2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM) | 978-1-6654-4897-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/SCAM52516.2021.00030

# SootFX: A Static Code Feature Extraction Tool for Java and Android

1st Kadiray Karakaya
*Heinz Nixdorf Institute*
*Paderborn University*
Paderborn, Germany
kadiray.karakaya@upb.de

2nd Eric Bodden
*Heinz Nixdorf Institute*
*Paderborn University & Fraunhofer IEM*
Paderborn, Germany
eric.bodden@upb.de

***Abstract*—Static code features are necessary components when using machine learning-based techniques to reason about a program of interest. To extract static code features, researchers develop their own feature extractors specific to their own studies. This causes two problems for the follow-up studies that build on the same set of features. First, the current feature extractors are intertwined with the rest of their codebases, and accessing them alone is time-consuming. Second, new kinds of features that are introduced in the follow-up studies are not incorporated back into the original feature extractors. Therefore, it is a tedious task for researchers to track all these individual feature extractors from different projects. In this work, we present SootFX, a generic stand-alone tool that enables the extraction of static code features from Java and Android programs. We explain its design, which makes it easily extensible for supporting new features and resource providers. We introduce its client APIs in Java as well as in Python, a popular programming language among machine learning practitioners. We illustrate a few of its possible use cases on a set of real-world Java libraries and Android applications.**

***Index Terms*—feature extraction, program analysis, machine learning, java, android**

## I. Introduction

Static program analysis reasons about interesting properties of a target program, without executing it [1]. Although it has been practically applied over many years for program optimization [2], error detection [3] or finding security vulnerabilities [4], the application of static program analyses to real-world programs in a precise and scalable manner is still considered an open challenge. In search of overcoming this challenge, researchers employ machine learning (ML) based techniques, either alone or to support traditional control-flow-based analyses. When used alone, these techniques usually aim to obtain as many features as possible, and then select the most relevant features for their goals. Common use cases include, but not limited to, defect prediction [5], malware detection [6] [7], vulnerability detection [8], and code quality assessment [9]. On the other hand, some of the recent approaches make use of ML techniques to support traditional static analyses. Heo et al. [10] aims to classify specific program locations where increased analysis precision, in terms of flow-, context-sensitivity and widening thresholds, would be applicable. Heo et al. [11] finds out a balance between a sound and an unsound analysis for different methods. Rasthofer et al. [12] and Piskachev et al. [13] classify arbitrary methods as sources and sinks, which are required for taint analyses. Whether they are used alone or to support static analyses, all of the ML-based approaches rely on the extraction of static code features from target programs.

A common goal when using ML-based approaches is to achieve a relatively better performance than the existing studies, in terms of precision and recall. This might require applications of novel feature selection techniques or classification algorithms to existing studies. For that, researchers would only need the extracted features but not necessarily the rest of the codebase from the existing studies. Therefore, they need to obtain only the feature extraction module from existing projects, that they want to improve on. However, not all of the project implementations are open source or accessible [7], and even if they are [6], their feature extractors are usually intertwined with the rest of their codebase, therefore it is not easy to *integrate* them into new ML pipelines for the follow-up studies. This causes the researchers to spend a substantial amount of time, either on decomposing the feature extractors from the rest of their codebase or on building their own feature extractors from scratch.

An inevitable necessity when using static code features is the maintenance of the feature extractor by adding support for new features. For instance, Arp et al. [7] introduced Drebin in 2014 as a method for detecting Android malware based on the features that they extract from apps' manifest and code. Among other features, Drebin makes heavy use of app permissions and related API calls. However, since its introduction, available permissions changed a lot with each new Android API level. Alone after 2020, Android has introduced 8 new permission with the API level 30, and 17 new permission with the API level 31 [14]. Incorporating these changes would require, at the minimum, the *extension* of manifest and method features to handle permissions and related API calls.

To address the above-mentioned *integration* and *extensibility* issues, we present SoofFX[1]. To provide easy integration, SootFX offers powerful client APIs that allow it to be easily integrated into ML pipelines. To enable extensibility, SootFX

[1] Available at https://github.com/secure-software-engineering/SootFX

978-1-6654-4897-0/21/$31.00 ©2021 IEEE
DOI 10.1109/SCAM52516.2021.00030



Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on September 08,2022 at 13:11:52 UTC from IEEE Xplore. Restrictions apply.

was built with a modular architecture that makes it easily extended for supporting new features and resource providers.

Specifically, our contributions in this paper are as follows:

- SootFX, a generic stand-alone tool that enables the extraction of static code features from Java and Android applications, as well as non-code features such as Android manifests.
- A set of built-in feature extraction units (FEUs) that correspond to some of the selected features from the related work.
- Illustration of a few of its possible use cases on real-world Java libraries and Android applications.

This paper is organized as follows, in Section II, we discuss the related work. In Section III, we introduce the design and implementation of SootFX. In Section IV, we present the initial evaluation results and use cases. In Section V, we conclude with a discussion and the possible future work.

## II. Related Work

Although many ML-based studies implement static code feature extractors, their main goal is not particularly making these feature extractors reusable. Therefore, in this section, we only consider the works that were built specifically as feature extraction tools.

Reif et al. [15] implemented Hermes for extracting static code features from Java bytecode. Hermes assesses the representitiveness of Java benchmark corpora by finding number of unique features they have. It is built on OPAL Java bytecode analysis framework [16]. Therefore, it can also extract static analysis-based features, such as control- and data-flow features, but it does not support Android at all. The majority of the features that it extracts are about testing the existence of Java bytecode instructions. Hermes can be extended to support new bytecode features, but it does not aim to be extensible in terms of supporting various resource providers other than OPAL.

Zhao et al. [17] presented Fest for extracting features from Android apps with the purpose of detecting malware. Fest is built on the Android decompilation tool, Apktool [18]. It can extract features from Android manifest files, as well as from application code. It can detect the existence of specific API calls or URLs. However, it is not able to extract nontrivial features such as control- and data-flow features, because it does not use any static analysis framework. Fest does not aim to be an extensible tool in terms of supporting different resource providers and features.

## III. SootFX

While designing SootFx, we aimed to achieve two high-level goals, ease of integration and ease of extension. To achieve ease of integration, SootFX was developed to be a stand-alone tool, whose only responsibility is to extract features from a target program. It provides a powerful Java API that enables flexible compositions to extract the desired set of features. Further, we extend its API to Python and provide a Python client, that enables obtaining the extracted features directly as Pandas [19] DataFrame object. Moreover, it allows the generation of CSV files that can potentially be used in any tool or framework, independent of any specific programming language.

To achieve ease of extension, SootFX was designed with a modular architecture where the resource providers and FEUs can be extended independently. Although its current built-in FEUs are primarily using the Soot framework [20] as a resource provider for Java and Android code, the core functionality is totally generic. By integrating various resource providers, a new set of features can be easily supported, as we also show by using FlowDroid [21] as another resource provider to extract Android manifest features.

### A. Architecture

Figure 1 shows the high-level overview of SootFX's architecture. The basic building blocks of the architecture are a unified Java API (*SootFX API*), resource providers (*Manifest Resource Provider*, *Code Resource Provider*), and feature extraction managers (*MethodFX*, *ClassFX*, *WholeProgFX*, *ManifestFX*). *SootFX API* provides client-facing methods for accessing the core functionality. The API takes as input a *Target Program*, and a set of *Desired Features (DF)*. The *Target Program* can be a directory path to Java class files, JARs, or APKs. The set of all available features can be queried via the API. The set of desired features can be individually selected from the list of built-in features, or all available features will be extracted if specific features are not selected.

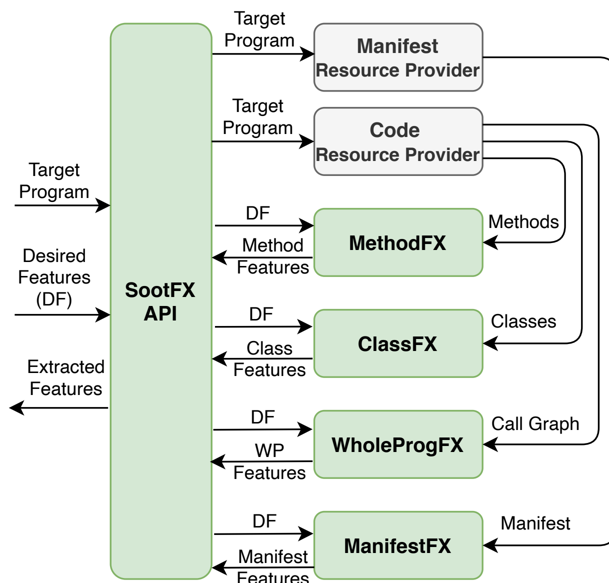



Fig. 1: Architecture of SootFX

Resource providers can be any external tool that supplies a *feature target*. In principle, they can simply be considered as parsers that operate on the given file. We use the Soot framework [20] as a code resource provider, that supplies *Methods*, *Classes*, and the *Call Graph* of the *Target Program*



Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on September 08,2022 at 13:11:52 UTC from IEEE Xplore. Restrictions apply.

as feature targets. In addition, we use FlowDroid [21] as a manifest resource provider, that supplies Android *Manifest* as a feature target. These feature targets are then handed over to individual FEUs via feature extraction managers.

Feature extraction managers, facilitate the connection between the resource providers and FEUs. Each feature extraction manager activates the set of selected FEUs that operate on the same feature target. For instance, *MethodFX* will activate the feature extractors that take as feature target a Method object, and aggregate the extracted method features and pass them to the *SootFX API*.

### B. Built-in Feature Extraction Units

In this section, we present the types of currently available FEUs. Each type of FEU is managed by a corresponding feature extraction manager. We classify the feature extraction managers concerning their result representations into two categories as multi-instance and single-instance. Multi-instance feature extraction managers return sets of extracted features on multiple instances of a feature target. For instance, in the case of method feature extraction, the result contains all the method instances of a target program. On the other hand, single-instance feature extraction managers return a set of extracted features on the single instance of the feature target, e.g. a target Android application contains only a single manifest file, hence extracted features correspond to the singleton instance of the manifest.

| Type | Count | Sample FEUs | Value |
|---|---|---|---|
| Method Signature | 34 | MethodNameContains | B |
| | | MethodReturnsVoid | B |
| | | MethodAccessModifier | C |
| Method Body | 9 | MethodReturnsConstant | B |
| | | MethodBranchCount | N |
| | | MethodCallsMethod | B |
| Method Data-flow | 3 | MethodParamFlowsToMethod | B |
| | | MethodParamFlowsToReturn | B |
| Class Signature | 11 | ClassNameContains | B |
| | | IsStaticClass | B |
| | | ClassAccessModifier | C |
| Class Body | 4 | ClassFieldCount | N |
| | | ClassContainsMethod | B |
| Whole Program | 6 | WholeProgramMethodCount | N |
| | | WholeProgramStmtCount | N |
| Manifest Permission | 184 | ManifestPermissionCount | N |
| | | ManifestPermReadContacts | B |
| Manifest Hardware | 61 | ManifestUsesHWCount | N |
| | | ManifestUsesHWTelephony | B |
| Manifest Software | 17 | ManifestUsesSWCount | N |
| | | ManifestUsesSWDeviceAdmin | B |
| **Total** | 329 | | |

TABLE I: Built-in Feature Extraction Units

Table I shows the built-in FEU types, along with the count of currently implemented FEUs that belong to each type, selected sample FEU instances, and the value types that belong to each feature that is being extracted by FEUs. Currently extracted feature value types are numeric (N), categorical (C), or binary (B).

Below we list the currently implemented FEU types by their result representation categories:

- Multi-Instance:
  - Method Signature: Method modifiers, name, parameters, return type.
  - Method Body: Statements in a method, e.g. assignments, branches, method invocations.
  - Method Data-flow: Obtained by applying intra-procedural data-flow analysis on a method's control-flow graph.
  - Class Signature: Class modifiers, name.
  - Class Body: Fields and methods in a class.
- Single-Instance:
  - Whole Program: Obtained from call graphs. E.g. number of actual method calls[2], or the number of reachable statements in a program. In principle, it also enables the extraction of inter-procedural features.
  - Manifest Permission: List of requested permissions.
  - Manifest Hardware: List of used hardware features.
  - Manifest Software: List of used software features.

### C. Integration

Figure 2 shows the integration capabilities of SootFX. Its *Java API* can directly be used within a Java project by adding it as a Maven dependency. As an alternative, we have also implemented a *Python API*, with that we aim to make SootFX be easily integrated into Python-based ML pipelines. The *Python API* works as a proxy of the *Java API*, their communication is facilitated by implementing a Py4J [22] gateway. Additionally, we also provide a command-line interface (*CLI*), in that case, the set of features to be included or excluded can be defined in a file. The extracted features can be obtained, depending on the preferred client API, as *FeatureSet* or Pandas *DataFrame* objects. It is also possible to generate *CSV* files with any of its APIs.

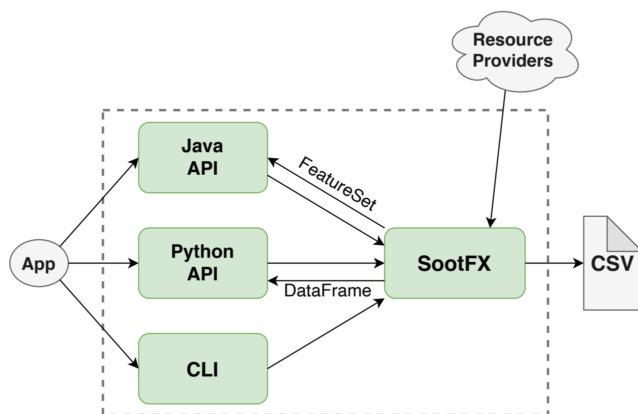



Fig. 2: Integration Capabilities of SootFX

Table II shows the API methods that a client can call for the case of method feature extraction. *listAllMethodFeatures*

[2]Exact number depends on the call graph algorithm



Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on September 08,2022 at 13:11:52 UTC from IEEE Xplore. Restrictions apply.

returns a list of *Feature Description* objects, which contain natural language descriptions of all the available method features, as well as their names. Feature names are automatically obtained by using the Java reflection API, it suffices that the user implements the corresponding feature extraction unit interface. *extractAllMethodFeatures* runs all the available method FEUs and returns a set of method features for each method instance in the target program. Additionally, the user can obtain the desired set of features by specifying them via the inclusion-based *extractMethodFeaturesInclude* method, or she can filter out a list of features by using the exclusion-based *extractMethodFeaturesExclude* method.

| Method Name | Input | Output |
|---|---|---|
| listAllMethodFeatures | - | List of Feature Description |
| extractMethodFeaturesInclude | Features List to Include | Extracted Features |
| extractMethodFeaturesExclude | Feature List to Exclude | Extracted Features |
| extractAllMethodFeatures | - | Extracted Features |

TABLE II: API Methods for Method Feature Extraction

### D. Extension

In this section, we describe the base classes and interfaces of SootFX that enable the easy extension with FEUs that can operate on diverse feature targets. We demonstrate how one can implement a new FEU that takes as input a method and extracts the number of branches in it. The core functionality of SootFX is implemented with Java generics, it is agnostic of any resource providers and FEUs. Therefore it can be extended with arbitrarily many resource providers and FEUs that process the feature targets obtained from these resource providers. The listing 1 shows the generic `Feature` class, that can be parameterized with the type of the feature value (`V`).

Listing 1: Generic Feature Class

```
class Feature<V> {

    String name;
    V value;
}
```

Listing 2 shows the generic `FeatureExtractionUnit` interface, that can be parameterized with the type of feature value (`V`) and feature target (`T`). As a convention, we use the class name of each FEU as the name of the corresponding extracted feature. Therefore, `FeatureExtractionUnit` interface contains a default implementation of `getName()` method.

Listing 2: Generic FeatureExtractionUnit Interface

```
interface FeatureExtractionUnit<V,T> {

    default String getName(){
        return this.getClass()
                .getSimpleName();
    }

    Feature<V> extract(T target);
}
```

All the method FEUs implement the `MethodFEU` interface shown in the Listing 3, method FEUs take as feature target a `SootMethod`, Soot's [20] method representation object, and its feature value (`V`) can be parameterized, for instance, with *Long* for numeric, *Boolean* for binary, *String* for categorical values.

Listing 3: Generic MethodFEU Interface

```
interface MethodFEU<V> extends
    FeatureExtractionUnit<V, SootMethod> {
}
```

Listing 4, shows a concrete implementation of `MethodFEU`, which extracts number of branching statements from a `SootMethod`. We suggest that the features should be named by using the default `getName()` method, which is implemented in the `FeatureExtractionUnit` interface.

Listing 4: An Implementation of MethodFEU

```
class MethodBranchCount implements
    MethodFEU<Long> {

    @Override
    public Feature<Long> extract(SootMethod
        target) {

        long n =
            target.getActiveBody()
                  .getUnits()
                  .stream()
                  .filter(Unit::branches)
                  .count();

        return new Feature<>(getName(), n);
    }
}
```

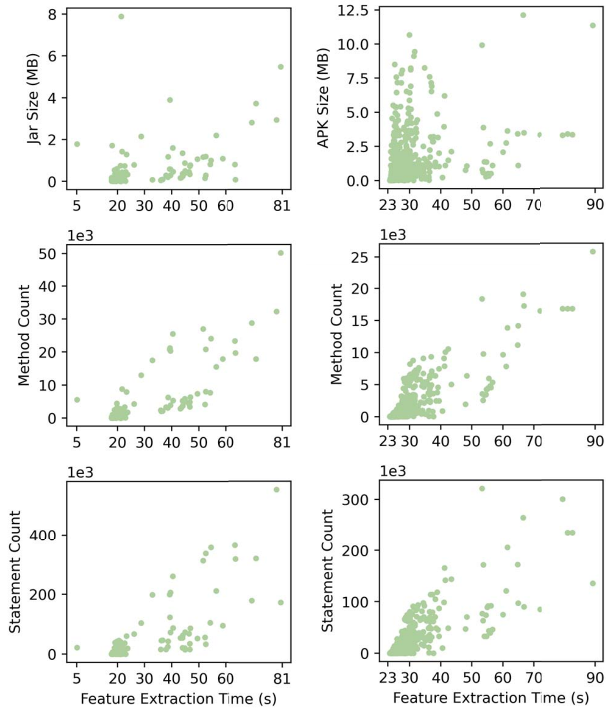
## IV. Evaluation

In this section, we present run-time performance of SootFX, when running it on real-world Java libraries and Android apps, then we show how it can be used for two example use cases. In total, we collected 112 JARs from the most popular libraries listed in the maven repository [23], that belong to well-known projects such as Apache Commons [24], Google Guava [25], Spring Framework [26], Junit [27]. We selected 700 APK files from the Drebin [7] dataset, which contains Android malware apps. We conducted the experiments on a Macbook Pro with a Quad-Core Intel i7 processor at 2,3 GHz and 32 GB memory.

Figure 3 shows the run-time of SootFX when applying all the built-in FEUs on JAR and APK files. APK run-times additionally contain the time spent on extracting manifest features, which varies between 5.8 and 20.4 seconds with a mean of 7 seconds. Feature extraction times usually correlate with the size of the given APK or JAR, however, they are also affected by the number of methods in each program and the number of statements in each method.



Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on September 08,2022 at 13:11:52 UTC from IEEE Xplore. Restrictions apply.




(a) Java Libraries (N=112) (b) Android Apps[3](N=700)

Fig. 3: Feature Extraction Run-time

**Code readability assessment:** As a first use case, we discuss extracting method features for evaluating code readability of Java libraries. For this purpose we consider, the number of total statements[4] in a method along with the number of branches and assignments based on features listed in [28]. For instance, when we consider the method with the most statements in Google Guava library, which is the static initializer of the static class `Crc32cHasher` in `Crc32cHashFunction` of package `com.google.common.hash`, all of its 258 statements are actually assignment statements. Albeit its *subjectively* poor readability, it is still rather readable when compared to the method with the most branches, `retryUpdate()` method in `Striped64` of package `com.google.common.cache` which contains 147 statements among which 45 are branches. As another example, when we look at the method with the most statements in Spring framework, which is the `accept()` method in `ClassReader` of package `org.springframework.asm`, it contains 982 statements, among which 256 are branches, which we subjectively find hardly. We can conclude that the number of branches contained in a method is a good feature candidate for assessing code readability.

[3] Only contains malware apps from Drebin

[4] Number of statements in the source code might differ, as we analyze bytecode

**Android malware classification:** As a second use case, we discuss extracting manifest features for Android malware classification. For this purpose, in addition to the malware apps from the Drebin dataset, we selected 6 popular Android apps from the Google Play store [29] as benign apps: Ebay, Twitter, Amazon, Instagram, WhatsApp, Cash App. As usual in many of the Android malware classification studies ([6], [7]), we considered interpreting the used permissions feature. Table III shows the permissions that are rarely requested both by malware and benign apps. We find it surprising that Malware apps do not usually request *Camera* and *GetAccounts* permissions. On the other hand, Benign apps usually do not request SMS-related permissions such as *SendSms*, *ReceiveSms*, *ReadSms*.

| Permission | Level | % Benign Apps Use | % Malware Apps Use |
|---|---|---|---|
| Camera | Dangerous | 100 | 5 |
| GetAccounts | Dangerous | 85 | 9 |
| SendSms | Dangerous | 17 | 54 |
| ReceiveSms | Dangerous | 17 | 38 |
| ReadSms | Dangerous | 0 | 38 |

TABLE III: Permissions Rarely Used by both Benign and Malware Apps

Table IV shows the permissions that are frequently requested by both kinds of apps. It is expected that the protection levels of commonly requested permissions mostly are of the *Normal* level. However, since some permissions with *Dangerous* protection levels are only requested by benign apps, one must consider exploiting additional features.

| Permission | Level | % Benign Apps Use | % Malware Apps Use |
|---|---|---|---|
| Internet | Normal | 100 | 95 |
| AccessNetworkState | Normal | 100 | 67 |
| WriteExternalStorage | Dangerous | 100 | 65 |
| WakeLock | Normal | 100 | 40 |
| ReadPhoneState | Dangerous | 84 | 87 |
| ReceiveBootCompleted | Normal | 84 | 48 |
| AccessWifiState | Normal | 67 | 45 |

TABLE IV: Permissions Frequently Used by both Benign and Malware Apps

**Discussion:** The use cases and the run-time evaluation were presented to motivate the practical usefulness of SootFX, they do not directly evaluate our high-level goals of ease of integration and extensibility. We show that the set of features that SootFX can currently extract, provides a good starting point for future research that utilizes static code features. Moreover, we can conclude that SootFX can perform reasonably well when processing real-world applications.

## V. Conclusion and Future Work

Researchers use ML-based approaches to reason about interesting program properties, for that they usually build their own feature extractors. Their main motivation is to achieve better precision or recall, but not necessarily the reusability of the feature extractors. However, when new algorithms or feature



Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on September 08,2022 at 13:11:52 UTC from IEEE Xplore. Restrictions apply.

selection approaches are applied to increment on an existing study, they need to be applied to the same set of features for correct comparability. Therefore, it is important to obtain existing feature extractors and integrate them into the follow-up studies. Besides, feature extractors need to be continuously extended to keep up with the new features introduced to the target programs. Therefore, we identified two drawbacks of the existing feature extractors regarding their integration and extensibility capabilities. We built SootFX to be easily integrated into various ML pipelines with its CLI, and client APIs in Java and Python. Furthermore, we designed it to be easily extensible with new resource providers and feature extraction units (FEUs). To motivate its usefulness, we run it on a set of real-world Java libraries and Android applications. We demonstrated two possible use cases where its built-in FEUs can be useful.

SootFX is built to be an open-source project, which we envision to be extended with new resource providers and FEUs with the feedback and contribution of researchers and engineers. However, as a first step, we will implement *feature groups* that enable combining custom of FEUs and assigning them unique identifiers for future reference and sharing for reproducibility.

## REFERENCES


[1] A. Møller and M. I. Schwartzbach, "Static program analysis," *Notes. Feb*, 2012.

[2] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '73. New York, NY, USA: Association for Computing Machinery, 1973, p. 194–206. [Online]. Available: https://doi.org/10.1145/512927.512945

[3] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "On the value of static analysis for fault detection in software," *IEEE Transactions on Software Engineering*, vol. 32, no. 4, pp. 240–253, 2006.

[4] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *14th USENIX Security Symposium (USENIX Security 05)*. Baltimore, MD: USENIX Association, Jul. 2005. [Online]. Available: https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static

[5] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010.

[6] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[8] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[9] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[10] K. Heo, H. Oh, H. Yang, and K. Yi, "Adaptive static analysis via learning with bayesian optimization," *ACM Trans. Program. Lang. Syst.*, vol. 40, no. 4, Nov. 2018. [Online]. Available: https://doi.org/10.1145/3121135

[11] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided selectively unsound static analysis," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 519–529.

[12] 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*. The Internet Society, 2014.

[13] G. Piskachev, L. N. Q. Do, and E. Bodden, "Codebase-adaptive detection of security-relevant methods," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 181–191. [Online]. Available: https://doi.org/10.1145/3293882.3330556

[14] "Manifest.permission - android developers," https://developer.android.com/reference/android/Manifest.permission, (Accessed on 07/29/2021).

[15] M. Reif, M. Eichberg, B. Hermann, and M. Mezini, "Hermes: Assessment and creation of effective test corpora," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 43–48. [Online]. Available: https://doi.org/10.1145/3088515.3088523

[16] M. Eichberg and B. Hermann, "A software product line for static analyses: The opal framework," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, ser. SOAP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–6. [Online]. Available: https://doi.org/10.1145/2614628.2614630

[17] K. Zhao, D. Zhang, X. Su, and W. Li, "Fest: A feature extraction and selection tool for android malware detection," in *2015 IEEE Symposium on Computers and Communication (ISCC)*, 2015, pp. 714–720.

[18] "Apktool - a tool for reverse engineering 3rd party, closed, binary android apps." https://ibotpeaches.github.io/Apktool/, (Accessed on 08/01/2021).

[19] W. McKinney *et al.*, "pandas: a foundational python library for data analysis and statistics," *Python for high performance and scientific computing*, vol. 14, no. 9, pp. 1–9, 2011.

[20] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[21] 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," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[22] "Welcome to py4j — py4j," https://www.py4j.org/, (Accessed on 07/29/2021).

[23] "Maven repository: Search/browse/explore," https://mvnrepository.com/, (Accessed on 08/01/2021).

[24] "Apache commons – apache commons," https://commons.apache.org/, (Accessed on 08/01/2021).

[25] "Guava," https://guava.dev/, (Accessed on 08/01/2021).

[26] "Spring framework," https://spring.io/projects/spring-framework, (Accessed on 08/01/2021).

[27] "Junit 5," https://junit.org/junit5/, (Accessed on 08/01/2021).

[28] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[29] "Google play," https://play.google.com/store, (Accessed on 08/01/2021).




Authorized licensed use limited to: UNIVERSITÄTSBIBLIOTHEK PADERBORN. Downloaded on September 08,2022 at 13:11:52 UTC from IEEE Xplore. Restrictions apply.