Software Italia
Milan, Italy
m.bessi@castsoftware.com

Francesca Arcelli Fontana
Dipartimento di Informatica Sistemistica e Comunicazione
Universitá degli Studi di Milano-Bicocca
Milan, Italy
arcelli@disco.unimib.it

***Abstract*—In the last decade, data security has become a primary concern for an increasing amount of companies around the world. Protecting the customers *privacy* is now at the core of many businesses operating in any kind of market. Thus, the demand for new technologies to safeguard user data and prevent data breaches has increased accordingly. In this work, we investigate a machine learning-based approach to automatically extract sources and sinks from arbitrary Java libraries. Our method exploits several different features based on semantic, syntactic, intra-procedural dataflow and class-hierarchy traits embedded into the bytecode to distinguish sources and sinks. The performed experiments show that, under certain conditions and after some preprocessing, sources and sinks across different libraries share common characteristics that allow a machine learning model to distinguish them from the other library methods. The prototype model achieved remarkable results of 86% accuracy and 81% F-measure on our validation set of roughly 600 methods.**

***Index Terms*—Java, Static Analysis, Sources, Sink, Machine Learning**

***Research paper* – This article has been submitted to the Research Paper track at SCAM2018.**

I. Introduction

A. Overview

In the last decade, data security has become a primary concern for an increasing number of companies around the world. Protecting the customer's *privacy* is now at the core of many businesses operating in any kind of market. Thus, the demand for new technologies to safeguard user data and prevent data breaches has increased accordingly. The impact of an hypothetical data breach is not restricted only to the company under attack, but also affects their customers, providers and users. Indeed, according to IBM, the potential cost of cybercrime to the global community is around 500 million dollars, while the average data breach cost per company is 3.62 million dollars [1].

Security vulnerabilities detection is one of the most important focal points of the whole matter. A secure software can guarantee a higher level of protection of the data it handles, preventing several attacks that might damage the economic stability of a company.

B. Tracking and measuring security vulnerabilities

The presented work aims at enhancing a particular technique of static analysis [2] known as Taint Analysis, or Information Flow Analysis [3], a popular method that consists in checking which variables have been modified by untrusted user input. Usually, all user input can be dangerous if it is not properly checked.

In general, the goal of Taint Analysis is to ensure that malicious input does not reach any possibly vulnerable function call. Dangerous input sources are usually referred to as *data sources*, while vulnerable function calls, or method invocations, are referred to as *data sinks*. Therefore, Taint Analysis' main goal is to find dataflow paths connecting a source with at least one sink. Each path represents a vulnerability in the application under analysis. To fix such vulnerabilities, malicious input can be sanitised by scanning untrusted input and neutralise, or remove, malicious characters.

The vulnerabilities detected by Taint Analysis share the same base idea: (a) inject malicious data into sensible data sinks to gain access to restricted areas of the application; (b) execute remote commands on the application's server; (c) leak personal information belonging to a single, or a group, of users of the application; or manipulate the behaviour of the application to exploit a specific victim in performing restricted operations.

C. Motivation

Taint Analysis can be computed statically [4] or dynamically [5]. Both approaches exploit a list of data sources and sinks to discover unsafe data paths. Usually, the list is provided as input before starting the analysis. The completeness of such a list is vital. If a sink is missing, several flows will remain undetected and could be exploited by an attacker to inject malicious code. The same occurs if a source is missing: unchecked data may enter the application and reach a sink.

Existing analysis tools, both static and dynamic, focus on a handful of hand-picked sources and sinks, and can thus be circumvented by malicious applications with ease [6]. There are several different issues with such an approach to select sources and sinks:

- big applications have thousands of methods that could be sources or sinks, requiring a lot of human time to complete a single list;
- new versions of the application may have modified the APIs, requiring to recompile or update the list;
- hand-picking sources and sinks is an error-prone task, especially when dealing with a huge amount of methods;
- the process is hardly scalable and, in some cases, requires consulting several different information sources before correctly assigning a tag.

In the presented work, we investigate a machine learning approach to automatically identify malicious data sources and sensitive data sinks in arbitrary Java libraries in order to achieve the following goals:

1) Study the distinctive traits, or features, of data sources and data sinks in arbitrary Java libraries that would allow a machine learning model to accurately classify them.
2) Create a tool for automatically extracting data sources and data sinks from arbitrary Java libraries that is flexible and easy to use.
3) Test and validate the performance of such a tool with real-world Java libraries.

D. Challenges

The problem addressed in this work presents a series of challenges that one should overcome in order to produce a complete solution to the original problem. Such challenges include:

- Cover several different types of injections, from SQL injection to log forging and XPath injection. This is non-trivial, so we restricted the addressed vulnerability types to the most important ones[1]:
  - OS Command Injection (CWE-78);
  - Log Forging (CWE-117);
  - Path Manipulation (CWE-73);
  - SQL Injection (CWE-89);
  - XPath Injection (CWE-91);
  - XSS Injection (CWE-79).

  However, the possibility to extend the types of data sources and sinks that the model could recognise was an additional crucial feature that we wanted to support.
- Generalise the approach so it is able to handle any arbitrary Java library that one might need to analyse. This implies that no library-specific cue can be explicitly exploited to complete the classification task.

E. Structure

The paper is organized as follows: in Section II, the main work on which our work is based and the differences between the two are presented. In Section III, the proposed approach is introduced along with the main definitions necessary to formally define the classification model and the feature classes that the model uses to classify the methods. Section IV is organised into two subsections: Section IV-A presents the classification model selection and other secondary results, while Section IV-B presents the obtained results. Section V reports an in-depth analysis of the contribution of a set of features to the overall classification and briefly describes the previous versions of the proposed approach. Next, Section VI presents possible issues, limitations and weaknesses of both our study and model. In the end, Section VII concludes the paper and explores some possible future developments to address the current gaps in the approach.

II. Related work

In this section, we briefly introduce the paper on which the presented work is based and highlight the differences between the two approaches and the problems addressed.

In 2013 Rasthofer et al. [6] presented an approach for automatically extracting the list of sources and sinks from the Android operating system. They used a machine learning approach to *classify* and *categorise* all of the 110.000 public methods of the Android Framework into sources and sinks. The tool developed is called SuSi and is available as an open source software[2].

Rasthofer et al. achieved a noteworthy result of over 90% precision and recall on the training set using ten-fold cross-validation. They also validated SuSi on methods that had never been seen by the model and achieved almost 100% precision and recall [6], using Google Glass and Google Chromecast APIs as a validation set. Further details on SuSi, as well as on data source and data sinks, are also available [7].

Comparing the two approaches, the main noticeable difference is that SuSi is designed to only recognise sources and sinks from the Android Framework, whilst we aim to achieve the same goal but on a more general level, trying to classify methods from *generic* Java libraries. The semantics resilient in the Android's bytecode have been explicitly exploited by SuSi, allowing interfaces, abstract classes and naming conventions to be used to create ad-hoc features. Using the same strategy for generic Java libraries is much more difficult since it is not possible to automatically deduce key interfaces, abstract classes and naming conventions precisely to exploit for feature calculation. Nevertheless, we model these characteristics by defining the concept of *Resource* and redefining the concept of *Resource Method* from [6], namely Java's key interfaces and classes for I/O operations[3]. Another difference lies in the validation approach we undertook with our tool. We decided to validate our tool using a validation set created from a much larger sampling pool than the one used by Rasthofer et al. [6]. In fact, we validated our approach on a set of methods that has been sampled from a completely different set of libraries than the one used to sample the training set.

Further works by Arzt and Rasthofer [4] have used the results produced by SuSi as input for their Taint Analysis tool specifically designed for the Android Framework.

[1] In parenthesis, MITRE's Common Weakness Enumeration for each vulnerability. Available at https://cwe.mitre.org/.

[2] See www.blogs.uni-paderborn.de/sse/tools/susi.

[3] The precise and complete definition will be presented in Section III-B.

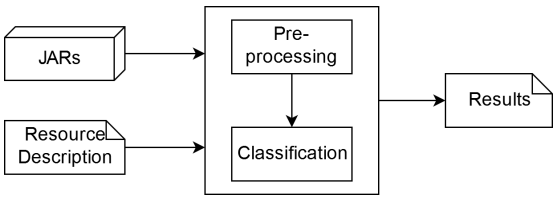

Fig. 1. A general view of the proposed approach.

In the domain of dynamic analysis, data sources and data sinks have been used by Gibler et al. [8] to develop an approach to dynamically detect potential leaks of private information.

In our case, the tool was developed to be integrated in CAST Software's[4] security analysis tool, in order to automate the manual step of adding sources and sinks for the different Java libraries internally developed by their customers.

III. The Proposed Approach

A. General description

The classification of data sources and data sinks is performed by analysing the bytecode of each method present in the libraries provided as input by the user. The model uses a combination of semantic and syntactic cues to assign to each method an adequate tag. As Figure 1 shows, the analysis is a two-fold process composed of the following phases:

1) ***Preprocessing***: a special set of classes, named *Resources*[5], and methods, named *Resource Methods*, are identified and extracted from the input JAR files using the additional information present in the resource description (Figure 1). Next, for each method, a set of features is computed.
2) ***Classification***: a machine learning model is used to classify every given method as:
   - *Sink*: the class that identifies data sinks;
   - *Source*: the class that identifies data sources;
   - *Mixed*: a special class used when a method has both the behaviour of a source and a sink;
   - *None*: the class that identifies methods that do not perform any I/O operation.

The final output of the model is a list containing the classification for each method contained in the given Java libraries.

As a final regard, the technologies used to implement our approach are: (1) the Soot Framework [9] for extracting information from the Java Bytecode of the JARs, and (2) the WEKA Data Mining Framework [10] for the machine learning part involving the classification process.

B. Definitions

Modelling the I/O operations of the Java environment is a key step in interpreting the nature of the analysed methods. However, describing how the approach models the different information available in the input JARs requires us to introduce some terminology. Thus, we present the list of definitions necessary to model I/O operations that are based on the work performed by [6], which were extended to better fit the needs of our approach:

Definition 1. *A **Resource** is an entity E on which a process is able to perform input and output operations with the external environment.*

Some examples of Resources are a file, a database or a web protocol. Java programs model resources using classes and interfaces, thus:

Definition 2. *A **Resource Class** (RC) is a class $C \in \Sigma$ that embodies I/O operations on one or multiple resources E.*

where Σ, called *Scene*[6], is the set of all classes under analysis. An example of a resource class is `java.io.FileReader`. Note that Resource Classes are not necessarily bound to belong to the Java Framework only; they can also be classes defined in the library under analysis.

Definition 3. *The method $m \in C$ is a **Resource Method** (RM) if $C \in \Sigma$ is a class that implements (or inherits) I/O operations from a Resource Class $R \in \Sigma$. Additionally, the name prefix of m must be in $\mathcal{P} = \mathcal{P}_I \cup \mathcal{P}_T \cup \mathcal{P}_M$. Note that it may be that $C = R$.*

$\mathcal{P}$ is the union of the prefix sets, where each element of such sets is a commonly used prefix by input methods ($\mathcal{P}_I$), sink methods ($\mathcal{P}_T$), or mixed methods ($\mathcal{P}_M$). For brevity's sake, we won't extensively enumerate the elements of each $\mathcal{P}_x$, but some example of the elements of such sets are *read*, *get*, and *execute*. The elements of these sets have been selected based on the work of Rasthofer et al. [6] and heuristically during the development of the tool used to test the approach.

Before continuing with the next definition, we would like to note that Resource Methods could be seen as *root* sources and sinks that are going to be invoked by other methods in the library which will read or write data from them.

Finally:

Definition 4. *A **data source** (or input) method is a declaration or an implementation of a method that has at least one* ingoing *dataflow interaction from a Resource Method m.*

*A **data sink** (or sink) method is a declaration or an implementation of a method that has at least one* outgoing *dataflow interaction to a Resource Method m.*

Declarations are considered sources or sinks because implementing classes will probably implement a data source or data sink behaviour. All Resource Methods are thus either sources or sinks.

C. Resource Modeling

Modelling resources is not a trivial task. In fact, our approach does not fully address this aspect of the problem. The

[4]Check out https://www.castsoftware.com/ for more information.

[5]See Section III-B for a detailed description.

[6]We adopt the same nomenclature used by Vallée-Rai et al. in [9].

main issue with resource classes is that they can be defined by the developers of a library, thus every library might have its very own set of resource classes which cannot be modelled a priori.

We adopted a partial workaround to this issue by compiling a brief list of well-known resource classes from the Java Framework – around 30 between classes and interfaces. At analysis-time, we extend the list using inheritance to extract from the current scene under analysis all the subclasses inheriting from or implementing any of the initial elements of the list. The resulting list is the initial set of resource classes used in the rest of the analysis. User-defined resource classes that do not inherit from or implement any class in the initial list are thus undetectable. To address this limitation, we propose a possible solution in Section VII-B.

The list has been divided into categories according to the type of operation performed by the classes. The categories reflect the CWEs mentioned in Section I-D. The advantage of a category-based approach is that new classes can be added to the model without further changes in the model (like retraining the machine learning model), including adding elements at run-time, before calculating the features for each method. The addition of new categories will require additional changes though.

D. The features used by the machine learning model

The underlying Machine Learning model classifies the data through a set of roughly 100 features of different types. The features are based on syntactic and semantic aspects of the Java software development process, such as naming conventions, redundancies, regularities and coding styles, as well as more technical aspects of each method analyzed.

As noted by Rasthofer et al., a single feature alone will not give enough information to any algorithm to correctly distinguish between the different categories. However, by instantiating several features from a set of feature classes, a fairly precise model can be trained [6]. The feature classes are:

Method prefix The method starts with one of the prefixes in $\mathcal{P}$. Usually input methods tend to use prefixes like *get* or *read*, whereas sink methods use prefixes like *set*, *put* or *write*.

Invokes a RM The method invokes a RM whose name starts with a prefix in $\mathcal{P}$. These features are based on the same assumption as the previous one.

Return type The return type of the method is of a certain type (or a subtype of a RC). Typically, input methods have a return value that is an object or an array, whereas sink methods often return `void`.

Declaring class' name contains The name of the declaring class of the method contains a specific substring or prefix. Normally, classes that expose I/O operations may contain some special keywords, such as *file*, *http*, *sql* or similar.

The number of parameters Simply counts the number of parameters. Sinks usually have at least one parameter, while sources may have zero (data is returned via the return construct) or more (data is returned via one or more parameter objects).

Paremeters' type The parameters are of certain types. Methods that accept as parameters Resource Classes (RC), or other special types, may perform a specific I/O operations.

Dataflow from parameters The method's parameters flow through its body until reaching a RM or `this` object. Since a RM represents I/O operations, dataflow from parameters to a RM or `this` are typical of sinks;

Dataflow from RM The RM's return value flows through its body until reaching the return value, `this` object, or to any of the method's parameters. Dataflow from RMs to `this` or method's parameters or to the return statement usually hints at a source.

Inherits or implements a RC The declaring class of the methods inherits from, or implements, a Resource Class, pointing at a possible correlation between the declaring class of the method, the RC and the type of the RC.

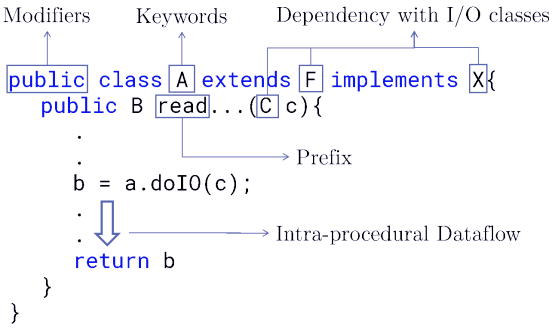

Fig. 2. Summary figure of the feature categories used to classify a method.

Some features might sound naïve at first, like the **Method prefix** one, but it turns out that they are among the ones that correlate the most with the correct classification of the methods, and used in combination with the other features they become more effective [6].

In Figure 2 we have summarized some of the feature classes used by the model to classify a method.

In addition to the feature classes mentioned above, the model also considers a user-assigned *category* for the resource classes, special keywords and prefixes defined in the resource description file (see Figure 1). Such categories are additional features that help the model to better generalise on unseen samples. An example of category (or feature) is `fileParamRCategoryFeature`, which checks whether a method has a file-related resource class among its parameters. The categories used by the model can be added, removed and edited by the user to customise the model according to its needs of detection. The default categories are: *general*, *file*, *log*, *web*, *xml*, *db*, and *gui*. Categories are formally defined in the next section.

E. The model

So far, we have described how the resources are modelled and what the features that we use to feed the methods to a classifier are. In this section we formally introduce the model that completely defines our approach.

Definition 5. *Let a model for classifying sources and sinks be defined as a tuple $U = \langle \mathcal{C}, \mathcal{R}, k, \mathcal{P}, \mathcal{W}, w \rangle$, where:*

- $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ *is the set of resource categories;*
- $\mathcal{R} = \{x \in \Sigma \mid x$ *is a class or interface modeling I/O operations on a resource* $\}$ *in some scene* Σ*;*
- $k : \mathcal{R} \rightarrow \mathcal{C}$ *is a map assigning a category to every resource class;*
- $\mathcal{P} = \mathcal{P}_I \cup \mathcal{P}_M \cup \mathcal{P}_T$*, is the set of prefixes, as defined in Section III-B;*
- $\mathcal{W} = \{x \mid x$ *is a special keyword*$\}$*;*
- $w : \mathcal{W} \rightarrow \mathcal{C}$ *is a map assigning a category to each keyword;*

Note that the model is not tied with the current scene Σ by default. The binding with a particular scene Σ will redefine $\mathcal{R}$ as $\mathcal{R}' = \mathcal{R} \cap \Sigma$, since only resource classes from the current scene can be used to model the I/O.

Definition 6. *Let $f : \Sigma' \rightarrow D$ be a **feature function** that encodes a particular characteristic of a method $m \in \Sigma'$ as $d \in D$, where $D \subset (\mathbb{N} \cup \{Undefined\})$. Then, a **feature set** $\mathcal{F}$ is a set of some, unique, feature functions*

$$\mathcal{F} = \{f \mid f \text{ is a feature function}\}$$

where Σ' is the set of all methods declared by all the classes contained in Σ. A feature set $\mathcal{F}$ encodes each classification instance (method) in a feature space D^n, where $n = |\mathcal{F}|$, by applying each feature function to such an instance[7]. In other words, a feature set can be seen as a function

$$\mathcal{F} : \Sigma' \rightarrow D^n$$

that encodes a method $m \in \Sigma'$ into an element $\boldsymbol{d} = (d_1, d_2, ..., d_n)$ from the vector space D^n such that $d_i = f_i(m)$, $f_i \in \mathcal{F}, 1 \leq i \leq n$. The advantage of defining a feature set as a set is that set notation can be used to easily define the functions composing the feature sets by only defining the classes of features.

For those who would like to dive into the very details of $\mathcal{F}$, we report all the features, and the notation used to define them, in Appendix A.

Feature sets can thus be associated with a source and sink validation model through the set of functions that define them and, therefore, encode methods into vectors $\boldsymbol{d}$, which can be fed to a generic machine learning classifier C.

Figure 3 depicts the complete model, from resource modelling to classification. Note how the mapping functions k and w encapsulate the actual resources and keywords from the feature set $\mathcal{F}_U$. This mapping is extremely important to avoid the classifier directly depending on specific elements of $\mathcal{R}$ and

[7]In our case, $D = \{0, 1, Undefined\}$.

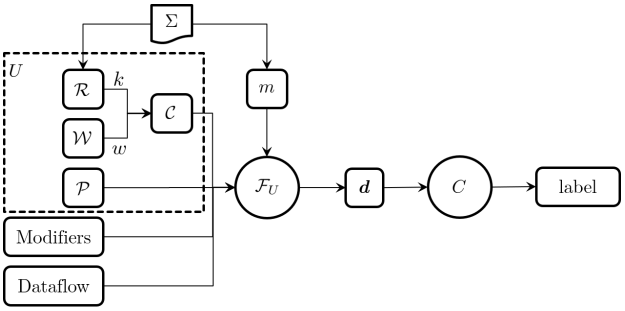

Fig. 3. The Source and Sink Classification Model (SSCM).

$\mathcal{W}$. Additionally, note how $\mathcal{R}$ is depending on the scene Σ for its definition since resource classes must be found in the scene in order to be usable. Generally speaking, the figure shows the encoding of a method $m \in \Sigma'$ into the vector $\boldsymbol{d}$ using all the features defined in $\mathcal{F}_U$.

The advantages of the model U are:

- third-party resources can be added to $\mathcal{R}$, helping the model to correctly recognise more instances without modifying the feature set and, thus, having no need to retrain the underlying machine learning model;
- third-party resources can be automatically mined and added to $\mathcal{R}$ as a preprocessing task, giving space to further automation of the process;
- it preserves a fine-grained representation of the methods, allowing the model to learn the eventual correlation between certain categories and some prefixes or keywords.

The next section reports the results of the validation of the model on real-world Java libraries.

IV. Results

The results obtained from the designed experiments show that the model was able to achieve 86% accuracy and 81% F-measure on our validation set by using a Support Vector Machine [11] as a classifier. The sections below discuss the details of the machine learning model selection and the validation process.

A. Model selection

Choosing and training the right machine learning model for the classification step required some testing and experiments with different configurations of models, features and other small adjustments.

1) Training set creation: Regarding the training set, we sampled 1079 methods from three different libraries: Apache Commons IO, Apache STRUTS2 and Apache HTTP Client, including their dependencies. However, the sampling was not purely random since that would have created a strong imbalance between the number of methods that are neither data sources nor data sinks (*None* class) and the data sources and sinks themselves. For such a reason, we created the dataset by mixing randomly selected methods and keyword-based selected methods. The former step guaranteed that the class distribution of the original population was preserved, while the

TABLE I
THE TRAINING SAMPLES DISTRIBUTION.

Class	Count	Frequency
None	698	64.7
Sink	223	20.7
Source	119	11.0
Mixed	39	3.6
Total	1079	100

latter step guaranteed a sufficient number of training samples for each class. Moreover, the selection was carefully executed in such a fashion to avoid any possible bias derived from the limited set of keywords used for the filtering. The composition of the training set is reported in Table I.

2) Models considered: The final model used by Rasthofer et al. [6] in SuSi was an SVM. However, we could not directly use a Support Vector Machine without checking the performances of other models since our problem was a little different from theirs and we also used a slightly different set of features. Hence, we have tested the following: Decision Tree [12], Support Vector Machines (SVM) [11], k-Nearest Neighbor (k-NN) [13] and the Gradient Boosting [14] ensemble model. Additionally, each model was used in combination with Bagging [15], an ensemble technique to aggregate multiple versions of the same model to create a single model. To test, evaluate and select the models, we used the well-known open-source machine learning framework **scikit-learn** [16] with Python 3.6. The main reason behind this choice was that scikit-learn allows to simply test and evaluate every model with little effort. However, for the validation of the tool, we decided to switch from scikit-learn to WEKA [10], since the models exported through PMML [17] had, on our machines, low-speed performances when invoked by our Java tool.

3) Model selection process: To select the best model among the ones considered, we used ten-fold cross-validation [18] to evaluate the performance of each model. We also performed a discrete search of the hyperparameters' space in order to consider the best instance for each model. The exhaustive search over the hyperparameters space to find the best model can produce a lot of overfitted model instances. To avoid using such instances, we selected the best instance by considering the F-measure value of each instance on both training and test cross-validation folds. An F-measure very close to 100% on the training folds and a test F-measure much lower may indicate overfitting. Thus, we selected the models that generalised the most over both training and test cross-validation folds.

The metrics[8] used to measure the performance of the models are *Accuracy* (A), *Precision* (P), *Recall* (R), and *F-measure* (F), defined as follows:

$$A = \frac{TP + TN}{TP + TN + FP + FN} \qquad R = \frac{TP}{TP + FN}$$
$$P = \frac{TP}{TP + FP} \qquad F = 2 \times \frac{R \times P}{R + P}$$

[8]The metrics are defined in the interval $[0, 1]$ and shown in percentage format. The highest the value the better the performance measured.

TABLE II
AVERAGE PERFORMANCES OF EACH MODEL USING TEN-FOLD CROSS-VALIDATION. BOLD VALUES ARE THE HIGHEST OF THE COLUMN.

Model	Acc. %	Prec. %	Rec. %	F-meas. %
k-NN	88.1	84.6	82.4	82.2
SVM	91.2	84.6	86.1	84.7
Decision Tree	83.1	72.0	70.1	69.8
Grad. Boosting	90.2	**88.0**	80.8	83.0
Bag. k-NN	88.8	86.1	78.2	80.8
Bag. SVM	**92.1**	87.0	**86.9**	**86.4**
Bag. G. Boost.	90.8	88.0	78.7	81.9

where TN is the number of true negatives, TP is the number of true positives, FN is the number of false negatives and FP is the number of false positives.

The scores obtained by the tested models are shown in Table II and are relative to the best instance for each model tested.

Although Bagging of SVM achieves the best scores overall, we decided to proceed for the validation tests with an SVM, due to a simpler configuration of the model and little difference in the performance achieved.

B. Model validation

The model validation is the final step undertook in assessing the validity of the proposed approach.

1) Validation strategy: The validation process started with the design of the experiments to be performed. The first step was to ensure that the method population used to sample the validation set was big enough to represent a real-world scenario, but at the same time, to keep the domain of each library close enough to the domains included in the training set. Otherwise, the tool would have been tested for different capabilities than the ones it was trained for. The criteria used to select such libraries are:

- Be an open-source library largely employed by the open source community, so the real-world performance should not differ from the one measured.
- Include methods and classes from various domains that are usually at risk of injection attacks, such that the validation set has a higher probability of including methods that may be effectively used by attackers.
- The library should have more than 500 methods.

The resulting set of libraries used as the sampling population for the validation set is reported in Table III and contains over 90.000 methods merged from five different libraries: Jersey[9], Hibernate[10], jOOQ[11], the commons file I/O package of Guava[12] and Tinylog[13]. Special attention was paid to avoid any intersection with the training set.

The second step was to classify all the methods in the available population and then sample the validation set using a stratified random sampling that followed the class distribution of the training set rather than the validation set's one. There

[9]Available at https://jersey.github.io/index.html.
[10]Available at http://hibernate.org/.
[11]Available at https://www.jooq.org/.
[12]Available at https://github.com/google/guava.
[13]Available at https://tinylog.org/.

TABLE III
SAMPLING POOL OF THE VALIDATION SET.

Library	Version	Domain	# of Methods
Jersey	2.27	RESTful web-service	10540
Hibernate	5.2.17	Relational Database (SQL)	39024
jOOQ	3.10.7	Relational Database (SQL)	24635
Guava (I/O)	25.0	Common I/O library	700
Tinylog	1.3.2	Logging library	616
		Total	90451

TABLE IV
VALIDATION SET COMPOSITION.

Class	Jersey	Hibernate	jOOQ	Guava	Tinylog	Count
None	36	131	94	1	120	382
Sink	7	94	20	4	25	150
Source	16	38	10	18	4	86
Mixed	0	3	0	2	0	5
Total	59	266	124	25	149	623

were two reasons behind this choice: the first one is that it is a good practice to follow when training machine learning algorithms; the second one is closely related to the first one since the class distribution of the validation set is extremely unbalanced in favor of the *None* class with over 97% of the samples, thus by using a stratified random sampling to sample a reasonable amount of validation samples the other classes would have ended up with less than 10 elements. Table IV reports the composition of the resulting validation set as well as the distribution of the predicted classes for each library. The stratified random sampling strategy performed had the main goal to randomly select at least 50 samples per each class. The *Mixed* class did not have enough samples, but, it was not important since it is a meta-class introduced to cope with special cases.

The next step was a manual and blind inspection of the 623 selected elements, in order to assign a class to each one of them using the name of the method, its documentation (when available), the full name of the declaring class and, eventually, the source code itself as main input.

2) Validation results: The results obtained are reported in Table V. Overall, the model has performed very well, achieving 86.2% *Accuracy* and an average of 85% *Recall* across the classes, meaning there was a low number of false negatives. However, there is a discrepancy between the *Precision* scores obtained during testing (84.6%) and during validation (77.9%). The model has recognised remarkably well the class *None*, but there were a lot of false positives with actual class *Source* and *Sink*. This behaviour can be deduced by noticing the slightly higher number of false negatives of the *None* class (roughly 50) and the number of false positives in the other two classes.

To get further insights into the actual performances of the model, we manually inspected the results and observed the following:

- Some of the *None* methods, belonging to the Hibernate library, were classified as *Sink* because they were actually logging input parameters into a logger, exposing the method to possible information leakage. However, this behaviour was controlled by a debugging flag and, thus, there were no sufficient elements to tag such methods as *Sink*s.
- Keywords, prefixes and their relative categories are the features that help the model to correctly classify methods with very specific domain behaviour when the model cannot rely on the dependencies with Resource Classes and on the dataflow-based features.

TABLE V
CLASSIFICATION SCORES PER EACH CLASS OBTAINED ON THE VALIDATION SET.

Class	Precision %	Recall %	F-Measure %
None	94.0	87.0	90.3
Sink	70.6	83.5	76.5
Source	75.6	86.1	80.5
Mixed	71.4	83.3	76.9
Average	77.9	85.0	81.3

As a final note, the validation results suggest that *none* class is more easily recognisable than the other classes. Domain-specific methods are hard to recognise without prior information on key resource classes of the library. Further insights into this aspect will be presented in the next section.

V. DISCUSSION

A. The internals of the model

In this section, we will perform an in-depth analysis of the category-based features of the model. Such features are among the novel contributions of this work and thus require to be further investigated. Also, the effectiveness of prefix, keyword-based and dataflow features has already been covered by Rasthofer et al. in their work [6].

To understand how the proposed methodology for extracting sources and sinks works internally, we can consider Figure 4 from Appendix A, which depicts the matrix of scatterplots of some of the category-based features of the validation set. This kind of representation allows visualizing the distribution of the classes of a dataset in function of the value assumed by a pair of features.

To explain Figure 4, let us consider the *general* category (abbreviated as *gen* in the figure) as an example, which is the category containing the general I/O resource classes from the Java Framework, like for example the input and output stream abstract classes. The feature *genParamCatF* uses the *general* category to assume a value in $D = \{Undefined, 1, 2\}$ when one of the parameters' type of a method is, inherits, or implements from a class belonging to the *general* category. More precisely, the feature *genParamCatF* assumes the value 2 if at least one parameters' type is a *"relative"* with a class from the *general* category; otherwise it assumes value 1, or $Undefined$ if the method has no parameters. The same idea is valid for *genReturnCatF*, but it is applied to the return type. The concept extends to the other two categories as well – *file* and *web* – and the two related features, which follow a similar naming pattern[14].

[14]The suffix of each feature name, **CatF*, stays for *Cat*egory *F*eature.

As it can be observed from Figure 4, there are pairs of features that discriminate very well between the four classes. Some pairs perform very well only for a particular class, whereas other pairs perform better with different classes. Sinks are discriminated particularly well when *genReturnCatF* is equal to 1, i.e. no general I/O type is returned, and *genParamCatF* is 2, i.e. the method receives I/O types as parameters. Differently, most of the sources are better discriminated when both of them are equal to 1, but also by other pairs, like *genReturnCatF* and *fileParamCatF*, or *genReturnCatF* and *webReturnCatF*.

A careful reader would have noticed that some *None* instances have a return type that is both in the *web* and in the *general* categories. Despite this anomaly, caused by a class implementing interfaces from both categories, we tried to avoid such circumstances when we designed the categories since, as can be seen here, it does not increase the recognition power of the model for the *None* class, indeed, it only decreases it.

The overall set of selected features is designed to allow a machine learning model to learn a discrimination strategy using different subsets of semantical and syntactical features. Thus, by combining category-based features with the other types features, the model is able to identify the correct class of an instance. Moreover, categories are fully customizable and resource classes can be added to them at run-time, enabling the model to easily generalise over methods that use library-specific resource classes. Finally, the customisable nature of categories allows the user to apply the model to different mining purposes, assuming the other features make sense for the problem of interest.

B. Previous prototypes

The model presented in this work had two predecessor prototypes that used a different set of features each. We briefly describe them to give the reader some insights into the evolution of the presented model.

The first prototype was based on a set of fewer than fifteen features. However, the domain of such features was not binary, but multi-valued. Features used a different approach to encode the prefix type, the dataflow, and the types of the return value and of the parameters in a very different way from its successor. Nevertheless, the model's performance was not satisfactory. Thus, we decided to switch to a binary feature space and use more fine-grained features. This change resulted in a more rigid model with over 200 features that reduced the capability of the model to generalize over new library types. However, the performance drastically increased.

Hence, we hypothesised that there must be a sweet spot between fine-grained and coarse-grained features that are flexible enough to adapt to different sources and sinks types while maintaining high performances. Although the model presented by this work is a step forward in that direction, it still needs further work to identify the, possibly new, features that would allow us to move towards that sweet spot.

VI. Possible issues, limitations and weaknesses

Although the developed tool, as shown in the previous section, is fairly precise in identifying sources and sinks, there are still some issues and limits.

1) Large feature space: The number of features is pretty high, thus it is hard to produce a training set that has a good number of samples that covers most of the cases. This limit is very hard to overcome and requires either a large amount of data, or a reduction of the feature space dimensions without impacting performance. Moreover, there are also other types of I/O domains that haven't been considered in this model, such as others vulnerable network communication protocols. Each one of them might use different prefixes, keywords, and categories which might result in new features to be added to the model, increasing the amount of learning required.

2) Troublesome tagging: The manual classification of the methods is a tedious task that requires a lot of time. Indeed, for each method a researcher has to first find if there is any available documentation of the current method, then he has to check that the documentation of the method is relative to the same version of the method in the dataset, then he reads the documentation or, if necessary, the code and assigns a class. Moreover, correctly tagging the methods requires a deep understanding of all the libraries involved, which is quite difficult to achieve. Because of these factors, there is a high risk of introducing errors during the process, which might affect both the training and the validation process. For such reasons, we have used only Apache libraries for the training set since they were easy to understand and well known to the authors. Such a choice, of course, has limited the generality of the training set.

3) Obfuscation: The proposed approach assumes that the libraries under analysis are not obfuscated and all the names and classes are in English. The former issue cannot be overcome and the performances of the tool on an obfuscated library are drastically reduced. Nonetheless, the latter issue can be overcome by manually switching the resource description to the necessary language.

4) Flexibility: Another limit with the proposed model is the inability to change its behaviour, without adding more training samples, when it misclassifies *None* methods as non-*None* methods (false positives).

5) Validation process: Although our validation process considers a high amount of methods as a sampling pool, the number of libraries used might not be enough to completely ensure a total coverage of all the possible methods types existing.

Nevertheless, our tests and validation experiments have shown that the predictions of the tool can be considered reliable when performed on libraries with a similar domain of the libraries used in the training set.

VII. Conclusion and Future Developments

A. Conclusions

In this paper, we have shown that, under certain conditions, data sources and data sinks exhibit distinctive traits shared

across multiple libraries that allow a machine learning model to identify them.

The results obtained are very encouraging, and demonstrate that this kind of approach can be shaped and fine-tuned to further increase the recognition accuracy of the model.

We believe that a similar approach, properly modified, could be applied to other domains in order to identify methods that exhibit certain types of behaviour and interactions with the external environment of the process being executed.

The implementation used in this work is available online[15].

B. Future Developments

Future works may concern a deeper analysis and further improvement of the concepts acquired throughout the testing and development of the current work.

The model allows the user to apply different changes on how the classification works without training again the machine learning model. By exploiting this feature to calculate at runtime the Resource Classes specific to each input received, the model automatically adapts the classification to the library it is analyzing. However, this implies that the model should be able to automatically identify Resource Classes, which is not a trivial problem.

Nevertheless, there are several ways to approach the problem, but we suppose that a semantic-oriented approach may be the most appropriate. Several third-party resources share similar keywords across the different components they are made of. The class name, the package name, the documenting comments, the name of the declared methods and of the implementing interface can all help to identify a correct category for the resource under analysis.

An example of a very naïve approach to the problem could be to use the edit distance between the methods and class names from the initial classes in $\mathcal{R}$. The closer a new instance is to a class in one specific category, the more likely it will be to get assigned to such a category. More generally, the *semantic similarity* of the context in which a class is defined can be compared with a pre-defined set of terms or classes using a certain policy. Cutting-edge research in the literature is already studying how to integrate Natural Language Processing and Ontologies [19] to Software Engineering problems like this.

References

[1] P. IBM Security, "2017 Cost of Data Breach Study – Global Overview," www.ibm.com/security/data-breach, 2017, accessed: 29-01-2018.

[2] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. L. Traon, "Static analysis of android apps: A systematic literature review," *Information & Software Technology*, vol. 88, pp. 67–95, 2017.

[3] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, Jun. 2014.

[15] See https://bitbucket.org/darius-sas/sscm.

[5] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USENIX Association, 2010, pp. 393–407.

[6] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[7] S. Rasthofer, "Improving mobile-malware investigations with static and dynamic code analysis techniques," Ph.D. dissertation, Darmstadt University of Technology, Germany, 2016.

[8] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," in *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings*. Springer, 2012, pp. 291–307.

[9] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1999, pp. 13–.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, Nov. 2009.

[11] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep 1995.

[12] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

[13] N. S. Altman, "An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[14] J. H. Friedman, "Greedy function approximation: A gradient boosting machine." *Ann. Statist.*, vol. 29, no. 5, pp. 1189–1232, 10 2001.

[15] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, Aug. 1996.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[17] R. Grossman, S. Bailey, A. Ramu, B. Malhi, P. Hallstrom, I. Pulleyn, and X. Qin, "The management and mining of multiple predictive models using the predictive modeling markup language," *Information and Software Technology*, vol. 41, no. 9, pp. 589–595, 1999.

[18] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143.

[19] R. Witte, Q. Li, Y. Zhang, and J. Rilling, "Text mining and software engineering: an integrated source code and document analysis approach," *IET Software*, vol. 2, no. 1, pp. 3–16, February 2008.

Appendix A
Feature functions

This appendix lists the feature functions that define the feature set $\mathcal{F}_U$ as defined in Section III-E.

A. Helper functions

The helper functions, required to define the feature functions listed in the next section, are defined below:

- $\phi : \Sigma \rightarrow \mathcal{R} \cup \perp$ maps a class in the scene with the highest hierarchy Resource Class it implements or inherits from. The $\perp$ value is used if the class implements no Resource Class, while R is the set of the resource classes considered by the model;
- $class : \Sigma' \rightarrow \Sigma$ returns the class of a method;
- $prefix(s, x)$ returns $true$ if x is prefix of s;
- $substr(s, x)$ returns $true$ if x is substring of s;

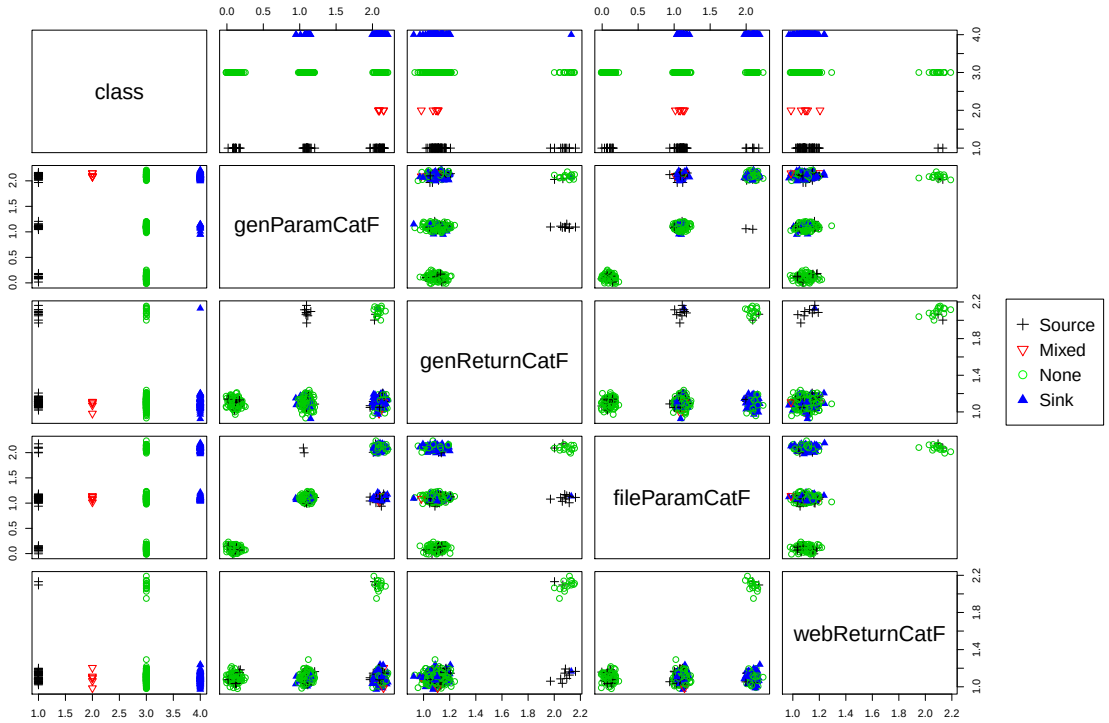

Fig. 4. The matrix scatterplot of some category features of the validation set. Features are mapped into $D = [0, 2] \subset \mathbb{N}$ with additional jitter to help for visualisation. For the feature columns: 0 is $Undefined$, 1 is $False$ and 2 is $True$. For the class column: 1 is *Source*, 2 is *Mixed*, 3 is *None*, 4 is *Sink*. (Best with colors.)

- $param : \Sigma' \to \wp(\Sigma)$ returns the types of the parameters of a method;
- $ret : \Sigma' \to \Sigma$ returns the return type of the method;
- $d : E \times \Sigma' \to \{true, false\}$ is an intra-procedural dataflow function, evaluating $true$ whenever $e \in E$ holds for $m \in \Sigma'$. E is the set of dataflow endpoints containing the following pairs:
  - $\langle p, this \rangle$ dataflow endpoint from any parameter p to $this$;
  - $\langle p, s \rangle$ dataflow endpoint from any parameter p to a Resource Method invocation statement s;
  - $\langle s, ret \rangle$ dataflow endpoint from any R.M. invocation statement s to the return statement ret;
  - $\langle s, this \rangle$ dataflow endpoint from any R.M. invocation statement s to $this$;
  - $\langle s, p \rangle$ dataflow endpoint from any R.M. invocation statement s to any parameter p;
- $\overline{\mathcal{P}} = \{\mathcal{P}_I, \mathcal{P}_T, \mathcal{P}_M\}$.

B. Feature functions

Let $\mathcal{F}_U$ be the feature set[16] of U, defined as the union of the following sub-feature sets:

$$\mathcal{F}_U^1 = \{f_p \mid f_p(m) = 1 \text{ iff } prefix(m, p) = true, \; m \in \Sigma', \; p \in \mathcal{P}\}$$

$$\mathcal{F}_U^2 = \{f_p \mid f_p(m) = 1 \text{ iff } m \text{ invokes a RM } n \text{ s. t. } prefix(n, p) = true, \; m, n \in \Sigma', \; p \in \mathcal{P}\}$$

$$\mathcal{F}_U^3 = \{f_P \mid f_P(m) = 1 \text{ iff } prefix(m, p) = true \text{ for some } p \in P \text{ with } m \in \Sigma', \; P \in \overline{\mathcal{P}}\}$$

$$\mathcal{F}_U^4 = \{f_c \mid f_c(m) = 1 \text{ iff } k(\phi(class(m))) = c \text{ with } m \in \Sigma', \; c \in \mathcal{C}\}$$

$$\mathcal{F}_U^5 = \{f_c \mid f_c(m) = 1 \text{ iff } k(\phi(p)) = c \text{ for some } p \in param(m) \text{ with } m \in \Sigma', \; c \in \mathcal{C}\}$$

$$\mathcal{F}_U^6 = \{f_c \mid f_c(m) = 1 \text{ iff } k(ret(m)) = c \text{ with } m \in \Sigma', \; c \in \mathcal{C}\}$$

$$\mathcal{F}_U^7 = \{f_c \mid f_c(m) = 1 \text{ iff } substr(class(m), x) = true \text{ and } w(x) = c \text{ for some } x \in \mathcal{W} \text{ with } m \in \Sigma', \; c \in \mathcal{C}\}$$

$$\mathcal{F}_U^8 = \{f_e \mid f_e(m) = 1 \text{ iff } d(e, m) = true, \; e \in E\}$$

$$\mathcal{F}_U^9 = \{f_s \mid f_s(m) = 1 \text{ iff } class(m) \text{ has modifier } s, \text{ with } m \in \Sigma', \; s \in \{interface, abstract\}\}$$

$$\mathcal{F}_U^{10} = \{f_s \mid f_s(m) = 1 \text{ iff } m \text{ has modifiers with } m \in \Sigma', \; s \in \{abstract\}\}$$

More precisely:

$$\mathcal{F}_U = \bigcup_i^{10} \mathcal{F}_U^i$$

With a total number of features equal to $|\mathcal{F}_U| = 2|\mathcal{P}| + |\overline{\mathcal{P}}| + 4|\mathcal{C}| + |E| + 3$.

[16] See Definition 6 from Section III-E.