Obfuscation-Resilient Code Recognition in Android Apps

Johannes Feichtner
Graz University of Technology
Secure Information Technology Center – Austria (A-SIT)

Christof Rabensteiner
Graz University of Technology

ABSTRACT

Many Android developers take advantage of third-party libraries and code snippets from public sources to add functionality to apps. Besides making development more productive, external code can also be harmful, introduce vulnerabilities, or raise critical privacy issues that threaten the security of sensitive user data and amplify an app's attack surface. Reliably recognizing such code fragments in Android applications is challenging due to the widespread use of obfuscation techniques and a variety of ways, how developers can express semantically similar program statements.

We propose a code recognition technique that is resilient against common code transformations and that excels in identifying code fragments and libraries in Android applications. Our method relies on obfuscation-resilient features from the Abstract Syntax Tree of methods and uses them in combination with invariant attributes from method signatures to derive well-characterizing fingerprints. To identify similar code, we elaborate an effective scoring metric that reliably compares fingerprints at method, class, and package level. We investigate how well our solution tackles obfuscated, shrunken, and optimized code by applying our technique to real-world applications. We thoroughly evaluate our solution and demonstrate its practical ability to fingerprint and recognize code with high precision and recall.

CCS CONCEPTS

• **Security and privacy → Mobile and wireless security**.

KEYWORDS

Android, Abstract Syntax Tree, Fingerprinting, Library Detection, Code Similarity, Code Recognition, Obfuscation

ACM Reference Format:
Johannes Feichtner and Christof Rabensteiner. 2019. Obfuscation-Resilient Code Recognition in Android Apps. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES 2019) (ARES '19), August 26–29, 2019, Canterbury, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3339252.3339260

1 INTRODUCTION

Nowadays, most Android applications are bundled with third-party libraries that potentially include vulnerable or outdated code [13]. The Apache Cordova library, e.g., was affected by a vulnerability that enabled an attacker to interfere with an application's behavior by sending malicious intents[1]. Providing the building blocks for a majority of cross-platform applications, this flaw immediately put the security of all of them at risk. Apart from introducing vulnerabilities, multiple studies [17, 20, 29] have demonstrated that code from external sources can also leak private information, exploit their privileges, or forward sensitive data to unauthorized parties.

Although Android libraries undoubtedly exhibit questionable security practices, insecure code snippets can also be located within app-specific code. If developers copy ready-to-use code snippets, e.g., from programming discussion platforms like Stack Overflow, they unknowingly might also introduce weaknesses. In 2017, a study [14] has revealed that 15.4% of 1.3 million inspected Android apps included security-related code snippets from Stack Overflow, whereas 97.9% of them contained security problems.

Since the use of third-party libraries and the integration of code snippets from external sources have evolved to common practices in app development, it is of utmost importance to find vulnerable code fragments. Despite significant research efforts to dissect apps and uncover such threats, reliable identification of insecure program parts remains challenging. Currently, code recognition in Android applications mostly targets third-party libraries and involves either whitelisting or a similarity-based strategy. In the former case, a precompiled whitelist of directories or package names is used as a reference to individual libraries. However, whitelists are usually gathered manually [4, 9, 22] and have to be maintained to stay up-to-date. Considering the constant intervention and the fact that it is practically infeasible to cover all libraries, this approach does not scale and is only suited for analysis scenarios without obfuscation.

The second approach consists in identifying libraries without prior knowledge [10, 25, 38]. Therefore, apps are decompiled and split into sets of potential library candidates. A similarity metric or hash-based comparison then measures the difference to candidates extracted from other apps. If the score exceeds a predefined threshold, candidates are considered to be the same libraries.

Although research has demonstrated the practical feasibility to identify code, existing work still leaves room for improvement:

(1) Current approaches for code recognition in Android apps focus on detecting individual libraries by name and version. They require large amounts of ground truth for training and do not work effectively if the reference codebase is small or a priori incomplete. Pinpointing specific code snippets instead of full libraries is, thus, infeasible.
(2) State-of-the-art methods strongly depend on Java package names, preserved directory hierarchies, and unaltered method signatures. However, package structures and names can be different in multiple versions of the same library. Also, during compilation, code can mutate as it undergoes automated

[1] https://cordova.apache.org/announcements/2015/05/26/android-402.html

ARES '19, August 26–29, 2019, Canterbury, United Kingdom

ACM ISBN 978-1-4503-7164-3/19/08…$15.00
https://doi.org/10.1145/3339252.3339260

performance-related code optimizations, such as method inlining, duplicate code merging, and removal of unused method parameters. Focusing too much on such auxiliary information can, thus, give a false sense of a good classifier that is only reliable if trained libraries and tested apps exhibit the required attributes and do not apply optimizations.

(3) Android applications commonly apply code transformation techniques, including obfuscation, identifier renaming, and shrinking not only to optimize code but also to harden against various forms of abuse, such as tampering, reverse engineering, and intellectual property theft. While existing classification approaches might yield useful results despite such modifications, the recognition rate with real-world apps could significantly be improved if techniques were resilient to common types of obfuscation and code mangling.

In this paper, we address shortcomings of existing approaches and introduce a solution that is able to recognize arbitrary code fragments in Android apps, even if code transformation techniques, like shrinking or obfuscation, are applied. We overcome the aforementioned limitations by extracting and processing features from the Abstract Syntax Tree (AST) of methods. Our approach does not rely on identifiers of packages, classes, and methods and uses them only as supplementary information. Instead of a hash-based comparison, we measure the similarity of methods using vectorized fingerprints we derive from the AST of methods and transformation-invariant representations of method signatures. To compare code segments, we design a scoring metric that accurately determines inclusion within other code parts and can express the similarity of classes and packages based on an aggregation of fingerprints.

Compared to previous research, our solution excels in reliably recognizing code fragments, even if a very high degree of code obfuscation is applied and if the majority of originally trained code is no longer present, e.g., due to code merging or inlining. Our approach is scalable and succeeds in accurately matching individual small code snippets as well as entire libraries. Aimed at conditions that can be found with real-world apps, our solution is suited for arbitrary tasks that involve code recognition in Android apps.

Contributions. Our key contributions are as follows:

- We present a framework to reliably recognize arbitrary code fragments in Android apps[2]. Our solution can overcome various limitations present in existing research and represents an effective method to identify used libraries, recognize specific code snippets, or find semantically similar candidates.
- We study features in code that are invariant to widely used code transformation techniques and propose a novel feature matching process that is resilient to code mangling, identifier renaming, shrinking, and optimizations, such as inlining, code merging, or removal of unused method parameters.
- We evaluate the quality of our algorithm by testing it with a set of open-source libraries. We compile all libraries multiple times with different forms of code transformations enabled and assess the impact on classification. Moreover, we ensure the soundness of our solution by thoroughly comparing the expressiveness of chosen features and threshold values for matching confidence and package particularity.

Outline. In Section 2, we discuss related work. Section 3 introduces our approach for code recognition and highlights our selection of code features to overcome obfuscation. Subsequently, in Section 4 we present our algorithms for fingerprinting and matching. We evaluate our solution in Section 5 and conclude in Section 6.

2 RELATED WORK

Existing work addresses code recognition in Android apps mostly in the context of third-party library detection. In the following, we point out differences to our solution and also present related research on code clone detection and obfuscation analysis.

Third-Party Library Detection. To investigate the security risks associated with using advertising libraries in mobile apps, early studies rely on a whitelist-based approach [8, 17, 22, 34]. Li et al. [20] extend this concept to cover a general set of libraries. After manually collecting directory and package names of known libraries in a list, it is used to find matches in apps at a large scale. As such approaches fail if obfuscation techniques are used, more elaborate solutions based on machine learning and clustering have been proposed. PEDAL [22] trains a classifier to detect libraries using features extracted from code. AdDetect [26], AnDarwin [12] and WuKong [35] build on the assumption that a library consists of only one packages and segregate package hierarchies into distinct clusters. LibRadar [25] augments the result by assigning each cluster a unique profile representing a library. However, the lack of ground truth and the use of heuristics comes at the cost of precision and does not consider partial library inclusions, e.g., as a consequence of code optimizations. Considering potentially obfuscated package names and deviating package hierarchies, LibD [21] extracts features from code based on method invocations and inheritance relations within classes. Compared to them, our solution also relies on opcode sequences to find and match similar code fragments. However, in contrast to their approach, we can also match partial library occurrences as our fingerprints are composed independently of class inheritance or method overloading.

Tackling the prevalence of code obfuscation, LibScout [4] comes closest to this work. Backes et al. leverage both method signatures and the package hierarchy structure to build profiles per library. An algorithm transforms method signatures into obfuscation-invariant fuzzy descriptors by removing identifiers and class types. These descriptors are then hashed and fed into a Merkle tree, representing the package hierarchy. Although their solution exhibits similar premises and requirements as ours, the differences are in how the overall problem has been approached. Besides obfuscation-invariant method signatures and symbolic package hierarchies, we also add features from the code implementation. While LibScout cannot handle libraries where more than 40% of the original code has been removed, fingerprints built on elements in the Abstract Syntax Tree enable us to recognize not only full libraries but also individual code snippets and library parts. At the same time, our approach helps to improve recognition rates if a *shrinking* code transformation has been applied. Consequently, our solution does not necessarily need large amounts of ground truth for matching and also works if at least a certain amount of code is available. Nonetheless, LibScout is expected to scale better as its approach to compare packages involves a smaller feature set than ours.

[2] The framework is available at: https://github.com/kstudent/astli

Code Clone Detection. Malicious Android applications are often distributed by repackaging legitimate apps [30, 39]. The problem of uncovering small differences between two program versions is commonly referred to as clone or plagiarism detection and conceptually exhibits requirements similar to code recognition. Techniques for clone detection work on semantic and syntactic features of programs and measure the similarity of code based on tokens [11, 33], parsing trees [3, 6], or dependency graphs [9, 23, 24]. Similar to our method, they coalesce code attributes to form a fingerprint that can then be pair-wise tested for equivalence with other candidates.

In a study, Potharaju et al. [28] investigate how attackers can leverage social engineering techniques and app repackaging to distribute malware in the Android market stores. They propose to compute fingerprints based on features extracted from the AST of methods. Therefore, the Android app archive is first transformed into a custom assembly language, followed by pruning the code of each method body, keeping only references to method calls and replacing all variable identifiers with the placeholder `local` for local variables or `param` otherwise. Of all method signatures, only the number of used arguments is preserved. The remaining instructions are then arranged as AST and used to derive a fingerprint vector. The algorithm leans on the hypothesis that two apps are similar if their fingerprints are located within a small neighborhood.

In our work, we adopt the concept as it yields a high detection rate with only 0.5% false positives and solves a problem that is close to ours. One advantage of working with feature vectors instead of full ASTs is the fact that comparing method becomes substantially cheaper than detecting graph isomorphism or computing the tree editing distance between ASTs [27]. As their algorithm creates an app fingerprint as a sum of all method fingerprints, it requires that all code is present during matching. In our case, however, this requirement is not satisfiable as we assume that libraries and code parts may be incomplete or could have been removed during compilation. We, thus, design our own similarity metric that is resilient to common code transformations.

App Code Obfuscation. In a survey from 2018, Wermke et al. [36] analyzed 1.7 million Android apps regarding the use of obfuscation techniques. According to the authors, 24.92% of apps are obfuscated, whereas the most prevalent obfuscation system is ProGuard. While the authors confirm that identifier renaming of classes, methods, and fields is among the most popular features, they make no statements about minified or shrunken apps. Nonetheless, in our solution, we address all variants of code obfuscation and optimizations that ProGuard offers to developers.

Most research of obfuscation in Android apps concentrated on reversing [5, 7] and analyzing applications in spite of obfuscation [16, 32, 37]. More recent studies specifically focus on obfuscated malware, such as a study by Hammad et al. [18], who assessed the impact of obfuscation on Android anti-malware products by inspecting 7 obfuscation strategies and 29 techniques. Also in this context, the work of Garcia et al. [15] inspects obfuscation-resilient properties to uncover malware using machine learning.

3 SYSTEM DESIGN

We design a static analysis framework to recognize code in Android app archives. The primary functionality can be split into two parts: In the **learning phase**, our tool is trained with code fragments or libraries. In the **matching phase**, we automatically analyze a given app and try to recognize code parts using previously learned data. The objectives of our solution can be summarized as follows:

(1) If an app includes a library or code fragment, the tool should identify it both by name and version, if known.
(2) The tool should work equally with obfuscated code.
(3) After analyzing an app, the tool should list packages that resemble previously learned libraries or code fragments with a score indicating how much code has been matched.

3.1 Overcoming Obfuscation

In regular apps, code fragments and libraries can be recognized with reasonable certainty by matching the names of packages, classes, and methods. If code transformation techniques are applied, these identifiers become inconclusive. For a reliable identification nonetheless, we rely on features that *(1)* are suited to identify a code segment and remain the same for semantically similar sections of code, and *(2)* are invariant to common code transformations.

With these two properties in mind, our fingerprinting approach, as detailed in Section 4.1, is based on **AST Vectors** and **Sanitized Signatures**. AST vectors are vectors obtained by extracting structural dependencies of a method's AST. Sanitized signatures result from removing all identifiers from a method signature. These identifiers include the method's name and the class identifiers in all parameter types and the return type. We combine an AST vector and a sanitized signature to a fingerprint. Consequently, a grouped set of fingerprints can represent a package hierarchy. In the following, we explain how we overcome code transformation techniques.

3.1.1 Identifier Renaming. In this transformation, the obfuscator replaces debug symbols with meaningless character sequences. If activated during compilation, package names in the app archive will not disclose hints on included libraries. As our solution does not rely on identifiers at all, it is invariant to identifier renaming.

3.1.2 Shrinking. In this step, an obfuscator removes unused code from an app. In the learning phase, we cannot tell which parts of a library will be removed during app compilation. In preliminary tests, we identified cases where more than 90% of code was pruned. Shrinking does not only decide if an entire package gets in- or excluded; it can also remove unused methods and classes. As all methods in a class and classes in a package can be subject to dead code elimination, we consider this in the fingerprinting process.

3.1.3 Optimizations. Code optimizations involve adding, replacing, rearranging and removing code fragments. Although some of them can affect our features in theory, we can show in our evaluation all of these modifications have a minor impact on detection rates. Basically, a slight change in the AST vector does not necessarily inhibit a correct mapping, since the similarity between AST vectors is based on their distance. However, a sanitized signature that has been altered cannot lead back to the original method, since we check for strict equality when comparing signatures.

The obfuscator ProGuard offers 29 optimizations[3] to developers. Some of them have an impact on our features:

[3] https://www.guardsquare.com/en/products/proguard/manual/usage/optimizations

Inlining Hereby, ProGuard replaces a method invocation with the body of the invoked method. This usually affects short or unique methods but also tail recursive methods could be inlined. Inlining alters the AST vector. However, if only a short method is being inlined, we argue that the alteration is also limited.

Code Merging With this transformation, ProGuard identifies duplicated code fragments and merges them by modifying branch targets. Merging affects the AST vector as it reduces AST nodes.

Method Parameter Removal Hereby, ProGuard identifies unused parameters in methods and removes them from the signature. If applied, the sanitized signature will be altered and we will not be able to match it with the corresponding method.

4 WORKFLOW

The workflow of our approach starts by converting a given code fragment or library into the `.dex` format. This task is delegated to the build tool `dx`, which is provided by the Android SDK. Based on the Dalvik bytecode obtained, for each available method, we extract a fingerprint and arrange all of them within a package hierarchy (see Section 4.1). When **learning** code, we stop after this step and store the results in a database. When **matching** code, we compute a similarity score between 0 and 1, indicating whether given code fragments can be recognized fully, partially, or not at all by comparing with learned fingerprints and package hierarchies.

4.1 Fingerprinting Code

To derive fingerprints, we explain AST vectors in Section 4.1.1 and sanitized signatures in Section 4.1.2. The aggregation of fingerprints in package hierarchies is elaborated in Section 4.1.3. Finally, in Section 4.1.4, we give a practical example of a fingerprint derivation.

Algorithm 1: Building a minimal AST from a method body

```
Input  : Method Body
Output : Abstract Syntax Tree
1  AST ← createRootNode();
2  foreach instruction ∈ method body do
3      keep instructions with opcode in {INVOKE_DIRECT,
         INVOKE_VIRTUAL};
4      instructionNode ← createNode(instruction.opcode);
5      foreach parameter ∈ instruction do
6          parameterNode ← createNode(parameter.type);
7          instructionNode.addChild(parameterNode);
8      end
9      AST.addChild(instructionNode);
10 end
11 return AST
```

4.1.1 *AST Vectors.*

We generate an AST vector by building an AST and conveying this tree to a vector. To compare methods, AST vectors are preferable over regular ASTs, as they allow to express the similarity of two methods by computing their distance.

As depicted in Listing 1, we first build a minimal AST over a method body. Starting at the root node of a tree (line 1), we iterate over all program statements contained in the method (line 2) and filter instructions of the type `INVOKE_DIRECT` and `INVOKE_VIRTUAL`[4] (line 3). We focus on these two as they are the most common method invocation calls according to Potharaju et. al. [28]. However, for a more detailed vector, we could also keep track of further suffixes, i.e., `_SUPER`, `_STATIC` and `_INTERFACE`. Next, we create an AST node for the current invocation call (line 4) and attach a child node for each parameter of the method (lines 5-8). Finally, we add the instruction node as a child to the tree root (line 9).

Algorithm 2: Conversion of an AST to an AST vector

```
Input  : Abstract Syntax Tree
Output : Abstract Syntax Tree Vector
1  vector = createVector();
2  //count horizontal features;
3  foreach invokeActionNode ∈ AST.getChildren() do
4      #locals ← |{c ∈ invokeActionNode.getChildren() |
         c.type = local}|;
5      #params ← |{c ∈ invokeActionNode.getChildren() |
         c.type = param}|;
6      vector[local_local] ← $\binom{\#locals}{2}$;
7      vector[param_param] ← $\binom{\#params}{2}$;
8  end
9  //count vertical features;
10 foreach lvl1Node ∈ AST.getChildren() do
11     increment(vector[lvl1Node]);
12     foreach lvl2Node ∈ lvl1Node.getChildren() do
13         increment(vector[lvl2Node]);
14         increment(vector[lvl1Node, lvl2Node]);
15     end
16 end
17 return vector
```

We convert the AST into a vector by counting *horizontal* and *vertical features*, as defined by Potharaju et al. [28]: A *horizontal feature* is a pair of leaf nodes with the same parent node, whereas a *vertical feature* is a directed path of arbitrary length, starting at the root node. Each dimension in our AST vector resembles the number of occurrences of a particular horizontal or vertical feature.

As shown in Listing 2, starting with an empty vector (line 1), we count horizontal features by going through all first level nodes of the AST, determining the number of leaf pairs for each node (lines 3-8). For each invocation call, we count the number of *local variables* and *parameters* (lines 4-5), and compute the amount of pairs of type `local-local` and `param-param` (lines 6-7). Determining the number of pairs is equivalent to the *handshake problem*[5], so we can compute it using the binomial coefficient over 2. Next, we count the vertical features by iterating over first level nodes of the AST again (lines 10-16) and increment the occurrence count of the current node by 1 (line 11). Finally, we iterate over all of its children and increment occurrences of both paths, be it either 2nd-level relation only or a conjunction of first and second-level nodes. (line 12).

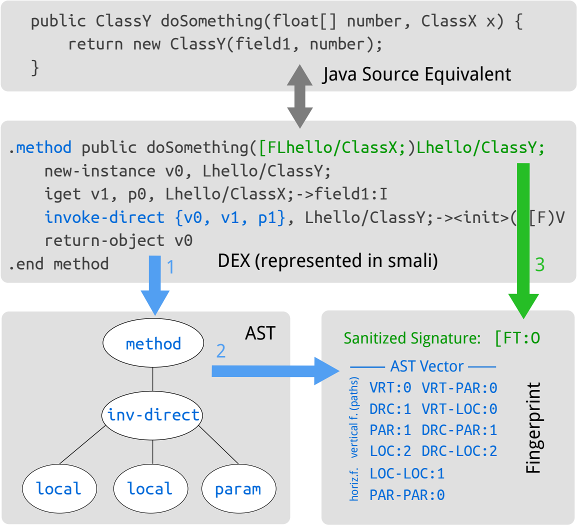
4.1.2 *Sanitized Signature.*

Sanitized signatures contain data from a method signature that is invariant to obfuscation. To sanitize the signature from features affected by obfuscation, we remove method identifiers, parameter names, and modifiers from the original signature. Further, we replace parameter types and the return type with a single letter code. For primitive types, we adopt the mapping from `smali`[6]. Since object types can be subject to identifier renaming, they are mapped to an obfuscation-invariant token depending on its type. If it equals the class we are currently processing, we assign

[4] https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html

[5] http://mathworld.wolfram.com/HandshakeProblem.html

[6] https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields



```
public ClassY doSomething(float[] number, ClassX x) {
    return new ClassY(field1, number);
}
```

```
.method public doSomething([FLhello/ClassX;)Lhello/ClassY;
   new-instance v0, Lhello/ClassY;
   iget v1, p0, Lhello/ClassX;->field1:I
   invoke-direct {v0, v1, p1}, Lhello/ClassY;-><init>(I[F)V
   return-object v0
.end method
```

Figure 1: Fingerprint extraction example.

the letter T, if belonging to the same package as the current class, we assign 0 and, otherwise, we set the letter E for external origin.

4.1.3 Fingerprint and Package Hierarchy. Having extracted an AST vector and a sanitized signature, we combine both features to a fingerprint. Although it could already be used for matching, just by itself the fingerprint does not provide information to map packages unambiguously. An example of ambiguous methods are *getters* and *setters*: They have a similar structure and, thus, a similar fingerprint but do not necessarily belong to the same class.

We overcome ambiguity by capturing the entire structure of a package. Two unrelated methods might share the same fingerprint, but two unrelated classes are unlikely to share all method fingerprints. The same logic holds for packages: We conclude that two packages are related if their classes share the same fingerprints. Consequently, we require a data structure to group fingerprints in classes and classes in packages. We refer to it as *package hierarchy*.

4.1.4 Example. Figure 1 shows a complete example of a fingerprint extraction. We convert the method doSomething from the class ClassX located in the package hello. Note that we use smali to represent the method and its implementation. The original Java source code equivalent should help to gain a better understanding of the .dex format but is not involved in the extraction process.

1. **Convert method body to minimal AST:** We add the top node method and create a node inv-direct that represents the instruction invoke-direct {v0,v1,p1}. The parameters v0,v1 of the invoke-direct statement are local parameters. v0 is an instance of ClassY and v1 an Integer that contains the value of ClassX.field1. For v0,v1 and the parameter number in p1, we generate two local nodes and one param node and add them to the inv-direct node as children.
2. **Convert AST to AST vector:** Vertical features are paths consisting of 1 or 2 nodes. We count the following paths of length 1: DRC:1, because of the inv-direct node; LOC:2, PAR:1 because of the respective child nodes; VRT:0 as there is no invoke-virtual node in the tree. Paths of length 2 are DRC-LOC:2 and DRC-PAR:1. Both INV-LOC and INV-PAR remain 0. Finally, we generate horizontal features by counting the pairs of local and param nodes. There is one local pair, thus, LOC-LOC:1 and no param pair.
3. **Generate sanitized signature:** We are interested in the parameters [F, representing float[] number, and hello/ClassX, followed by the return type hello/ClassY. Parameters and return type are colored green in the .dex-box of the figure. We leave [F as is and substitute hello/ClassX with letter T, as the type matches the currently processed class. We add a colon : to divide parameter types from return type and replace the return type hello/ClassY with the character 0, as the type is located in the hello package. This results in the sanitized signature [FT:0.
4. **Form fingerprint from AST vector and sanitized signature**

4.2 Recognizing Code

In the matching process, we are given a set of package hierarchies P_a, which we extracted from an Android app archive. For each package hierarchy $p_a \in P_a$ we pursue the following steps:

(1) We sort all fingerprints in p_a by *particularity* in descending order, such that we can choose a set of particular fingerprints. An explanation of particularity is given in Section 4.2.1.
(2) For each fingerprint, we query the database for previously learned fingerprints with the same AST vector and sanitized signature. We collect the package hierarchies of similar fingerprints and store them in the candidate set P_l.
(3) For each candidate $p_l \in P_l$, we check if $p_a \subseteq p_l$, which means that the app package is **included** in the known package. Section 4.2.2 defines this relation and steps to compute it.
(4) If $p_a \subseteq p_l$, we compute the **similarity** $s(p_a, p_l)$, which depends on the AST vectors in p_a and p_l. Otherwise, we set $s(p_a, p_l)$ to 0. Section 4.3 elaborates on the definition and computation of the similarity score.
(5) We sort package candidates by similarity score in descending order and return the package with the highest score that meets a minimum threshold as a match.

4.2.1 Fingerprint Particularity. Some fingerprints are more likely to match with unrelated fingerprints than others. When populating a set of candidates, rare fingerprints are preferable over frequent ones as they will less likely yield *false positive* candidates. In a test with 120,000 fingerprints, we observed that fingerprints with long AST vectors are more particular as they occur less frequently.

We approximate the particularity of a fingerprint with a score. Let $m = (s, v)$ be a method fingerprint with a sanitized signature s and an AST vector v. The *particularity score* of m is defined as:

$$\text{score}(m) := w_s \cdot \text{length}(s) + w_v \cdot \|v\|_1, \tag{1}$$

whereas length(s) returns the amount of character of s and $\|v\|_1$ denotes the Manhattan distance of v. We weigh both dimensions with w_s and w_v in order to rectify the distributions.

4.2.2 Inclusion. The inclusion relation $\subseteq$ expresses if a package hierarchy p is included in a package hierarchy p'. Inclusion depends on the sanitized signatures in both package hierarchies. We use inclusion instead of equivalence in order to handle the loss of code when an obfuscator removes dead code from an app. Therefore, inclusion is reflexive and transitive, but not symmetric:

$$p \subseteq p' \Leftrightarrow \exists f_c : p \mapsto p',\ f_c \dots \text{ injective.} \tag{2}$$

In other words: Package p is included in package p' if and only if there exists an injective mapping f_c for all classes in p to the classes in p'. We require injectivity because we expect each code class to end up at most as one class in the app.

We add further requirements for our class mapping f_c. Let $c \in p$ be a class in the package p and $c' \in p'$. Then we have:

$$f_c(c) = c' \Rightarrow c \subseteq c'. \tag{3}$$

If we map an ap class c to a library class c', then the app class is included in the library class. The inclusion relation between classes is defined analogously to the inclusion relation between packages:

$$c \subseteq c' \Leftrightarrow \exists f_m : c \mapsto c',\ f_m \dots \text{ injective.} \tag{4}$$

However, we can only map a fingerprint $m \in c$ to a fingerprint $m' \in c'$ if their sanitized signatures are equal, or:

$$f_m(m) = m' \Rightarrow \text{Signature of } m \text{ and } m' \text{ are equal.} \tag{5}$$

Having defined inclusion for both packages and classes, we can now describe how our solution practically determines inclusion.

Determining Class Inclusion. Let $c = \{s_1, ..., s_n\}$ be a class consisting of sanitized signatures s_i (for the sake of simplicity, we ignore AST vectors and fingerprints for now). Then we can determine if $c \subseteq c'$ in a *greedy* manner as described in Listing 3. The idea behind this approach is to find all signatures from c in c'. If we find one, we delete it from the c' set such that we do not pick the given signature in c' twice. If we found all signatures from c in c', we know that there is an injective mapping f_m and, thus, $c \subseteq c'$.

Determining Package Inclusion. In order to determine if package p is included in package p', we need to find an injective mapping f_c for all classes in p to classes in p'. This task is more challenging than the mapping f_m for methods because of the lacking symmetry in the class inclusion relation. Figure 2 illustrates an example, where the greedy approach from Listing 3 fails.

We are given the packages p and p', and we can tell that $p \subseteq p'$ because there exists an injective mapping f_c with $f_c(c_1) = c'_1$ and $f_c(c_2) = c'_2$. However, the greedy approach fails because class c_1 can also be assigned to c'_2, since $c_1 \subseteq c'_2$ holds. If we assign c_1 to c'_2, we end up with c_2 being unassigned, because of $c_2 \not\subseteq c'_1$.

In the case where a valid assignment is not possible, we could backtrack by amending some assignments until we explore all possibilities. However, we opted to reduce the problem such that we can solve it with the **Hungarian Algorithm** [19]. Given a set of workers, a set of tasks and a cost matrix, this algorithm assigns workers to tasks such that the overall costs are minimized. Instead of workers and tasks, we use classes of p and p'. We construct our cost matrix M_s as follows:

$$M_s \in \{0,1\}^{|p| \times |p'|},\ M_s[i,j] = \begin{cases} 0 & \text{if } c_i \subseteq c'_j \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

If we apply the Hungarian Algorithm on M_s, we end up with an assignment f_c. Since the algorithm minimizes the cost of f_c, it prefers assignments that cost 0 over the ones that cost 1. Eventually, we compute the overall cost of f_c with

$$\text{cost}(M_s, f_c) := \sum_{i=1}^{|p|} M_s[i, f_c(i)] \tag{7}$$

and can argue that if

$$\text{cost}(M_s, f_c) = 0 \Leftrightarrow \exists f_c : p \mapsto p',\ f_c \dots \text{ injective.} \Leftrightarrow p \subseteq p' \tag{8}$$

Algorithm 3: *Greedy* Class Inclusion Check

```
Input  : Class $c$, Class $c'$
Output : True if $c \subseteq c'$
foreach $s_i \in c$ do
    if $s_i \in c'$ then
        remove $s_i$ from $c'$
    else
        return False;
    end
end
return True;
```

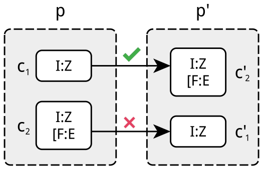


Figure 2: Example with failing Greedy Package Inclusion.

4.3 Similarity Score

The similarity score helps us to determine how similar two packages hierarchies p and p' are. The score depends on the similarity of AST vectors in the respective packages. Packages that are similar yield a higher score than packages that are not. Next, we explain how we measure the similarity between packages, classes, and AST vectors.

4.3.1 Package Similarity. We compute the similarity score $s(p, p')$ by leveraging the Hungarian Algorithm, as it helps us to find the mapping between classes with maximum similarity. We fill the cost matrix S with the similarity score of the respective classes $s(c, c')$:

$$S \in \mathbb{R}^{|p| \times |p'|},\ S[i,j] = \begin{cases} s(c, c') & \text{if } c_i \subseteq c'_j \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Since the Hungarian Algorithm minimizes the costs in the cost matrix, we apply it on an inverted matrix S_{inverted}, where we negate each entry and shift it by the maximum:

$$S_{\text{inverted}} = (\max(S) - S[i,j])_{ij} \tag{10}$$

After the Hungarian Algorithm generated a mapping f_c, we can compute the similarity with $\text{cost}(S, f_c)$, as defined in Definition 7.

4.3.2 Class Similarity. Let $c = \{m_1, ..., m_n\}$ be a class consisting of a list of fingerprints m_i where each fingerprint consists of a sanitized signature s_i and an AST vector v_i. Then the class similarity $s(c, c')$ can be determined with the Hungarian Algorithm once again. First, we generate the cost matrix T:

$$T \in \mathbb{R}^{|c| \times |c'|},\ T[i,j] = \begin{cases} s(v_i, v'_j) & \text{if } s_i = s'_j \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

We invert T as in Equation 10 and let the Hungarian Algorithm find the best assignment. We use the cost function $\text{cost}(T, f_c)$ in Definition 7 to determine the similarity score $s(c, c')$.

4.3.3 AST Vector Similarity. We express the similarity between two AST vectors v and v' with the following formula:

$$s(v, v') = \max(0, \|v\|_1 - \|v - v'\|_1) \tag{12}$$

We use the Manhattan distance to determine distance and length of vectors, as proposed in [27]. Our formula fulfills these requirements:

- We want the similarity to be 0 if the vectors are too far apart. The threshold where similarity becomes 0 is reached if the difference between to vectors is greater than the vector itself.
- We do not accept negative values for similarity, as we want to avoid the situation where a mismatch of vectors worsens the overall score of an assignment. We ensure this requirement by taking the maximum between the difference and 0.
- We require maximum similarity when both vectors are equal. In that case $\|v - v'\|_1$ becomes 0, such that $s(v, v') = \|v\|_1$.
- If $\|v_1\|_1 > \|v_2\|_1$, we require $s(v_1, v_1) > s(v_2, v_2)$ because we want larger and therefor more particular vectors to have more influence on the assignment cost.

5 EVALUATION

The goal of this evaluation is twofold. First, we investigate how well AST vectors and sanitized signatures fingerprint code and tackle obfuscation (see Section 5.2). Second, applying our solution on a set of open-source Android apps, we assess *(1)* how much code of a library is needed to accurately recognize it (see Section 5.3), *(2)* how much confidence a match should have to be significant (see Section 5.4), and *(3)* how well we can recognize individual libraries when different obfuscation techniques are applied (see Section 5.5).

5.1 Method and Dataset

In the **learning phase**, we seek to assign similar fingerprints to semantically similar code fragments. Distinctive features should characterize unrelated code. To better understand how well AST vectors and sanitized signatures identify code, as a first step, we test our features using a home-made app that is obfuscated, includes two libraries, and 150 packages. The results give an intuition on how changes in the codebase or parameters affect the tool's accuracy.

To evaluate how our solution can **recognize code** in real-world apps, we chose to crawl the *F-Droid* Repository[7]. Since this app repository offers only *Free and Open Source Software (FOSS)*, we can download the source codes of apps including configuration files for the Gradle build system. From these files, which are needed for compilation only but are not included in final Android app archives, we can extract a list of used library packages. On their basis, we split the code into different parts to derive fingerprints for each of them. Moreover, we adapted the build files of all downloaded FOSS apps to compile multiple versions with different code transformation techniques enabled: *shrunken, obfuscated, shrunken + obfuscated, shrunken + obfuscated + optimized.* For this evaluation, we crawled source codes of 800 FOSS apps and compiled one regular and four transformed versions, resulting in 4,000 app samples overall.

The collected dataset enables us to unambiguously verify how well learned code can be recognized. Based on the assumption that FOSS apps exhibit the same library inclusion as other apps, our results should also hold for arbitrary Android applications.

5.2 Fingerprint Quality

Our algorithm derives a representation of code using *AST vectors* and *sanitized signatures.* We designed these techniques to be invariant to commonly used code transformations while still being able to characterize individual code parts precisely.

Question: *How well can our features describe code fragments?*

To answer this question, we deploy a confusion matrix $M = (m_{ij})$. It depicts how well we assign labels to Android app packages and visualizes incorrectly recognized packages. Each row resembles a label, each column an actual package. The color of a cell m_{ij} indicates the confidence that package i of an arbitrary app matches package j of a library. We can draw conclusions on the matching quality from the structure of M: If its main diagonal is confident and the rest is not, we identify code segments without confusion.

5.2.1 Setup. To build a confusion matrix, we use the sample set of 150 packages included in our obfuscated test app. The set is large enough to feature a variety of different packages and sufficient to visualize confusion. We compute the similarity score between each app package with each library package we learned before.

5.2.2 Results. As depicted in Figure 3, we derived three matrices that show confusion when only AST vectors or sanitized signatures are used for code identification, and with both features combined. The x and y-axes are sorted by package particularity (see Section 4.2.1). Packages with small particularity are located on the top/left, whereas packages on bottom/right have high particularity.

With AST vectors, code similarity is confident on the main diagonal but overall prone to confusion. The upper right part of the matrix shows many packages that have been mislabeled with high confidence. We can explain this observation by the fact that a small app package can be easily mapped to a large library package. The other way around does not hold: large app packages are not confused with small library packages.

The confusion matrix for sanitized signatures shows that the similarity measure is either absolutely confident that a package is *included* (see Section 4.2.2) in another one, or otherwise not confident at all. Compared to AST vector similarity, we observe less confusion in the upper right part of the matrix, where less particular app packages are compared to more particular library packages. Confusion, however, occurs with small app packages as large packages are likely to contain all signatures of small packages.

With both features combined, confusion is mostly eliminated aside from low particularity packages in the top rows of the matrix. The small clusters around the main diagonal in the top left area result from packages that implement the same interfaces and are, thus, semantically related. Overall, we see that the combination of AST vectors and sanitized signatures can identify code almost without confusion and, thereby, paves the way for accurate recognition.

5.3 Threshold for Package Particularity

The confusion matrix for AST vectors and sanitized signatures combined showed that the remaining confusion was caused by packages with low particularity. As a remedy for confusion and to improve accuracy, we introduce the threshold for minimum package particularity (t_{pp}). It decides whether a particular package is sufficiently expressive to be used for learning and matching.

[7] https://f-droid.org

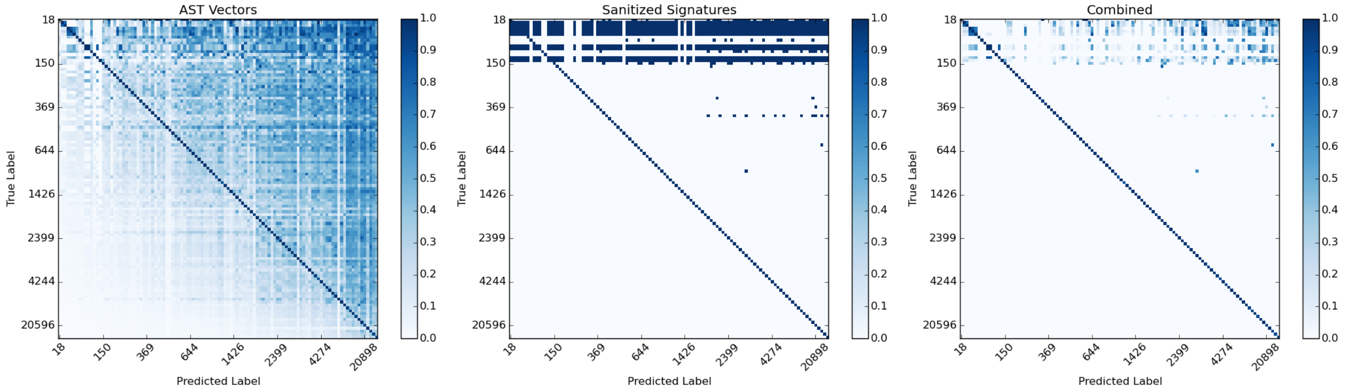


Figure 3: Confusion matrices based on AST vectors (left), sanitized signatures (middle), and both features combined (right).

Before processing an app package, we check if the package is particular enough. If not, we simply ignore it as we cannot rely on its matches. The higher we choose t_{pp}, the more accurate recognition becomes. However, with a high t_{pp} we ignore more packages and, thus, identify fewer code fragments overall. In the following, we measure this influence with the *keep ratio*:

$$\text{keep ratio} = \frac{|\text{Analyzed Packages of App}|}{|\text{Packages of App}|} \tag{13}$$

Our goal is to find a reasonable value for t_{pp} that represents a compromise between recognition accuracy and keep ratio.

Question: *How much particularity is needed for precise recognition?*

5.3.1 Setup. We use our set of real-world apps and iteratively try recognition with values between 0 and 200 for t_{pp}. After each round, we build a confusion matrix and compute accuracy and keep ratio.

5.3.2 Results. Figure 4 shows how t_{pp} influences accuracy and keep ratio. For small values of t_{pp}, the keep ratio stays near 1, which means that all packages are used for analysis. The accuracy in this area is at 0.7, indicating 30% incorrect matches. The higher t_{pp} becomes, the more packages we drop and the more accurate our results become. At $t_{\text{pp}} = 75$, accuracy reaches 0.9 and stagnates from there on, whereas the keep ratio keeps declining.

As also observed in Figure 3, confusion arises below a certain package particularity but decreases above a specific particularity. Hence, we see that a t_{pp} value of 80 delivers the highest accuracy without dropping too many packages. If a high keep ratio is targeted instead of accuracy, any value for t_{pp} below 80 would be reasonable.

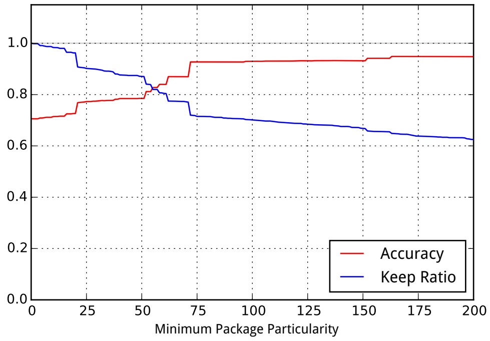


Figure 4: Influence of t_{pp} on accuracy and keep ratio.

5.4 Threshold for Matching Confidence

As explained in Section 4.3, we express the similarity between an app package p_a and a library package p_l with the similarity score $s(p_a, p_l)$. This score depends both on how similar and on how particular packages are: More particular packages result in a higher score, whereas less particular packages are scored lower. This imbalance is problematic when deciding on whether a recognition result is significant. A constant threshold for all packages favors more particular packages over less particular ones with no regard to the actual similarity. To counteract, we derive the *confidence* we put into a match from the package similarity as follows:

$$\text{confidence}(p_a, p_l) = \frac{s(p_a, p_l)}{s(p_a, p_a)} \tag{14}$$

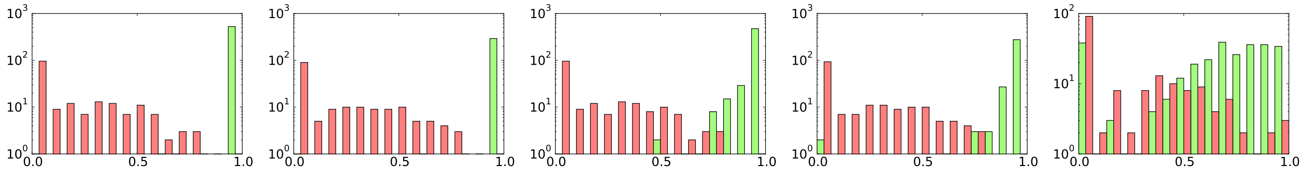
The resulting value is $\in [0, 1]$ because $0 \leq s(p_a, p_l) < s(p_a, p_a)$. Our goal is to find a reasonable threshold (t_{mc}) that indicates if a recognition result is significant enough to be accepted.

Question: *How much confidence makes a recognition result reliable?*

5.4.1 Setup. To find the best value for t_{mc}, we use our sample set of FOSS apps and remodel the multi-class problem into a binary classification problem with the One-Vs-All approach. The binary classifier tells whether a package is known (*positive*, +) or unknown to the system (*negative*, -). We transform each match into the new problem domain by expressing learned library packages as *positive*, and all others, e.g., unlearned packages or app packages, as *negative*.

With our recognition results transformed into binary classifications, we build *Receiver Operational characteristics* (ROC) curves that can illustrate the performance of a binary classifier and reveal how the accept threshold for confidence influences both true and false positive rate. The ROC curves shed light on the separability of known and unknown packages and help in finding a reasonable threshold value for matching confidence t_{mc}. We repeat this process for all app samples to determine how well we can distinguish known from unknown packages if code transformations are used.

5.4.2 Results. Figure 6 compares different ROC curves of the binary classifier. The classifier separates known from unknown packages with high accuracy in almost all build types. The *Area Under the ROC Curve* (AUC) for these build types is above 99.5%. The only build type where our classifier performs suboptimally is the one



(a) regular (b) shrunken (shr.) (c) obfuscated (obf.) (d) shr. + obf. (e) shr. + obf. + optimized

Figure 5: Confidence histograms for known (green) and unknown (red) packages.

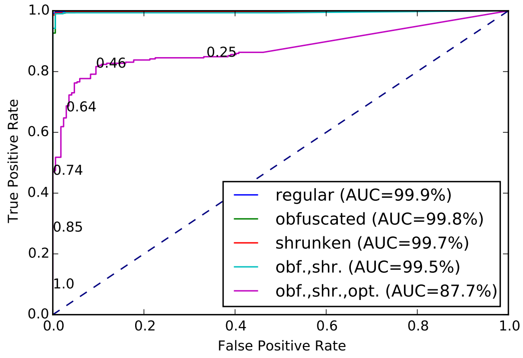


Figure 6: Comparing ROC curves of different build types.

with obfuscation, shrinking and optimizations activated. In this build type the AUC is 87.7%, which is still acceptable.

Figure 5 shows how known and unknown package matches are distributed over their confidence. The red bar indicates the occurrence of unknown packages, the green bar the occurrence of known packages. Note that the y-axis is scaled logarithmically since matches with $confidence = 1$ and $confidence = 0$ tend to dominate the histogram. We notice that *regular* and *shrunken* app packages can be separated perfectly at $t_{\text{mc}} = 0.8$. Separability in the confidence histogram of *obfuscated* app packages (Figure 5c) is still good since the distributions barely overlap. The same holds for *obfuscated and shrunken* app packages in Figure 5d. As seen in Figure 6, app packages with all possible transformations applied are the hardest to separate. Judging from the corresponding ROC curve, the best value for t_{mc} to optimize accuracy is around 0.5.

5.5 Code Recognition

In Section 4.2, we elaborated a workflow to recognize learned code by computing fingerprints of methods, aggregating them in package hierarchies and testing for similarity with known packages.

Question: *How well does our approach recognize app packages?*

5.5.1 Setup. For this scenario, use our FOSS sample set and analyze all apps in all build types. Aimed at highest recognition accuracy, we set the thresholds for matching confidence t_{mc} to 0.5 (see Section 5.4) and for package particularity t_{pp} to 80 (see Section 5.3). To all obtained results, we apply the following multiclass performance metrics [31] by using the formulas shown in Figure 7:

Accuracy: Determines how many app packages have been labeled correctly in relation to all recognition matches.
Precision: Ability of our solution to not mislabel packages.
Recall: Ability to find all instances of a package.
F1 Score: Harmonic mean between Precision and Recall.

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{l} tp_i \qquad \text{Precision}_\text{M} = \frac{1}{l}\sum_{i=1}^{l}\frac{tp_i}{tp_i + fp_i}$$

$$\text{Recall}_\text{M} = \frac{1}{l}\sum_{i=1}^{l}\frac{tp_i}{tp_i + fn_i} \qquad \text{F1Score}_\text{M} = \frac{2\,\text{precision}_\text{M}\text{recall}_\text{M}}{\text{precision}_\text{M} + \text{recall}_\text{M}}$$

Figure 7: Multiclass metrics [31]. $n\ldots$ amount of matches, $l\ldots$ amount of known packages, $p_i\ldots$ app package.

5.5.2 Results. The recognition results are summarized in Table 1. As shown, all metrics perform well in all build types except for the set of *obfuscated, shrunken, and optimized* apps. By manually investigating classifications, we noticed that the optimization technique *Method Parameter Removal* causes the weaker performance with this build type (see Section 3.1.3). With this code transformation enabled, ProGuard prunes method signatures from unused parameters, leading to different sanitized signatures. Nonetheless, the use of AST vectors ensures that recognition is still feasible.

Table 1: Code recognition performance on real-world apps.

	regular	obfuscated	shrunken	obf.,shr.	obf.,shr.,opt.
Accuracy	96.76%	96.61%	93.30%	93.40%	78.83%
Precision	98.03%	97.80%	94.81%	94.69%	70.64%
Recall	99.15%	98.92%	97.05%	96.80%	71.95%
F1 Score	98.17%	97.94%	94.94%	94.80%	70.32%

5.6 Summary

We examined how AST vectors and sanitized signatures align with real-world apps that use code transformations. First, we assessed how well our techniques describe obfuscated code fragments. Three confusion matrices revealed that although each technique is capable of identifying obfuscated code on its own, results are significantly improved when both features are combined. We also noticed that most confusion arises in packages with low particularity. Therefore, we introduced a threshold t_{pp} value that indicated how much code was relevant to keep high accuracy high while not dropping too much packages. We also introduced a match confidence threshold t_{mc} to decide on whether to accept or to reject a recognition result. To find a reasonable value for t_{mc} we remodeled our problem into binary classification. ROC curves underlined how well we can distinguish known from unknown packages despite code transformation. In our final study, we tested code recognition with a set of Android app samples and found that our solution delivers high values for accuracy, precision, recall, and F-Score in all scenarios.

6 CONCLUSION

The use of third-party libraries and the integration of code snippets from public sources have become common practices in Android application development. Security issues and vulnerabilities in such components reach a high number of end-users and put sensitive data at risk. However, a precise recognition of such program parts is challenging if code transformation techniques were applied.

In this work, we presented a solution that can reliably recognize code snippets or libraries even if obfuscation, shrinking, or similar techniques are used. By extracting fingerprints from the Abstract Syntax Tree of methods and combining them with obfuscation-resilient features of method signatures, we succeed in accurately characterizing code. We thoroughly evaluated the applicability of our technique and demonstrated that can we describe and recognize code fragments with high precision. Our solution contributes to an effective identification of problematic code in Android applications.

REFERENCES

[1] [n. d.]. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM.

[2] [n. d.]. *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM.

[3] Marat Kh. Akhin and Vladimir M. Itsykson. 2013. Tree slicing: Finding intertwined and gapped clones in one simple step. *Automatic Control and Computer Sciences* 47 (2013), 427–432.

[4] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications, See [1], 356–367.

[5] Richard Baumann, Mykolai Protsenko, and Tilo Müller. 2017. Anti-ProGuard: Towards Automated Deobfuscation of Android Apps. In *Workshop on Security in Highly Connected IT Systems – SHIS*. ACM, 7–12.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *International Conference on Software Maintenance – ICSM 1998*. IEEE Computer Society, 368–377.

[7] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. 2016. Statistical Deobfuscation of Android Applications, See [1], 343–355.

[8] Theodore Book, Adam Pridgen, and Dan S. Wallach. 2013. Longitudinal Analysis of Android Ad Library Permissions. *CoRR* abs/1303.0857 (2013).

[9] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *International Conference on Software Engineering – ICSE 2014*. ACM, 175–186.

[10] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *IEEE Symposium on Security and Privacy – S&P 2016*. IEEE Computer Society, 357–376.

[11] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *European Symposium on Research in Computer Security – ESORICS 2012 (LNCS)*, Vol. 7459. Springer, 37–54.

[12] Jonathan Crussell, Clint Gibler, and Hao Chen. 2013. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In *European Symposium on Research in Computer Security – ESORICS 2013 (LNCS)*, Vol. 8134. Springer, 182–199.

[13] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Conference on Computer and Communications Security – CCS 2017*. ACM, 2187–2200.

[14] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *IEEE Symposium on Security and Privacy – S&P 2017*. IEEE Computer Society, 121–136.

[15] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, obfuscation-resilient detection and family identification of Android malware, See [2], 497.

[16] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. CodeMatch: obfuscation won't conceal your repackaged app. In *Foundations of Software Engineering – FSE 2017*. ACM, 638–648.

[17] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2012*. ACM, 101–112.

[18] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products, See [2], 421–431.

[19] Harold W. Kuhn. 2010. The Hungarian Method for the Assignment Problem. In *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*. Springer, 29–47.

[20] Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An Investigation into the Use of Common Libraries in Android Apps. In *Software Analysis, Evolution, and Reengineering – SANER 2016*. IEEE Computer Society, 403–414.

[21] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: scalable and precise third-party library detection in android markets. In *International Conference on Software Engineering – ICSE 2017*. IEEE / ACM, 335–346.

[22] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *Mobile Systems – MobiSys 2015*. ACM, 89–103.

[23] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Conference on Knowledge Discovery and Data Mining – SIGKDD 2006*. ACM, 872–881.

[24] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Foundations of Software Engineering – FSE 2014*. ACM, 389–400.

[25] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *International Conference on Software Engineering – ICSE 2016*. ACM, 653–656.

[26] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. 2014. AdDetect: Automated detection of Android ad libraries using semantic analysis. In *International Conference on Intelligent Sensors, Sensor Networks and Information Processing – ISSNIP 2014*. IEEE, 1–6.

[27] Hoan Anh Nguyen, Tung Thanh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection. In *Fundamental Approaches to Software Engineering – FASE 2009 (LNCS)*, Vol. 5503. Springer, 440–455.

[28] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. 2012. Plagiarizing Smartphone Applications: Attack Strategies and Defense Techniques. In *Engineering Secure Software and Systems – ESSoS 2012 (LNCS)*, Vol. 7159. Springer, 106–120.

[29] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. 2016. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *Network and Distributed System Security Symposium – NDSS 2016*. The Internet Society.

[30] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. 2014. Towards a scalable resource-driven approach for detecting repackaged Android applications. In *Annual Computer Security Applications Conference – ACSAC 2014*. ACM, 56–65.

[31] Marina Sokolova and Guy Lapalme. 2009. A systematic analysis of performance measures for classification tasks. *Inf. Process. Manage.* 45 (2009), 427–437.

[32] Mario Linares Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2014. Revisiting Android reuse studies in the context of code obfuscation and library usages. In *Mining Software Repositories – MSR 2014*. ACM, 242–251.

[33] Mario Linares Vásquez, Andrew Holtzhauer, and Denys Poshyvanyk. 2016. On automatically detecting similar Android apps. In *International Conference on Program Comprehension – ICPC 2016*. IEEE Computer Society, 1–10.

[34] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of google play. In *Measurement and Modeling of Computer Systems – SIGMETRICS 2014*. ACM, 221–233.

[35] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: a scalable and accurate two-phase approach to Android app clone detection. In *Symposium on Software Testing and Analysis – ISSTA 2015*. ACM, 71–82.

[36] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In *Annual Computer Security Applications Conference – ACSAC 2018*. ACM, 222–235.

[37] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: towards obfuscation-resilient mobile application repackaging detection. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2014*. ACM, 25–36.

[38] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *Software Analysis, Evolution, and Reengineering – SANER 2018*. IEEE Computer Society, 141–152.

[39] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy – S&P 2012*. IEEE Computer Society, 95–109.