# Statistical Deobfuscation of Android Applications

Benjamin Bichsel Veselin Raychev Petar Tsankov Martin Vechev
Department of Computer Science
ETH Zurich
bichselb@student.ethz.ch {veselin.raychev, ptsankov, martin.vechev}@inf.ethz.ch

## ABSTRACT

This work presents a new approach for deobfuscating Android APKs based on probabilistic learning of large code bases (termed "Big Code"). The key idea is to learn a probabilistic model over thousands of non-obfuscated Android applications and to use this probabilistic model to deobfuscate new, unseen Android APKs. The concrete focus of the paper is on reversing layout obfuscation, a popular transformation which renames key program elements such as classes, packages and methods, thus making it difficult to understand what the program does.

Concretely, the paper: (i) phrases the layout deobfuscation problem of Android APKs as structured prediction in a probabilistic graphical model, (ii) instantiates this model with a rich set of features and constraints that capture the Android setting, ensuring both semantic equivalence and high prediction accuracy, and (iii) shows how to leverage powerful inference and learning algorithms to achieve overall precision and scalability of the probabilistic predictions.

We implemented our approach in a tool called DeGuard and used it to: (i) reverse the layout obfuscation performed by the popular ProGuard system on benign, open-source applications, (ii) predict third-party libraries imported by benign APKs (also obfuscated by ProGuard), and (iii) rename obfuscated program elements of Android malware. The experimental results indicate that DeGuard is practically effective: it recovers 79.1% of the program element names obfuscated with ProGuard, it predicts third-party libraries with accuracy of 91.3%, and it reveals string decoders and classes that handle sensitive data in Android malware.

## 1. INTRODUCTION

This paper presents a new approach for deobfuscating Android applications based on probabilistic models. Our approach uses large amounts of existing Android programs available in public repositories (referred to as "Big Code") to learn a powerful probabilistic model which captures key features of non-obfuscated Android programs. It then uses this probabilistic model to suggest a (statistically likely) deobfuscation of new, obfuscated Android applications. Our approach enables a variety of security applications. For instance, our system successfully deobfuscates Android APKs produced by ProGuard [6], the most popular obfuscation tool for Android applications.



*CCS'16, October 24 - 28, 2016, Vienna, Austria*



ISBN 978-1-4503-4139-4/16/10…$15.00

DOI: http://dx.doi.org/10.1145/2976749.2978422

### *Focus: Layout Deobfuscation.*

The focus of this paper is on reversing layout obfuscation of Android APKs. While general obfuscation can include other transformations (e.g., changes to the program's data representation or control-flow [25]), layout obfuscation remains a key part of virtually all obfuscation tools. In layout obfuscation, the names of program elements that carry key semantic information are replaced with other (short) identifiers with no semantic meaning. Examples of such elements are comments, variable, method and class names. Renaming these program elements makes it much harder for humans to read and understand what the program does and is useful in a variety of security scenarios including protection of intellectual property.

### *Benefits and Challenges.*

Among others, reversing layout obfuscation for Android APKs has various benefits including: (i) it makes it easier for security analysts to inspect Android applications obfuscated with ProGuard, (ii) it identifies third-party libraries embedded in Android APKs, and (iii) it enables one to automatically search for certain identifiers in the code.

However, reversing layout obfuscation is a hard problem. The reason is that once the original names are removed from the application and replaced with short meaningless identifiers, there is little hope in recovering the original names by simply inspecting the application alone, in isolation.

### *Probabilistic Learning from "Big Code".*

To address challenges that are difficult to solve by considering the program in isolation, the last couple of years have seen an emerging interest in new kinds of statistical tools which learn probabilistic models from "Big Code" and then use these models to provide likely solutions to tasks that are difficult to solve otherwise. Examples of such tasks include machine translation between programming languages [18], statistical code synthesis [32, 30], and predicting names and types in source code [31, 9]. Interestingly, due to their unique capabilities, some of these probabilistic systems have quickly become popular in the developer community [31].

*This Work: Android Deobfuscation via "Big Code".*

Motivated by these advances, we present a new approach for reversing Android layout obfuscation by learning from thousands of readily available, non-obfuscated Android applications. Technically, our approach works by phrasing the problem of predicting identifier names (e.g., class names, method names, etc.) renamed by layout obfuscation as structured prediction with probabilistic graphical models. In particular, we leverage Conditional Random Fields (CRFs) [23], a powerful model widely used in various areas including computer vision and natural language processing. To our knowledge, this is the first time probabilistic graphical models learned from "Big Code" have been applied to address a core security challenge. Using our approach we present a tool called DeGuard, and show that it can automatically reverse layout obfuscation of Android APKs as performed by ProGuard with high precision.

*Main Contributions.*

The main contributions of this paper are:

- A structured prediction approach for performing probabilistic layout deobfuscation of Android APKs.
- A set of features and constraints cleanly capturing key parts of Android applications. Combined, these ensure our probabilistic predictions result in high precision *and* preserve application's semantics.
- A complete implementation of our approach in a scalable probabilistic system called DeGuard[1].
- An evaluation of DeGuard on open-source Android applications obfuscated by ProGuard and Android malware samples. Our results show that DeGuard is practically effective: it correctly predicts 79.1% of the program elements obfuscated by ProGuard, it identifies 91.3% of the imported third-party libraries, and reveals relevant string decoders and classes in malware.

# 2. OVERVIEW

In this section we provide an informal overview of our statistical deobfuscation approach for Android. First, we discuss ProGuard, which is the most widely used tool for obfuscating Android applications. We then present the key steps of our DeGuard system. The purpose here is to provide an intuitive understanding of the approach. Full formal details are presented in the later sections.

## 2.1 ProGuard

ProGuard obfuscates program elements including names of fields, methods, classes, and packages, by replacing them with semantically obscure names. It also removes unused classes, fields, and methods to minimize the size of the resulting Android application package (APK) released to users. ProGuard processes both the application and all third-party libraries that the application imports (e.g., advertising and analytics libraries). All third-party libraries imported by the application are therefore concealed in the released APK.

ProGuard cannot obfuscate *all* program elements as that would change the application's semantics. For example, the names of methods part of the Android API and the names of classes referenced in static files, are kept intact.

[1]http://apk-deguard.com

*Example.*

Figure 1(a) shows a fragment of an Android application that has been obfuscated with ProGuard (the obfuscated program elements are highlighted with red). The depicted code fragment can be easily obtained from the APK using standard tools, such as Dex2Jar [2] and Java Decompiler [4].

Here, the name of the class is replaced with the non-descriptive name `a` and similarly, the private field of type `SQLiteDatabase` and the method returning a `Cursor` object are renamed with the obscure names `b` and `c`, respectively. It is evident that inspection of this code, as well as any other code using the obfuscated class `a`, is challenging. For example, the intended behavior of the following two lines of code is concealed due to the non-descriptive class and method names:

```
a obj = new a();
obj.c(str);
```

As mentioned, ProGuard keeps the names of some program elements to preserve the application's semantics. For example, the name of the class `SQLiteOpenHelper` and its methods `getWritableDatabase` and `rawQuery` are not renamed because this class is part of the core Android API.

## 2.2 DeGuard

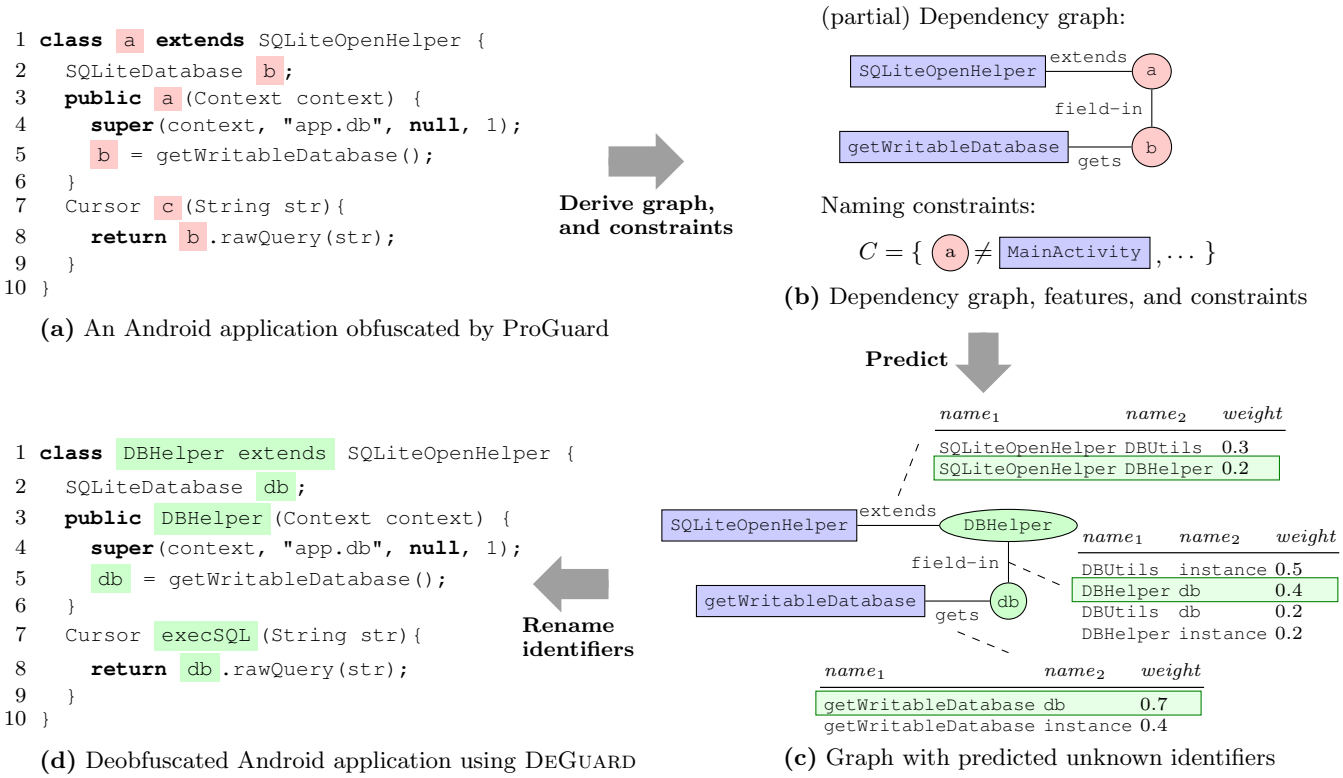
Given an Android APK as input, DeGuard outputs a semantically equivalent Android APK with descriptive names for fields, methods, classes, and packages. We depict the source code of the output APK produced by DeGuard in Figure 1(d). The key steps of our approach are shown with thick gray arrows ( ➡ ) in Figure 1. We now describe these steps.

*Dependency Graph.*

DeGuard analyzes the input APK and formalizes the structure of the Android application as a graph over the program elements, where an edge signifies that the corresponding two program elements are related. The graph in Figure 1(b) illustrates a fragment of the generated dependency graph for our example.

The red circular nodes denote the *unknown* (i.e., obfuscated) program elements whose names the tool will try to predict, and the purple rectangular nodes are the *known* program elements, which will not be modified by the tool. The name of the class `a` is therefore represented with a red node, while the class `SQLiteOpenHelper` with a purple one. The graph's edges are labeled with a particular *relationship*, which represents how the two program elements are related. For example, the edge from node `a` to node `SQLiteOpenHelper` is labeled with the relationship `extends` to formalize that the former class extends the latter. Since program elements can have multiple relationships, the dependency graph may in general contain multiple edges between two nodes (thus, technically, the dependency graph is a multigraph).

Formally, the relationships between two nodes represent different feature functions. The constructed dependency graph, along with all feature functions specifies a Conditional Random Field (CRF) [23], a powerful probabilistic graphical model. We formally define the dependency graph, feature functions, and CRFs in Section 3, and the features for Android applications in Section 4.




**(a)** An Android application obfuscated by ProGuard

**(b)** Dependency graph, features, and constraints

**(c)** Graph with predicted unknown identifiers

**(d)** Deobfuscated Android application using DeGuard

Figure 1: Statistical deobfuscation of Android applications using DeGuard. The red color indicates the elements whose names are to be renamed (in the input), the green color are the same elements with the new names (in the output), and the purple color denotes the elements whose names are known and remain the same.

### *Syntactic and Semantic Constraints.*

Given an Android APK, DeGuard automatically derives a set of *constraints* which restricts the possible names assigned to the unknown program elements. These naming constraints guarantee that the deobfuscated APK generated by DeGuard is: (i) a syntactically well-formed program, and (ii) semantically equivalent to the input APK. Two example constraints are: all fields declared in the same class must have distinct names and all classes that belong to the same package must have distinct names. Any well-formed application must satisfy these two syntactic properties. Naming constraints of methods are more intricate due to method overriding. For example, if a method in a subclass overrides a method in a superclass (in the input APK), then the two methods must have the same name after deobfuscation to preserve the overriding property.

For example, suppose the package of class `a` also contains a class with the (non-obfuscated) name `MainActivity`. The constraint `a` $\neq$ `MainActivity` in Figure 1(b) specifies that the predicted name for node `a` must be distinct from the name `MainActivity`. Indeed, if these two classes have identical names, then the resulting output APK would not be syntactically well-formed.

In Section 5, we describe an algorithm that, for any APK, generates all necessary syntactic and semantic constraints.

### *Probabilistic Prediction.*

Using the derived dependency graph and constraints, DeGuard infers the most likely names for all obfuscated elements. The predicted names for our example are depicted in Figure 1(c). DeGuard predicts that the name of the obfuscated class `a` is `DBUtils` and that the name of the obfuscated field `b` is `db`. Below, we describe how DeGuard concludes that these are the most likely names for this example.

To predict the names of the obfuscated elements, DeGuard performs a *joint* prediction that considers *all* program elements, known and unknown. To illustrate this inference step, consider the graph in Figure 1(c). The tables associated with the graph's edges represent the likelihood, of each (pairwise) assignment of names to program elements (nodes). We remark that each table is derived from feature functions associated with weights, which together represent (log-)likelihoods. We formally define feature functions and explain the derivation of the likelihood tables in Section 3. Here, we illustrate how these likelihood tables are used to choose the most likely names. Our goal is to find an assignment for *all* program elements that maximizes a *score* that is the sum of the weights in each table.

For our example, according to the top-most table, the weight of assigning the name `DBUtils` to the class is 0.3. However, DeGuard does not select `DBUtils` as the name of this class. This is because selecting the name `DBUtils` does not result in the highest possible overall score. Suppose we select the name `DBUtils`. Then, we have two possible names for the obfuscated field `b`, namely `db` and `instance`. According to the likelihood tables, both the former and the latter choice result in a total score of 1.2. However, if we select the name `DBHelper` and `db`, then the total score is 1.3, which is the highest possible score for this example. DeGuard therefore returns these names as most likely.

Formally, DeGuard performs a *Maximum a Posteriori* (MAP) inference query on the CRF model defined by the dependency graph and the feature functions. We define MAP inference in Section 3.

## 2.3 Security Applications

DeGuard can be used to tackle several practical security problems. In our evaluation, we show that DeGuard can effectively reverse ProGuard's layout obfuscation for benign Android APKs. Although ProGuard obfuscates 86.7% of the program elements on average, DeGuard correctly predicts the names for 79.1% of those elements.

Predicting libraries is another important problem, which is particularly relevant for Android [14]. Mobile developers tend to rely on a large number of libraries which often contain security vulnerabilities — from personal information release [15, 11] to severe man-in-the-middle vulnerabilities [1]. In our experiments, DeGuard reveals over 90% of the third-party libraries concealed by ProGuard.

Further, numerous security analyses rely on descriptive program identifiers. Examples include analyses that perform statistical filtering of potential vulnerabilities [37] and probabilistic systems for detecting privacy leaks [11]. These systems assume that the application's program elements are non-obfuscated. DeGuard can be used to deobfuscate applications before they are analyzed by such systems.

## 2.4 Challenges

We discuss three key challenges when building a prediction system for deobfuscating Android applications:

(i) *Capturing the rich structure of Android applications*: precisely encoding the structure of Android applications using a concise, yet adequate set of program elements and relationships is important to ensure the predictions made by the system are accurate. This is difficult as a large set of relationships may hurt the scalability of the system while missing important relationships, or defining bad ones, can reduce the prediction accuracy.

(ii) *Semantic equivalence*: the rich structure of Java poses nontrivial constraints that must be captured to ensure the resulting deobfuscated Android APK has equivalent semantics to the input APK.

(iii) *Scalable learning*: the expressive structure of Android applications inevitably results in complex dependency graphs and a large variety of features that cannot be handled efficiently by off-the-shelf machine learning systems. According to our experiments, the most scalable available prediction system for programs [31] required an order of magnitude longer than DeGuard to learn a probabilistic model for Android.

## 2.5 Scope and Limitations

In this work, we focus on deobfuscating Android applications that have been transformed using layout obfuscation mechanisms, which rename fields, methods, classes and packages with semantically obscure names. Other obfuscation techniques, such as data obfuscation mechanisms, which alter data structures, control-flow and cryptographic obfuscation mechanisms fall outside the scope of this work. We remark that malicious Android applications often uses multiple obfuscation techniques to prevent reverse engineering. Security experts must thus use a combination of deobfuscation tools to effectively deobfuscated Android malware.

# 3. BACKGROUND

In this section we provide the necessary background on probabilistic models, queries, and learning which we leverage in this work. These concepts are well known in the field of probabilistic graphical models [20]. The main purpose here is to review these parts and to illustrate how they are used by our approach.

### *Problem Statement.*

We phrase the problem of predicting the most likely names assigned to all obfuscated program elements as a problem in structured prediction. Intuitively, we model the elements of a program as a tuple of random variables $(V_1, \ldots, V_n)$ ranging over a set of name labels *Names*. The set *Names* in our case contains all possible names from which we can choose to name program elements. Then, the joint distribution $P(V_1, \ldots, V_n)$ (discussed later in this section) over these variables assigns a probability to each assignment of names to variables.

Let $\vec{O} = (V_1, \ldots, V_{|\vec{O}|})$ be the variables representing *obfuscated* program elements, i.e., the variables whose names we would like to predict. The names of the remaining variables $\vec{K} = (V_{|\vec{O}|+1}, \ldots, V_n)$ are *known* and will not be affected by the renaming. Then, to predict the most likely names for the obfuscated program elements, we compute the Maximum a Posteriori (MAP) inference query:

$$\vec{o} = \underset{\vec{o'} \in \Omega}{argmax}\ P(\vec{O} = \vec{o'} \mid \vec{K} = \vec{k})$$

where $\Omega \subseteq Names^{|\vec{O}|}$ is the set of all possible assignments of names to the obfuscated variables $\vec{O}$, and $\vec{k} \in Names^{|\vec{K}|}$ defines the names assigned to the known variables $\vec{K}$. Next, we describe how we actually represent and compute the conditional probability $P(\vec{O} = \vec{o} \mid \vec{K} = \vec{k})$ for a given assignment of names $\vec{k}$.

### *Dependency Graph.*

A *dependency graph* for a given program is an undirected multigraph $G = (V, E)$, where $V$ is a set of random variables representing program elements and $E \subseteq V \times V \times Rels$ is a set of labeled edges. Here, *Rels* is a set of relationships between program elements; we instantiate this set for Android applications in Section 4. An edge $(V_i, V_j, rel)$ says that elements $V_i$ and $V_j$ are related via *rel*. An example of a dependency graph is shown in Figure 1(b).

### *Features and Weights.*

We define a *pairwise feature function* $\varphi$ as follows:

$$\varphi : Names \times Names \times Rels \to \mathbb{R}$$

This function maps a pair of names and their relationship to a real number. In Section 4, we define several kinds of feature functions and based on these we obtain the entire set of pairwise features $\{\varphi_1, \ldots, \varphi_m\}$ automatically during the learning phase (described at the end of this section). For example, for each observed edge $(V_i, V_j, rel)$ in the training set of dependency graphs where the names assigned to $V_i$ and $V_j$ are $n_i$ and $n_j$, respectively, we define a pairwise feature $\varphi(N, N', Rel) = 1$ if $N = n_i$, $N' = n_j$, and $Rel = rel$; otherwise, $\varphi_i(N_1, N_2, rel) = 0$. Further, for any $\varphi_i$, we associate a weight $w_i$, also computed during the learning phase.

Given a dependency graph $G = (V, E)$, a prediction $\vec{o}$ for the obfuscated variables $\vec{O}$, and an assignment $\vec{k}$ for the fixed, known variables $\vec{K}$, we associate a *feature function* $f_i$ to each pairwise feature $\varphi_i$ defined as follows:

$$f_i(\vec{o}, \vec{k}) = \sum_{(V_j, V_l, rel) \in E} \varphi_i((\vec{o}, \vec{k})_j, (\vec{o}, \vec{k})_l, rel)$$

Here, $(\vec{o}, \vec{k})_j$ denotes the $j$th name in the vector $(\vec{o}, \vec{k})$. We can think of $f_i$ as lifting $\varphi_i$ to quantify $\varphi_i$'s effect on all the edges in the graph (i.e., adding up $\varphi_i$'s effect on each edge). The end result computed by $f_i$ is a real number capturing the overall effect of $\varphi_i$.

### *Conditional Random Fields.*

A *conditional random field* (CRF) is a probabilistic model which defines a conditional probability distribution, that is, $P(\vec{O} = \vec{o} \mid \vec{K} = \vec{k})$ as follows:

$$P(\vec{O} = \vec{o} \mid \vec{K} = \vec{k}) = \frac{1}{Z} exp(\sum_{i=1}^{m} w_i f_i(\vec{o}, \vec{k})),$$

where each $f_i$, $1 \leq i \leq m$, is a feature function associated with a weight $w_i$, and $Z$ is a normalization constant. We do not define $Z$ as it can be omitted for our specific type of query.

It is then immediate that the dependency graph, together with the feature functions $f_1, \ldots, f_m$ and their associated weights $w_1, \ldots, w_m$, define a CRF.

### *Prediction via MAP Inference.*

To compute the most likely assignment $\vec{o}$ for the variables $\vec{O}$, we perform a MAP inference query:

$$\vec{o} = \underset{\vec{o'} \in \Omega}{argmax}\ P(\vec{O} = \vec{o'} \mid \vec{K} = \vec{k})$$

Using our CRF model, we can compute the probability of an assignment $\vec{o}$ using the formula:

$$P(\vec{O} = \vec{o'} \mid \vec{K} = \vec{k}) = \frac{1}{Z} exp(\sum_{i=1}^{m} w_i f_i(\vec{o}, \vec{k}))$$

We omit the constant $Z$ (as it does not affect the result of the MAP inference query), expand $f_i(\vec{o}, \vec{k})$, and rewrite the formula as follows:

$$\begin{aligned} P(\vec{O} = \vec{o'} \mid \vec{K} = \vec{k}) \sim \\ \sim exp(\sum_{i=1}^{m} w_i \sum_{(V_j, V_l, rel) \in E} \varphi_i((\vec{o}, \vec{k})_j, (\vec{o}, \vec{k})_l, rel)) = \\ = exp(\sum_{(V_j, V_l, rel) \in E} \sum_{i=1}^{m} w_i\ \varphi_i((\vec{o}, \vec{k})_j, (\vec{o}, \vec{k})_l, rel)) \end{aligned}$$

We refer to the above as the total score of an assignment $(\vec{o}, \vec{k})$.

### *MAP Inference Example.*

We now explain the above equation by referring to our example from Section 2. The product $w_i\ \varphi_i((\vec{o}, \vec{k})_j, (\vec{o}, \vec{k})_l, rel)$ scores a particular pairwise feature function $\varphi_i$. In our example, each row in the tables given in Figure 1(c) defines a pairwise feature function and its weight. Consider the first row of the top-most table. This row denotes a pairwise feature function which returns 1 if its inputs are (`SQLiteOpenHelper`, `DBUtils`, `extends`) and 0 for all other inputs. That feature function also has a weight of $w_i = 0.3$, which is determined during learning. We do not include the type of relationship in the tables since in this example the program elements are connected via a single relationship. In our example, the MAP inference query will return the assignment $\vec{o}$ highlighted in green (i.e., `DBHelper` and `db`) as that assignment satisfies the constraints in $\Omega$ (in our example we have a single inequality constraint) and the total score of $\vec{o}$ is the highest: $0.2 + 0.4 + 0.7 = 1.3$. To compute this score, DeGuard implements a greedy MAP inference algorithm which we describe in Section 6.1.

### *Learning from "Big Code".*

The input to the learning phase of DeGuard is a set of $p$ programs $\{\langle \vec{o}^{(j)}, \vec{k}^{(j)} \rangle\}_{j=1}^{p}$ for which both vectors of names $\vec{o}$ and $\vec{k}$ are given. That is, the training data contains non-obfuscated applications, which can be downloaded from repositories for open-source Android applications, such as F-Droid [3]. From this input, the learning outputs weights $\{w_i\}_{i=1}^{m}$ such that names in the training data programs are correctly predicted. There are several variations of this learning procedure [29, 20]. For our application we use learning with pseudo-likelihoods as described in [35, §5.4]. We describe this algorithm in more detail in Section 6.1.

## 4. FEATURE FUNCTIONS

In this section, we present the pairwise feature functions used for our Android deobfuscation task. As described earlier in Section 3, these feature functions are used to build the dependency graph. Recall that the construction of an application's dependency graph amounts to introducing a node for each program element and then connecting the program elements that are related. The signature of a dependency graph is therefore defined by the application's *program elements* and the *relationships* between them.

### 4.1 Program Elements

A program's dependency graph is defined over nodes that represent different program elements. To capture the structure of an Android application, we introduce nodes for each of the following program elements:

- **Types.** We introduce a node for each primitive type (e.g., **`int`**, **`long`**, **`float`** etc.), reference type (e.g., `Object`, `ArrayList`, etc.), and array type (e.g., **`int`**`[]`, `Object[]`, etc.) that appears in the application. For example, we introduce a node to represent the reference type `SQLiteDatabase` in the example of Figure 1(a).
- **Fields.** We introduce a node for each field declared in the application's classes. For example, we introduce a node to represent the field of type `SQLiteDatabase` in the example of Figure 1(a).
- **Packages.** We introduce a node for each package in the application. For example, given a package `a.b`, we introduce two nodes: one to represent the package `a`, and another to represent the package `a.b`.
- **Methods.** We introduce a node for each method declared in the application's classes. For the example of Figure 1(a), we introduce two nodes: one node to represent the constructor `<init>()` and another node for

the method `c()`. If a class overrides a given method, we use one node to represent both the method declared in the superclass and the one declared in the subclass. This guarantees that overriding methods are renamed consistently, which is necessary for preserving the application's semantics.

- **Expressions.** We introduce a node to represent constant values (e.g., integers, strings, etc.) and the value **`null`**. For example, we add a node `5` to capture the constant value 5. Nodes for other kinds of constant values are introduced analogously.
- **Access Modifiers.** We introduce nodes to represent access modifiers, such as **`static`**, **`synchronized`**, **`private`**, **`protected`**, **`public`**, and so forth.
- **Operations**. We introduce nodes to represent operations (e.g., `+`, `-`, etc.).

We remark that we ignore generic types because they are removed by the compiler during the type erasure process [7]. Furthermore, we ignore the names of local variables and method parameters since they are not part of the application's APK. For example, we do not introduce a node to represent the method parameter's name `str` of the method `c()` in Figure 1(b). We do however capture the types of local variables and method parameters, e.g. we capture that the method `c()` has a parameter of type `String`.

### *Known and Unknown Program Elements.*

We capture whether a node's name is obfuscated and is to be predicted, or is known and should not be predicted, using the following set of rules:

- Nodes that represent packages, classes, methods, and fields that are part of the Android API are known. For example, the node of the class `SQLiteOpenHelper` in our example is known because this class is part of the Android API. All program elements that are part of the Android API are referred by their name, and thus any obfuscator keeps their names intact.
- Constructor methods (both dynamic and static) have fixed names and are thus known.
- If a method overrides a known method (e.g., a method that is part of the Android API), then the nodes representing both methods are known. We enforce this rule implicitly by keeping a single node to represent both methods. We explain this shortly.
- All remaining packages, classes, methods, and fields are unknown.

### *Grouping Method Nodes.*

In the context of inheritance and method overriding, introducing a node for each declared method leads to issues. The reason is that the two methods must have the same signature, where a method's signature is defined by the method's name along with the number and types of its parameters. To guarantee that the deobfuscation is semantic-preserving, we combine all methods related via inheritance in a single node. We refer to the process of combining all such methods as *grouping*.

To detect all overrides that occur in a given set of classes, for each class we collect all of the methods it implements. Then, for each of these methods, we link it with all the methods it overrides. Finally, we combine all methods that are (possibly indirectly) linked together into a single node.

## 4.2 Relationships

To capture the structure of an Android application, we introduce relationships between its program elements. We define all such relationships in Figure 2. The second column in the table defines the type of each edge. For example, the first edge type is ($m$, $op$, `performs-op`). This edge type says that it connects a method $m$ to an operation $op$ with the relationship `performs-op`. The third column specifies under what condition the edge is added between two nodes (of the correct type). The edge of the first type is added whenever the method $m$ performs an operation $op$. The types $m$ and $op$ are the ones we already defined in Section 4.1. We organize the relationships into two broad categories: method relationships and structural relationships.

### *Method Relationships.*

Method relationships capture the semantic behavior of methods. This is important because method names typically describe the method's behavior. For example, the method name `execSQL` in our example of Figure 1 describes that this method executes an SQL command. We remark on several points pertaining to method relationships. First, the program elements denoted by $o$, $arg$ and $v$ are not necessarily fields. For example, the method call `field.foo().bar()` results in two edges of type `receiver`: (`foo`, `bar`, `receiver`) and (`field`, `foo`, `receiver`). Second, for every loop occurring in a method, we capture how different values and fields are used within the loop using the relationships `loop-read` and `loop-write`. To capture which classes are accessed by a method, we introduce the relationship `writes-classfield` and the relationship `calls-classmethod`.

Finally, we remark that the relationships defined above, in addition to capturing the semantics of methods, also capture how fields, classes and methods *are used* by the application. For example, adding the information that method $m$ reads field $f$ also conveys that $f$ is read by $m$.

### *Structural Relationships.*

The structural relationships capture the relations between the nodes, such as whether two classes are defined within the same package or not. These features are particularly important for predicting obfuscated third-party libraries, as the correct prediction of a small number of classes within the library's package significantly aids to accurately predict the library's remaining program elements. Note that for method parameters, we express not only their type, but also how often they occur. This is captured with the relationship `argtype-N`. Further, we define the two relationships `read-before` and `written-before` to capture the order of reads and writes to fields.

### *Comparison to Other Prediction Systems.*

Android applications have significantly more complex structure compared to programs encoded in, e.g., untyped, dynamic languages. Precisely capturing this structure is key to enable the accurate prediction of Android applications. DeGuard is the first prediction system for programs that sup-

| Relationship | Type | Condition for Relating Two Program Elements |
|---|---|---|
| | *Method Relationships* | |
| Method operation | ($m$, $op$, `performs-op`) | method $m$ performs operation $op$ (e.g. addition +, xor, etc.) |
| | ($m$, $t$, `performs-cast`) | method $m$ performs a cast to type $t$ |
| | ($m$, $t$, `instance-of`) | method $m$ performs an **`instanceof`** check for type $t$ |
| | ($m$, $e$, `returns`) | method $m$ returns an expression $e$ (e.g. a field or a method call) |
| Reads and writes | ($m$, $e$, `uses`) | expression $e$ appears in $m$ |
| | ($m$, $f$, `writes`) | method $m$ modifies field $f$ |
| Arguments and receivers | ($m$, $arg$, `flows-into`) | there is a call $o.m(\ldots, arg, \ldots)$ with argument $arg$ |
| | ($o$, $m$, `receiver`) | there is a call $o.m(\ldots)$ |
| Loops | ($v$, $m$, `loop-read`) | method $m$ uses the value $v$ within a loop |
| | ($f$, $m$, `loop-write`) | method $m$ writes to the field $f$ within a loop |
| Accessed Classes | ($m$, $c$, `writes-classfield`) | class $c$ that contains a field that is read by $m$ |
| | ($m$, $c$, `calls-classmethod`) | class $c$ that contains a method called by $m$ |
| | *Structural Relationships* | |
| Packages | ($e$, $p$, `contained-in-package`) | package $p$ contains a class, a method, or a field $e$ |
| | ($p_1$, $p_2$, `direct-subpackage-of`) | package $p_1$ that is directly contained within a package $p_2$ |
| | ($p_1$, $p_2$, `subpackage-of`) | package $p_1$ is contained within a package $p_2$, but not directly |
| Classes | ($f$, $c$, `field-in`) | field $f$ is declared in a class $c$ |
| | ($m$, $c$, `method-in`) | method $m$ declared in a class $c$ |
| | ($c_1$, $c_2$, `overrides`) | class $c_1$ overrides a program element in class $c_2$ |
| | ($c$, $i$, `implements`) | class $c$ that implements an interface $i$ |
| | ($c_1$, $c_2$, `extends`) | class $c_1$ extends class $c_2$ |
| Types | ($f$, $t$, `field-type`) | the type of field $f$ is $t$ |
| | ($m$, $t$, `return-type`) | method $m$ that returns an object of return type $t$ |
| | ($m$, $t$, `argtype-N`) | method $m$ has $N$ parameters of type $t$ |
| Access modifiers | ($e$, $am$, `has-modifier`) | method or field $e$ has access modifier $am$ |
| Fields | ($f$, $e$, `gets`) | assignment expression $f = e$ (e.g. a field name or a method call) |
| | ($f$, $e$, `initialized-by`) | there is an initialization statement statements $f = e$ |
| | ($f_1$, $f_2$, `read-before`) | field $f_1$ is read before field $f_2$ |
| | ($f_1$, $f_2$, `written-before`) | field $f_1$ is written before field $f_2$ |

Figure 2: Relationships used to relate the program elements of Android applications. The second column defines the edge type (i.e. the program elements it related). Each relationship is added if the condition in the third column is true.

ports a rich set of structural relationships, including types, structural hierarchies, and access modifiers. In our evaluation, we show that DeGuard strikes a balance between the accuracy and efficiency of prediction: the set of relationships defined above are sufficient to accurately predict a significant part (roughly 80%) of the program elements obfuscated by ProGuard, while keeping the complete prediction time reasonable (under a minute on average).

## 4.3 Pairwise Feature Functions

The pairwise features $\varphi_i$ are derived from the relationships defined above, based on the relationships observed in the dependency graphs used in the learning phase. Formally, let $G_1 = (V_1, E_1), \ldots, G_m = (V_m, E_m)$ be the set of dependency graphs used in the learning phase with naming assignments. For each edge $(V_i, V_j, rel) \in E_1 \cup \ldots \cup E_m$ that appears in the dependency graphs, we define a pairwise feature

$$\varphi_i(N, N', Rel) = \begin{cases} 1 & \text{if } N = n_i, N' = n_j, Rel = rel \\ 0 & \text{otherwise} \end{cases}$$

where $n_i$ and $n_j$ are the names assigned to the program elements denoted by $V_i$ and $V_j$. The pairwise features define an indicator function for each pair of labels and kind of relationship observed in the training set of non-obfuscated programs. While the pairwise features are derived from the training data, the weights associated to these features are learned during the learning phase.

# 5. CONSTRAINTS

In this section we define the constraints that our deobfuscation mechanism must satisfy while renaming program elements to ensure both syntactic and semantic validity of the deobfuscated application. First, we describe method naming constraints, which are more complex to define due to method overrides. Afterwards, we describe naming constraints for fields, classes, and packages.

## 5.1 Naming Constraints for Methods

Method naming constraints are necessary for both semantic reasons and for syntactic well-formedness. According to Java's semantics, whenever a class extends another class, the method declared in the subclass overrides a method declared in the super class if the two methods have the same

```
1 public class A {
2    public void a(A a) {}
3    public void b(Object o) {}
4    public void c() {}
5 }
6 public class B extends A {
7    public void g() {}
8    private int h() {}
9 }
10 public class C extends B {
11    public void x() {}
12 }
```

Figure 3: An example that illustrates different types of method naming constraints

signature, i.e. the same name and list of parameter types. Method overrides thus change the application's semantics. Further, all methods within the same class must have distinct signatures. Below, we give an example to illustrate different kinds of method naming constraints. Afterwards, we describe how DeGuard derives these constraints.

### *Example.*

We illustrate method naming constraints with an example. Consider the program in Figure 3. Here we have three classes that exemplify different cases of method naming constraints.

The name of the method `A.a(A)` is not constrained by any method declared in Figure 3 because it has a unique list of parameter types. Here, the method `A.a(A)` is the only method that has one parameter of type `A`. In contrast, `A.b(Object)` cannot be renamed to `equals`, because then it would override `java.lang.Object.equals(Object)` from the Java standard library. This constraint is needed because `A` implicitly extends `Object` and the list of parameter types `a` matches that of the method `equals` declared in the class `Object`.

The names of the methods `B.g()` and `B.h()` must be distinct even though their return types and access level modifiers are different. This is because neither the return type nor the access level modifier is part of a method signature, and therefore renaming both methods to the same name results in the same method signature.

The names of the methods `B.g()` and `A.c()` must be distinct due to the semantics of method overriding in the presence of inheritance. Here, the class `B` extends `A`. Therefore, a potential change of the name of `B.g()` to that of `A.c()` would result in *overriding* method `A.c()`. The names of the methods `B.h()` and `A.c()` must be also distinct due to the semantics of method overriding, even though `B.h()` is private. The method `C.x()`'s name is not constrained by the name of `B.h()` because `B.h()` is declared as **`private`**. According to Java's semantics, no method may override private methods.

### *Expressing Method Constraints.*

Our example shows that in addition to equality constraints, inequality constraints are also needed to formalize all naming properties of methods. Equality constraints can be handled implicitly by representing methods that must have an identical signature with one node in the dependency graph (see Section 4). This guarantees that all such methods are renamed consistently. Inequality constraints, necessary to avoid accidental overrides due to inheritance, must be explicitly specified. We specify inequality constraints as sets of program elements that have distinct names.

**Algorithm 1** Detecting inequality constraints for method names

```
1: function findConstraints
2:     object ← java.lang.Object
3:     handleClass(object, ∅)
4: end function
5: function handleClass(class, aboveMethods)
6:     methods ← aboveMethods ∪ class.nonPrivateMethods()
7:     reportConstraints(methods ∪ class.privateMethods())
8:     subClasses ← classes that directly extend/implement class
9:     for subClass ∈ subClasses do
10:        handleClass(subClass, methods)
11:    end for
12: end function
13: function reportConstraints(methods)
14:    ▷ Report all inequality constraints for methods
15:    partitions ← partition methods by parameter types
16:    for partition ∈ partitions do
17:        report partition as an inequality constraint
18:    end for
19: end function
```

Formally, let $V = \{V_1, \ldots, V_n\}$ be the set of nodes in the dependency graph. We define an *inequality constraint* as a set of nodes $C \subseteq V$. An assignment $\vec{y} = (y_1, \ldots, y_n)$ of names to program elements *satisfies* the inequality constraint $C$ if $\forall V_i, V_j \in C.\ y_i \neq y_j$. For example, to specify that the methods `C.x()` and `B.g()` must have distinct names, we use the inequality constraint $C = \{$`C.x()`, `B.g()`$\}$. Note that we define inequality constraint using *sets* of elements whose elements must be pairwise distinct, as opposed to standard binary inequality constraints (e.g., `C.x()` $\neq$ `B.g()`) because the encoding of the former is more concise.

### *Deriving Inequality Constraints for Methods.*

We next describe how we derive inequality constraints for methods. Without loss of generality, we treat interfaces identically to classes. The inequality constraints for methods are derived using Algorithm 1. The function findConstraints calls handleClass with the class `Object` and the empty set of method names (since `Object` has no super classes). The recursive call on Line 10 reaches all other classes as every class (transitively) extends `Object`.

The function handleClass(`class`, `aboveMethods`) reports all inequality constraints for `class`, where the parameter `aboveMethods` contains all methods that the methods declared in `class` can potentially override.

The function reportConstraints reports the inequality constraints for the methods contained in `methods`. To do this, it first partitions the `methods` based on their parameter types. All methods in a given partition must have distinct names, because otherwise, they would have the same signature. On the other hand, no method is constrained by the methods in the other partitions.

### *Result on the Example.*

Here, we show the result of applying Algorithm 1 on the program in Figure 3. For simplicity, we assume that `A` does not extend `java.lang.Object`. More precisely, a call to handleClass(`A`, $\varnothing$) results in the inequality constraints {`A.c()`, `B.g()`, `B.h()`} and {`A.c()`, `B.g()`, `C.x()`}. Note that we implicitly remove singleton inequality constraints as these are always satisfied.

## 5.2 Naming Constraints for Fields, Classes, and Packages

The deobfuscation mechanism must satisfy the following properties: (i) any two packages contained in the same package must have distinct names, (ii) any two classes contained in the same package must have distinct names, and (iii) any two fields declared in the same class must have distinct names. We remark that the types of fields are irrelevant for naming constraints. This is because a field is referred to by its name, and the type of a field is not part of this name. This is in contrast to methods, which are called by their signature and where the types of a method's parameters are part of the method's signature.

For a given Android app, the naming constraints for fields, classes, and packages, are formalized using inequality constraints in the same way we formalize the constraints for method names. We derive all inequality constraints for fields, classes, and packages, by iterating over all classes and packages and reporting inequality constraints as defined by the above properties.

# 6. IMPLEMENTATION AND EVALUATION

In this section we describe the implementation of DeGuard and the experiments we conducted with it.

## 6.1 The DeGuard System

We now present our DeGuard system, which is publicly available at http://apk-deguard.com. DeGuard is implemened using Soot [38], a framework for static analysis of Java and Android applications. Given an Android APK, Soot transforms it into an intermediate format (called Jimple) that simplifies the analysis of the application. To construct the application's dependency graph, we use Soot's API to traverse all program elements.

To predict the names of all obfuscated elements for a given application, DeGuard performs a MAP inference query on the CRF model constructed from the application's program elements, the set of pairwise features (described in 4), and the feature weights. Next, we describe how DeGuard learns a probabilistic model (the pairwise features and their weights) from non-obfuscated Android applications, and how it uses this probabilistic model to predict likely names of obfuscated program elements using MAP inference.

### *Feature Functions and Weights.*

To learn all feature functions and weights, we downloaded 1784 non-obfuscated Android applications from F-Droid [3], a popular repository for open-source Android applications. Out of these 1784 applications, we randomly selected 100 which we intentionally left as our *benchmark* applications, i.e., the ones we later use in our evaluation. We used the remaining 1684 applications as our *training* set of applications.

The set of possible names assigned to obfuscated program elements is drawn from the names observed in the training set. The pairwise features $\varphi_1, \ldots, \varphi_m$ are also derived from the training set, as described in Section 4.3. The only component missing in our probabilistic model are weights $\vec{w} = [w_1, \ldots, w_m]$ associated with the pairwise features. One way to learn the weights is to use *maximum likelihood estimation*, where the weights $\vec{w}$ are chosen such that the training data has the highest probability. That is, we chose weights $\vec{w}$ that maximize the probability $P(\vec{O} = \vec{o}^j \mid \vec{K} = \vec{k}^j)$, computed as defined in Section 3, for all programs $\langle \vec{o}^j, \vec{k}^j \rangle$ in the training set. Unfortunately, computing the weights using precise maximum likelihood estimation is prohibitively expensive in our context, due to the large number of nodes and possible labels that can be assigned to them. DeGuard therefore learns the weights using *pseudo likelihood*, which approximates the conditional distribution $P(\vec{O} \mid \vec{K})$ as the product of the conditional distributions $P(O_i \mid N(O_i), \vec{K})$ of each unknown node $O_i \in \vec{O}$ conditioned on the node's neighbors $N(O_i)$ and the known nodes $\vec{K}$. For the complete details on training using pseudo likelihoods see [35, §5.4].

Using the training described above, the training of this model took about 2 hours on a 32-core machine with four 2.13GHz Xeon processors running Ubuntu 14.04 with 64-Bit OpenJDK Java 1.7.0_51.

### *MAP Inference.*

To predict likely names $\vec{o}$ to be assigned to all obfuscated elements $\vec{O}$, DeGuard computes the MAP inference query

$$\vec{o} = \underset{\vec{o'} \in \Omega}{argmax}\ P(\vec{O} = \vec{o'} \mid \vec{K} = \vec{k})$$

where $\vec{k}$ are the names assigned to the known elements $\vec{K}$. For this step, we use the publicly available Nice2Predict framework [5]. Nice2Predict computes the MAP query using a scalable, greedy algorithm, where names assigned to obfuscated program elements are iteratively changed one-by-one or in pairs until the score stops improving. At every iteration, all naming constraints are checked for violations. More details on this algorithm are provided in [31]. After predicting the names for all obfuscated elements, DeGuard renames them using the Soot API, and then constructs and outputs the deobfuscated APK.

## 6.2 Experimental Evaluation

We now present our experiments with DeGuard. First, we evaluate DeGuard's accuracy on deobfuscating benign, open-source applications obfuscated using ProGuard. Second, we discuss our experience in inspecting malware samples deobfuscated using DeGuard.

### 6.2.1 ProGuard Experiments

We perform two tasks to evaluate DeGuard's performance on ProGuard-obfuscated applications. First, we measure DeGuard's accuracy on predicting the names of program elements obfuscated by ProGuard. Second, based on the results of the first task, we report DeGuard's accuracy on the task of predicting the names of obfuscated third-party libraries imported in the APK.

To conduct the above tasks, we obfuscated 100 benign applications from F-Droid. These are the 100 applications that we intentionally did not use during the learning phase. For all 100 applications, we enabled ProGuard obfuscation by modifying their build files, without modifying ProGuard's obfuscation rules, which specify which elements are obfuscated. In our experiments, we use the non-obfuscated versions of the applications as an oracle to check whether DeGuard correctly deobfuscates the program elements' names by renaming them to their original (i.e., non-obfuscated) names.

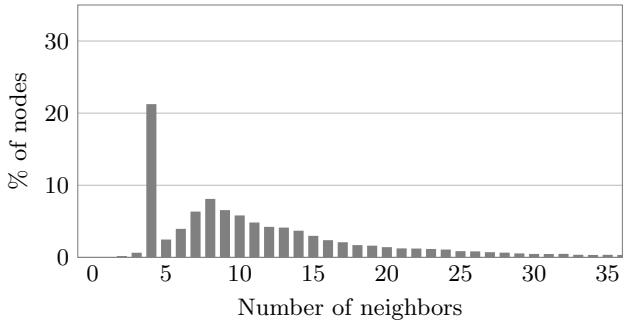



Figure 4: Distribution of total number of neighbors over the 100 ProGuard-obfuscated Android applications.

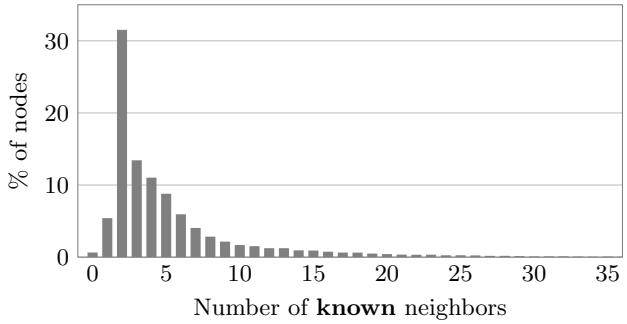



Figure 5: Distribution of known number of neighbors over the 100 ProGuard-obfuscated Android applications.

### *ProGuard-obfuscated APKs.*

In Figures 4 and 5 we show two relevant metrics that reveal the dependency structure of the 100 applications that we obfuscated using ProGuard. The bar chart depicted in Figure 4 shows the distribution of total number of neighbors. This figure shows one bar for each neighborhood size, where the bar's height indicates the percentage of nodes that have exactly that number of neighbors. For example, the fifth bar indicates that the percentage of nodes with exactly 4 neighbors is around 22%. Similarly, the bar chart shown in Figure 5 shows the distribution of *known* neighbors. The data in these two figures reveals two key points about our features: (i) the nodes are well-connected (99% of the nodes have at least 3 neighbors), and (ii) most nodes have known neighbors (99% have at least one known neighbor). That is, our features lead to dependency graphs where informed prediction seems possible (rather than graphs which are mostly disconnected where there would be little or no flow into a node whose name is to be predicted).

### *Task 1: Predicting Program Element Names.*

For this task, we deobfuscated the 100 benchmark applications, which we previously obfuscated with ProGuard. We remark that ProGuard, in addition to renaming program elements, also removes some elements. For example, it removes fields, methods, and classes that are not used by the application. Hence, we evaluate whether DeGuard correctly deobfuscates elements *not* removed by ProGuard.

Figure 6 shows the percentage of known elements (which DeGuard does not try to reverse), correctly predicted elements, and mis-predicted elements, averaged over all 100 applications deobfuscated by DeGuard. Each bar has three segments, which represent the three kinds of program elements: (i) *known*, which are the elements not obfuscated by ProGuard and which the system keeps as is, (ii) *correct*, which are the elements that DeGuard correctly renames to their original names, and (iii) *mis-predicted*, which are the elements for which DeGuard predicts names that differ from the original ones. Here, the first four bars show data about fields, methods, classes, and packages, respectively, and the fifth bar shows the aggregate data for all program elements. We use the predictions made for the package names in the second task discussed in this section.

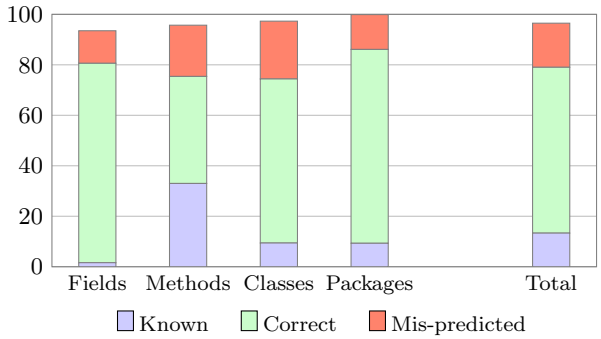



Figure 6: Average percentage of known, correctly predicted, and mis-predicted program elements calculated over the 100 Android applications deobfuscated by DeGuard.

The data shows that ProGuard obfuscates a substantial number of the program elements. On average, only 1.6% of the fields, 33% of the methods, 9.4% of the classes, and 9.3% of the packages are known. Thus, on average, ProGuard obfuscates 86.7% of each application's program elements.

The data further shows that DeGuard correctly deobfuscates a significant part of the obfuscated program elements. For example, while only 1.6% of the fields in the obfuscated applications are known, after DeGuard deobfuscates them, 80.6% of all fields have names identical to the original ones. We remark that 80.6% is a lower bound on how well DeGuard deobfuscates fields. This is because some of the names classified as mis-predicted are semantically close to the original ones. For example, in the application `FacebookNotifications`, DeGuard suggested `appView` and `mWindowManager` as names for two fields, while the original names are `webview` and `windowManager`, respectively.

Overall, the data shows that among all program elements, DeGuard increases the percentage of names that are identical to the original ones from 13.3% (in the obfuscated APK) to 79.1% (in the deobfuscated APK). We remark that the applications used in this experiment are benign. DeGuard's prediction accuracy on malicious applications may therefore be lower.

### *Task 2: Predicting Third-party Libraries.*

We next use the deobfuscation results for package names obtained from Task 1 in order to evaluate DeGuard's effectiveness for predicting third-party libraries.

We first explain what we mean by the term library. We identify libraries by their package names. We classified package names into *library* and *application-specific* using a simple heuristic: any package name that appears in multiple applications is classified as corresponding to a library. This heuristic works well because most application-specific pack-

```
public final class d {
  private String a =
    System.getProperty("line.separator");
  private char[] b;
  private byte[] c;

  public static byte[] a(String str) {...};
}
```

(a) Obfuscated code

```
public final class Base64 {
  private String NL =
    System.getProperty("line.separator");
  private char[] ENC;
  private byte[] DEC;

  public static byte[] decode(String str) {...};
}
```

(b) Deobfuscated code

Figure 7: Deobfuscating the Base64 decoder found in the GingerMaster malware sample.

age names are unique to a particular application. For example, `org.apache.commons.collections4` is classified as a library, while `com.pindroid.providers` is classified as an application-specific package name. Based on our heuristic, we identified a total of 133 libraries imported in the APKs.

To measure DeGuard's effectiveness in predicting libraries we use two (standard) metrics — precision and recall. Let $L$ denote the set of all obfuscated libraries imported by an application and $P$ denote the set of predicted libraries by DeGuard. Here, $P$ contains all package names that map to one of the 133 names that we have previously classified as libraries. We compute DeGuard's *precision* using the formula $precision = |L \cap P|/|P|$, and *recall* using the formula $recall = |L \cap P|/|L|$. Intuitively, precision shows the percentage of libraries that DeGuard correctly predicts, and recall captures the percentage of libraries that DeGuard attempts to predict.

DeGuard's precision and recall for predicting libraries is, on average, 91.3% and 91.0%, respectively. This result indicates that DeGuard predicts libraries more accurately compared to arbitrary program elements. Further, DeGuard almost never mis-predicts a library. This is likely because the training set, which we use to learn the weights of all features, may contain applications that import the same libraries that we attempt to predict.

We remark that the benchmark applications in this experiment are not malicious, and so the libraries that they import are also benign. Malicious third-party libraries embedded in applications can be more difficult to identify.

### *Prediction Speed.*

For most applications, DeGuard takes on average less than a minute to deobfuscate its APK. Around 10% of the time is spent in constructing the dependency graph and deriving all syntactic and semantics constraints. The remaining 90% of the time is spent in computing the most likely naming assignment using the approximate MAP inference query. An interesting future work item is to investigate faster MAP inference algorithms leveraging the specifics of our dependency graphs.

```
public class SearchOfficesView extends BaseView {
  private void g() {
    m = getSystemService("location");
    local = m.getBestProvider(...);
    o = m.getLastKnownLocation(local);
    ...
  }

  private void j() {...}
}
```

(a) Obfuscated code

```
public class SearchOfficesView extends BaseView {
  private void init() {
    locationManager=getSystemService("location");
    local = locationManager.getBestProvider(...);
    location =
      locationManager.getLastKnownLocation(local);
    ...
  }

  private void requestLocationUpdates() {...}
}
```

(b) Deobfuscated code

Figure 8: Deobfuscating fields and methods that store and, respectively, handle location data. The code snippet is taken from the Bgserv malware sample.

### *Summary of ProGuard Experiments.*

In summary, our experiments demonstrate that: (i) DeGuard correctly predicts an overwhelming part of the program elements obfuscated by ProGuard, thereby effectively *reversing* ProGuard's obfuscation mechanism; (ii) it precisely identifies third-party Android libraries, and (iii) it is robust and efficient, taking on average less than a minute per application.

### 6.2.2 *Experiments with Malware Samples*

We randomly selected one sample from each of the 49 malware families reported in [40]. We used DeGuard to deobfuscate the selected samples and manually inspected some of them. While we cannot report DeGuard's exact precision on the selected samples (since they are all obfuscated), we report on several interesting examples that suggest that DeGuard can be useful when inspecting malware. We also discuss DeGuard's limitations in the context of malware.

### *Revealing Base64 String Decoders.*

Malware often disguises text strings using the Base64 encoding scheme. DeGuard can be used to discover the classes that implement such standard encoding schemes. Concretely, DeGuard discovered the Base64 decoders in three of the malware samples that we inspected. We remark that eight other samples also have Base64 decoders, but the classes that implement this encoding are not obfuscated.

As an example, in Figure 7 we show code snippets taken from the GingerMaster [39] malware sample[2]. Figure 7(a) shows the obfuscated code, and Figure 7(b) the corresponding deobfuscated code obtained using DeGuard. DeGuard discovers that the class `d` implements a Base64 decoder and

[2]SHA1: `2e9b8a7a149fcb6bd2367ac36e98a904d4c5e482`

renames it to `Base64`. Further, DeGuard reveals that the method `a(String)` decodes strings formatted in Base64 (we confirmed this by inspecting the implementation of method `a(String)`) and renames it to `decode(String)`. The statement `Base64.decode(String)` is more descriptive compared to `d.a(String)`, and we thus believe that DeGuard can help security analysts in inspecting this malware sample.

### *Revealing Sensitive Data Usage.*

Malware often steals personal information, such as location data, device identifiers, and phone numbers. We search the deobfuscated code of our malware samples for identifiers such as `location` and `deviceId` and discovered that DeGuard often deobfuscates the names of fields that store sensitive data and methods that handle sensitive data. As an example, in Figure 8 we show an obfuscated code snippet taken from the Bgserv malware sample[3], along with the deobfuscated code output by DeGuard. We observe that DeGuard renames the obfuscated field `o`, which stores the device's location, to `location`. Further, it renames the method `j()` to `requestLocationUpdates()`. We inspected the method `j()` to reveal that it instantiates the interface `LocationListener` and implements the method `onLocationChanged()` to receive location updates. The name `requestLocationUpdates()` assigned by DeGuard captures the behavior of this method.

In the remaining malware samples, we discovered that DeGuard renames a number of fields and methods that handle other kinds of sensitive data, such as device identifiers. We believe that DeGuard can help in inspecting malware that misuses sensitive data, e.g., by allowing security experts to search for certain identifiers in the deobfuscated code.

### *Limitations.*

In addition to obfuscating program identifiers, most of malware samples we inspected are obfuscated with additional techniques to further hinder reverse engineering. Examples include custom encoding of strings (e.g. using custom encryption), extensive use of reflection, control flow obfuscation, code reordering using **`goto`** statements, and others. Reversing these additional obfuscation steps is beyond the scope of DeGuard. To inspect malware, security analysts must therefore use a combination of deobfuscation tools tailored to reversing different obfuscation techniques.

## 7. RELATED WORK

This section summarizes the works that are most closely related to ours.

### *Suggesting names for program elements.*

Several works have studied the effect of identifier names [36, 33, 12] and have shown that good names have significant impact on one's ability to understand the source code. These studies have inspired tools [13, 33] that rename identifiers within a project to make them follow a given coding convention. In contrast to DeGuard, the systems presented in [13] and [33] cannot be used for deobfuscation. The tool described in [13] relies on the names to be replaced to have meaningful, non-obfuscated names, so it can improve them. This system cannot predict meaningful replacements of names such as "a". The tool of [33] does not suggest new names: it only identifies bad names based on syntactic guidelines and provides that feedback to developers.

The works of Allamanis et al.[8, 9] suggest names for Java program elements using n-gram language models and neural networks. Their technique, however, only allows predicting the name of a single program element and is thus not applicable to a deobfuscation task where most names are missing.

LibRadar [26] detects third-party Android libraries by extracting a unique fingerprint from each library and creating a mapping from fingerprints to library names. Obfuscated libraries are then identified by their fingerprints. In contrast to DeGuard, LibRadar is less general as it predicts names only for packages, and it is less robust because it relies on stable features (i.e., completely unaffected by obfuscation).

### *Probabilistic models for programs.*

A recent surge in the number of open-source repositories has triggered several authors to create large-scale probabilistic models for code. These models are then used for novel applications such as code completion [32], generating code from natural language [17, 10], sampling code snippets [27], programming language translation [18], type annotating programs [19, 31] and others.

Closest to our work is [31] which also uses structured prediction and the Nice2Predict framework [5] to guess names of local variables for JavaScript programs. Our setting however is different and requires more diverse feature functions, constraints and range of elements for which names are to be predicted; further, we use an order of magnitude more scalable learning mechanism than [31].

Several works [22, 21, 16, 24] use graphical models to discover properties about programs such as function specifications and invariants. These works, however, do not use MAP inference to discover overall optimal solutions for all predicted properties (and most do not learn from existing programs). The work of Shin et al. [34] uses neural networks and a large training set to identify libraries in binaries. In the context of Android, a recent paper by Octeau et al. [28] uses a probabilistic model and static analysis to determine if two applications may communicate via the Android intent mechanisms. However, theirs is a rather different task than the one addressed by our work.

## 8. CONCLUSION

We presented a new approach for layout deobfuscation of Android APKs. The key idea is to phrase the problem of reversing obfuscated names as structured prediction in a probabilistic graphical model and to leverage the vast availability of non-obfuscated Android programs to learn this model. We implemented our approach in a system called DeGuard and demonstrated that DeGuard can successfully and with high precision reverse obfuscations performed by ProGuard, a task beyond the reach of existing systems. We believe that our work indicates the promise of leveraging probabilistic models over "Big Code" for addressing important challenges in security.

## 9. ACKNOWLEDGMENTS

The research in this work has been partially supported by ERC starting grant #680358. We thank Matteo Panzacchi for extending Nice2Predict [5] with support for pseudo likelihood estimation.

[3]SHA1: `03f9fc8769422f66c59922319bffad46d0ceea94`

## 10. REFERENCES


[1] Advertising SDK Can Be Hijacked for Making Phone Calls, Geo-Locating. http://www.hotforsecurity.com/blog/advertising-sdk-can-be-hijacked-for-making-phone-calls-geo-locating-7461.html.

[2] dex2jar. https://github.com/pxb1988/dex2jar.

[3] F-Droid. https://f-droid.org/.

[4] Java Decompiler. http://jd.benow.ca/.

[5] Nice2Predict. https://github.com/eth-srl/Nice2Predict.

[6] ProGuard. http://proguard.sourceforge.net/.

[7] Type Erasure. https://docs.oracle.com/javase/tutorial/java/generics/genTypes.html.

[8] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *FSE*, 2014.

[9] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *FSE*, 2015.

[10] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In *ICML*, 2015.

[11] 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 *PLDI*, 2014.

[12] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *CSMR*, 2010.

[13] B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM*, 2000.

[14] K. Chen, X. Wang, Y. Chen, P. Wang, Y. Lee, X. Wang, B. Ma, A. Wang, Y. Zhang, and W. Zou. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *S&P*, 2016.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.

[16] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *POPL*, 2007.

[17] T. Gvero and V. Kuncak. Synthesizing java expressions from free-form queries. In *OOPSLA*, 2015.

[18] S. Karaivanov, V. Raychev, and M. Vechev. Phrase-based statistical translation of programming languages. Onward!, 2014.

[19] O. Katz, R. El-Yaniv, and E. Yahav. Estimating types in binaries using predictive modeling. In *POPL*, 2016.

[20] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[21] T. Kremenek, A. Y. Ng, and D. Engler. A factor graph model for software bug finding. In *IJCAI*, 2007.

[22] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: Inferring the specification within. In *OSDI*, 2006.

[23] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *ICML*, 2001.

[24] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *PLDI*, 2009.

[25] D. Low. Protecting Java Code via Code Obfuscation. *Crossroads*, 4(3), Apr. 1998.

[26] Z. Ma, H. Wang, Y. Guo, and X. Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *ICSE 2016 - Companion Volume*, 2016.

[27] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. In *ICML*, 2014.

[28] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *POPL*, 2016.

[29] N. D. Ratliff, J. A. Bagnell, and M. Zinkevich. (Approximate) Subgradient Methods for Structured Prediction. In *AISTATS*, 2007.

[30] V. Raychev, P. Bielik, M. Vechev, and A. Krause. Learning programs from noisy data. In *POPL*, 2016.

[31] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *POPL*, 2015.

[32] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *PLDI*, 2014.

[33] P. A. Relf. Tool assisted identifier naming for improved software readability: an empirical study. In *ISESE*, 2005.

[34] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security*, 2015.

[35] C. Sutton and A. McCallum. An introduction to conditional random fields. *Found. Trends Mach. Learn.*, 4(4):267–373, Apr. 2012.

[36] A. A. Takang, P. A. Grubb, and R. D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

[37] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. Aletheia: Improving the usability of static security analysis. In *CCS*, 2014.

[38] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1999.

[39] R. Yu. Ginmaster: A case study in android malware. https://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/Yu-VB2013.pdf.

[40] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *S&P*, 2012.