Deobfuscating Android Applications through Deep Learning

Fang-Hsiang Su
Columbia University
Email: mikefhsu@cs.columbia.edu

Jonathan Bell
George Mason University
Email: bellj@gmu.edu

Gail Kaiser
Columbia University
Email: kaiser@cs.columbia.edu

Baishakhi Ray
University of Virginia
Email: br8jr@virginia.edu

***Abstract*—Android applications are nearly always obfuscated before release, making it difficult to analyze them for malware presence or intellectual property violations. Obfuscators might hide the true intent of code by renaming variables, modifying the control flow of methods, or inserting additional code. Prior approaches toward automated deobfuscation of Android applications have relied on certain structural parts of apps remaining as landmarks, un-touched by obfuscation. For instance, some prior approaches have assumed that the structural relationships between identifiers (e.g. that A represents a class, and B represents a field declared directly in A) are not broken by obfuscators; others have assumed that control flow graphs maintain their structure (e.g. that no new basic blocks are added). Both approaches can be easily defeated by a motivated obfuscator. We present a new approach to deobfuscating Android apps that leverages deep learning and topic modeling on machine code, MACNETO. MACNETO makes few assumptions about the kinds of modifications that an obfuscator might perform, and we show that it has high precision when applied to two different state-of-the-art obfuscators: ProGuard and Allatori.**

I. INTRODUCTION

Android apps are typically obfuscated before delivery, in an effort to decrease the size of distributed binaries and reduce disallowed reuse. In some cases, malware authors take advantage of the general expectation that Android code is obfuscated to pass off obfuscated malware as regular code: obfuscation will hide the actual purpose of the malicious code, and the fact that there is obfuscation will not be surprising, as it is already a general practice. Hence, there is great interest in automated *deobfuscators*: tools that can automatically find the original structure of code that has been obfuscated.

Deobfuscators can be used as a part of various automated analyses, for instance, plagiarism detection or detecting precise versions of third party libraries that are embedded in apps, allowing auditors to quickly identify the use of vulnerable libraries. Similarly, deobfuscators can be used to perform code search tasks among obfuscated apps using recovered identifiers and simplified control flow. Deobfuscators can also be used as part of a human-guided analysis, where an engineer inspects applications to determine security risks.

In general, deobfuscators rely on some training set of non-obfuscated code to build a model to apply to obfuscated code. Once trained, deobfuscators can recover the original names of methods and variables, or even the original structure and code of methods that have been obfuscated. For example, some deobfuscation tools rely on the structure of an app's control flow graph. However, they are susceptible to obfuscators that introduce extra basic blocks and jumps to the app's code and can be slow to use, requiring many pair-wise comparisons to perform their task [1], [2]. Using another approach, DeGuard [3] is a state-of-the-art deobfuscator that builds a probabilistic model for identifiers based on the co-occurrence of names (e.g., knowing that some identifier a is a field of class b which is used by method c). While this technique can be very fast to apply (after the statistical model is trained), this approach is defeated by obfuscators that change the layout of code (e.g. move methods to new classes or introduce new fields).

We present a novel approach for automated deobfuscation of Android apps: MACNETO, which applies recurrent neural networks and deep learning to the task. MACNETO leverages a key observation about obfuscation: an obfuscator's goal is to transform how a program looks as radically as possible, while still maintaining the original program semantics. MACNETO deobfuscates code by learning the deep semantics of what code does through topic modeling. These topic models are a proxy for program behaviors that are stable despite changes to the layout of code, the structure of its control flow graph, or any metadata about the app (features used by other deobfuscators). These topic models are trained using a relatively simple feature set: a language consisting of roughly 20 terms that represent the different low-level bytecode instructions in Android apps and roughly 200 terms that represent the various Android APIs. MACNETO's topic model is resilient to many forms of obfuscation, including identifier renaming (as employed by ProGuard [4]), method call injection, method splitting, and other control flow modifications (as employed by Allatori [5]).

MACNETO uses deep learning to train a topic classifier on known obfuscated and un-obfuscated apps offline. This training process allows MACNETO to be applicable to various obfuscators: supporting a new obfuscator would only require a new data set of obfuscated and deobfuscated apps. Then, these models are saved for fast, online deobfuscation where obfuscated code is classified according to these topics, and matched to its original code (which MACNETO hadn't been trained to recognize). This search-oriented model allows MACNETO to precisely match obfuscated code to its deobfuscated counterpart. This model is very applicable to many malware-related deobfuscation tasks, where a security researcher has various malware samples and is trying to identify if those samples had been hidden in an app. Similarly, it is immediately

applicable to plagiarism-related deobfuscation tasks, were an analyst has the deobfuscated version of their code and is searching for obfuscated versions of it.

We evaluated MACNETO by building several deobfuscation models based on over $1,500$ real android apps using two popular, advanced obfuscators: ProGuard [4] and Allatori [5]. Compared to a state-of-the-art statistical approach [3], MACNETO had significantly higher precision at recovering method names obfuscated by ProGuard 96% vs 67%.

On Allatori (which employs significantly more complex obfuscations), MACNETO maintained a good precision (91%), while the state-of-the-art approach could not be applied at all. Moreover, we found that MACNETO performs well even with a relatively small training set. Based on these findings, we believe that MACNETO can be very successful at deobfuscating method names.

The contributions of this paper are:

- A new approach to deobfuscation leveraging deep learning and machine topics.
- A new approach to automatic classification of programs with similar semantics.
- An evaluation of our tool on two popular obfuscators.
- An open source implementation of our approach, MACNETO, released via an MIT license on GitHub [1].

II. BACKGROUND

In general, obfuscators make transformations to code that result in an equivalent execution, despite structural or lexical changes to the code — generating code that looks different, but behaves similarly. Depending on the intended purpose (e.g. hiding a company's intellectual property, disguising malware, or minimizing code size), a developer may choose to use a different kind of obfuscator. At a high level, these obfuscations might include lexical transformations, control transformations, and data transformations [6]. Obfuscators might choose to apply a single sort of transformation, or several.

Lexical transformations are typically employed by "minimizing" obfuscators (those that aim to reduce the total size of code for distribution). Lexical transformations replace identifiers (such as method, class or variable names) with new identifiers. Since obfuscators are applied only to individual apps, they must leave identifiers exposed via public APIs unchanged. Similarly, if some obfuscated class C_1 extends some non-obfuscated class C_2 and overrides method m, then the obfuscator can't change the name of m without breaking inheritance structures.

Control transformations can be significantly more complex, perhaps inlining code from several methods into one, splitting methods into several, reordering statements, adding jumps and other instructions [7], [8]. Control transformations typically leverage the limitations of static analysis: an obfuscator might add additional code to a method, with a jump to cause the new code to be ignored at runtime. However, that jump might be based on some complex heap-stored state which is tricky for a static analysis tool to reason about.

Finally, data transformations might involve encoding data in an app or changing the kind of structure that it's stored in. For instance, an obfuscator might encrypt strings in apps so that they can't be trivially matched, or change data structures (e.g. in Java from an `array` to an `ArrayList`) [7].

In this paper we define the deobfuscation problem as follows. A developer/security analyst has access to a set of original methods and their corresponding obfuscated versions, and her job is to identify the corresponding original version given obfuscated program. Then the developer/analyst can analyze the original program to identify malware variants which becomes a significantly easier problem. Thus, in our case, deobfuscation essentially becomes a search problem, similar to DeGuard [3].

We assume that obfuscators can make lexical, control, and data transformations to code. We do not base our deobfuscation model on any lexical features, nor do we base it on the control flow structure of or string/numerical constants in the code. When inserting additional instructions and methods, we assume that obfuscators have a limited vocabulary of no-op code segments to insert. That is, we assume that there is some pattern (which need not be pre-defined) that our deep learning approach can detect. MACNETO relies on a training phase that teaches it the rules that the obfuscator follows: if the obfuscator is truly random (with no pattern to the sort of transformations that it makes), then MACNETO would be unable to apply its trained deobfuscation model to other obfuscated apps. However, we imagine that this is a reasonable model: an adversary would have to spend an incredible amount of resources to construct a truly random obfuscator.

Since it relies on static analysis, MACNETO could also be defeated by an obfuscator that inserts many reflective method calls (which are dynamically resolved at runtime). This obfuscator could effectively mask all of the 250 features that MACNETO uses to classify methods to topic vectors. In that case, MACNETO would be relying only on the bytecode instructions. MACNETO could be adapted to better analyze reflection through existing techniques [9].

III. MACNETO OVERVIEW

From a set of original and obfuscated methods, MACNETO intends to identify the original version of a given obfuscated method. Here we describe an overview of MACNETO.

Although obfuscators may perform significant structural and/or naming transformations, the semantics of a program before and after obfuscation remain the same. MACNETO leverages such semantic equivalence between an original program executable and its obfuscated version at the granularity of individual methods. The semantics of a method are captured as the hidden topics of its machine code ("machine topics") instead of human texts such as identifier names in methods. By construction, an obfuscated method is semantically equivalent to its original, non-obfuscated method that it is based on. MACNETO assumes that the machine topics of an obfuscated

[1] MACNETO will not be public during double-blind review.

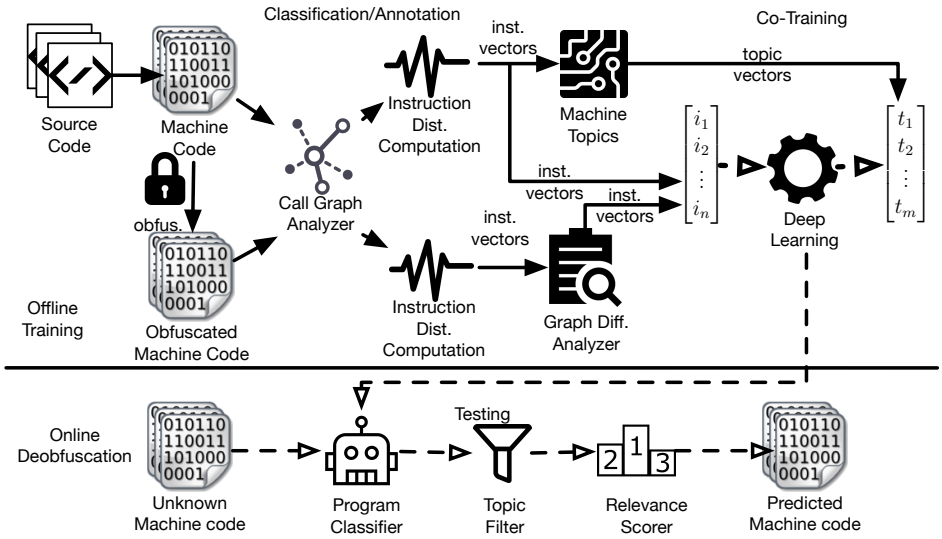


Fig. 1: The system architecture of MACNETO, which consists of four stages: instruction distribution, classification, deep-learning on machine topics and online scoring, to deobfuscate Android apps.

method will match those of the original method. In its learning phase, MACNETO is provided a training set of methods, which are labeled pairs of obfuscated and non-obfuscated methods. Once training is complete, MACNETO can be presented with an arbitrary number of obfuscated and deobfuscated methods, and (assuming that the deobfuscated method exists in the input set) accurately match each obfuscated method with its original version. In the event that the original method body isn't known, MACNETO can return a suggested method (that it had been trained on) which was very similar to the original method, and, in our evaluation, typically has the same name.

MACNETO utilizes a four stage approach:

(i) *Computing Instruction Distribution.* For an application binary (original or obfuscated), MACNETO parses each method as a distribution of instructions, which is analogous to the term frequency vector for a document. Instead of considering just the instructions of each method, MACNETO also recursively considers the instructions of each method's callee(s), which helps MACNETO to deeply comprehend behavioral semantics.

(ii) *Machine Topic Modeling.* Identifies machine topics from the instruction distribution of the original method. These machine topics are used as a proxy for method semantics. The same topic model is used later to annotate the corresponding obfuscated method as well.

(iii) *Learning.* Uses a two-layered Recurrent Neural Network (RNN) where the input is the instruction distribution of a method (original and obfuscated), and the output layer is the corresponding machine topic distribution of the original method. MACNETO uses this two-layered RNN as a program classifier that maps an original method and its obfuscated version to the same class and represented by machine topic vector . This is the training phase of the RNN model. Such model can be pre-trained as well.

(iv) *Deobfuscating.* This is basically the testing phase of the RNN model. It operates on a set of original and obfuscated methods that form our testing set. Given an obfuscated method, the above RNN model tries to infer its topic distribution; MACNETO in turn tries to find a set of original methods with similar topic distribution and ranks them as possible deobfuscated methods.

Figure 1 shows a high level overview of MACNETO's approach for deobfuscation. The first three stages occur offline and can be pre-trained.

Consider the example `readAndSort` program shown in Figure 2, assuming that this is a method belonging to an Android app that we are using to train MACNETO. To compute the instruction distribution of `readAndSort`, MACNETO first calculates the callgraph and identify its two callees `read` and `sort`. The instruction distributions of these two callee methods will be inlined into `readAndSort`. Then MACNETO moves to the next step, applying topic modeling on the instruction distributions of all methods including `readAndSort` in the training app set. The result of this step is a vector containing the probability/membership that a method has toward each topic: a Machine Topic Vector (MTV). MACNETO annotates both the original and obfuscated versions of this method with this same MTV. This annotation process allows our learning phase to predict similar MTVs for a method and its obfuscated version, even when their instruction distributions are different.

IV. MACNETO APPROACH

This section describes the four stages of MACNETO in detail, illustrating our several design decisions. We have designed MACNETO to target any JVM-compatible language (such as Java), and evaluate it on Android apps. MACNETO works at the level of Java bytecode; in principle, its approach could be applied to other compiled languages as well. In this paper, executable/binary actually means Java bytecode executable and machine code means Java bytecode.

A. Computing Instruction Distribution

We use the instruction distribution of a method to begin to approximate its behavior. This involves two main steps: 1) for a target method m (original or obfuscated), MACNETO recursively identifies all methods that it invokes (its callees) and then cumulatively computes the instruction distribution of m and its callees using term frequency distribution, 2) MACNETO identifies the differences in callees between an original and obfuscated method using a *graph diff* algorithm, and optionally filters out the additional callees from obfuscated methods. Here, we will explain each of these steps.

*1) **Instruction Distribution (ID)**:* ID is a vector that represents the frequencies of important instructions. For a method M_j, its instruction distribution can be represented as $ID_{M_j} = [freq_j^{I_1}, freq_j^{I_2}, ...freq_j^{I_a}]$, where a represents the index of an instruction and $freq_j^{I_a}$ represents the frequency of the a_{th} instruction in the method M_j. This step is similar to building the term frequency vector for a document.

MACNETO considers various bytecode operations as individual instructions (e.g. loading or storing variables), as well as a variety of APIs provided by the Android framework. Android API calls provide much higher level functionality than simple bytecode operators, and hence, including them as "instructions" in this step allows MACNETO to capture both high and low level semantics. However, including too many different instructions when calculating the distribution could make it difficult to relate near-miss semantic similarity.

To avoid over-specialization, MACNETO groups very similar instructions together and represents them with a single word. For instance, we consider all instructions for adding two values to be equivalent by abstracting away the difference between the instruction `fadd` for adding floating point numbers and the instruction `iadd` for adding integers. All instructions for adding different data types are categorized as a single one `xadd`. Table I lists all of the instructions MACNETO considers.

Further, for a target method under analysis, MACNETO inlines the instructions from the target's callee method(s) recursively in the instruction distribution to capture the target's context. For that, MACNETO constructs callgraphs for applications and libraries using FlowDroid [10], a state-of-the-art tool that uses context-, flow-, field-, and object-sensitive android lifecycle-aware control and data-flow analysis [10]. For example, consider the method `readAndSort` as shown in Figure 2. `readAndSort` simply reads the first line of a file as a string and then sorts this string. It delegates its functionality to two subroutines, `readFile` and `sort`. Both `readFile` and `sort` also invoke several methods, such as `toCharArray` and `readLine` APIs included in Android to help them complete their tasks. The corresponding call graph is shown in Figure 2a.

MACNETO considers following two classes of callee methods: (i) *Android APIs.* These methods are offered by the Android framework directly. MACNETO models these APIs as single instructions. `readLine` and `toCharArray` in Figure 2 belong to this category. (ii) *Application methods.*

TABLE I: MACNETO'S INSTRUCTION SET.

Opcode	Description
`xaload`	Load a primitive or an object from an array, where `x` represents a type of primitive or object.
`xastore`	Store a primitive or an object to an array, where `x` represents a type of primitive or object.
`arraylength`	Retrieve the length of an array.
`xadd`	Add two primitives on the stack, where `x` represents a type of primitive.
`xsub`	Subtract two primitives on the stack, where `x` represents a type of primitive.
`xmul`	Multiply two primitives on the stack, where `x` represents a type of primitive.
`xdiv`	Divide two primitives on the stack, where `x` represents a type of primitive.
`xrem`	Compute the remainder of two primitives on the stack, where `x` represents a type of primitive.
`xneg`	Negate a primitive on the stack, where `x` represents a type of primitive.
`xshift`	Shift a primitive on the stack, where `x` represents integer or long.
`xand`	Bitwise-and two primitives on the stack, where `x` represents integer or long.
`xor`	Bitwise-or two primitives on the stack, where `x` represents integer or long.
`x_xor`	Bitwise-xor two primitives on the stack, where the first `x` represents integer or long.
`iinc`	Increment an integer on the stack.
`xcomp`	Compare two primitives on the stack, where `x` represents a type of primitive.
`ifXXX`	Represent all `if` instructions. Jump by comparing value(s) on the stack.
`xswitch`	Jump to a branch based on the index on the stack, where `x` represents `table` or `lookup`.
`android_apis`	The APIs offered by the Android framework, which usually starts from `android.`, `dalvik.`, `java.`. MACNETO records 235 android apis.

These are all the other methods beside Android APIs. These are methods from an application or third party libraries used by an application. While MACNETO treats Android APIs as individual instructions, all other application method calls are inlined into the calling method, resulting in the instruction distributions of those callee methods being merged directly in the target method. We can then define the instruction distribution of a target method M_j as:

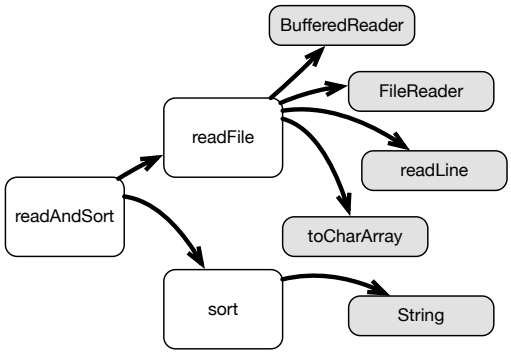
$$ID_{M_j} = ID_{M_j} + \sum_{M_k \in callees(M_j)} ID_{M_k} \quad (1)$$

, where j is the index of the current method and k represents the indices of its callee methods. We use these instruction distributions as the input source for next step to identify topics embedded in programs.

To calculate these instruction distributions, MACNETO uses the ASM Java bytecode library [11], and Dex2Jar [12]. This allows MACNETO to deobfuscate Android apps (which are distributed as APKs containing Dex bytecode), while only needing to directly support Java bytecode. For collecting Android APIs, we analyze the core libraries from Android API level 7 to 25 [13].

*2) **Graph Diff**:* Some obfuscators may inject additional instructions into the original methods, which might then change instruction distribution and hence the perceived semantics of those methods. Since MACNETO inlines application method

```
public String readAndSort(String f) {
  char[] data = readFile(f);
  return sort(data);
}
```



(a) The `readAndSort` method and its callgraph.

```
public static char[] readFile(String f) {
  try {
    BufferedReader br =
      new BufferedReader(new FileReader(f));
    String first = br.readLine();
    return first.toCharArray();
  } catch (Exception ex) {
  }
  return null;
}
public static String sort(char[] data) {
  for (int i = 1; i < data.length; i++) {
    int j = i;
    while (j > 0 && data[j - 1] > data[j]) {
      char tmp = data[j];
      data[j] = data[j - 1];
      data[j - 1] = tmp;
      j = j - 1;
    }
  }
  return new String(data);
}
```

(b) The callee methods of `readAndSort`.

Fig. 2: The `readAndSort` method and its callgraph (a). Two callee methods, `readFile` and `sort`, of `readAndSort` (b).

calls, inserting additional method calls could cause significant distortions to the instruction distribution of a method. MACNETO combats this obfuscation by learning (and then filtering out) superfluous method calls that are systematically inserted by obfuscators from the callgraphs before/after obfuscation of the original method. We call this learning module on callgraphs as *graph diff*.

For each method in an Android app, MACNETO first analyzes the callee difference of the current method M_j before and after the obfuscation

$$\Delta M_j = callees(M_j^{obfus}) - callees(M_j) \tag{2}$$

, where $callees(M_j)$ and $callees(M_j^{obfus})$ represent the callees of M_j before and after the obfuscation, respectively. If ΔM_j is not empty, MACNETO records the instruction distributions of the methods in ΔM_j as patterns. In a training app set, MACNETO computes the appearances of each pattern (instruction distribution) it learns from the callee differences and reports those patterns having high support. We first define the total callee differences detected by MACNETO as

$$Diff_{callee}(T) = \bigcup_k \Delta M_k \tag{3}$$

, where T represents a training app set and k is the method index. Then we define the support of a pattern as

$$Support(Pattern_i) = \frac{Count(Pattern_i)}{\sum_j Count(Pattern_j)} \tag{4}$$

, where i is pattern index, $Pattern_i \in Diff_{callee}(T)$ and $Count(.)$ returns the appearance number of a pattern.

We again use `readAndSort` example in Figure 2 to demonstrate how graph diff works. Let's assume an obfuscator injects a method `decrypt(String)` to method `readAndSort`; the callees of obfuscated `readAndSort` becomes $callee(readAndSort^{obfus}) = \{\texttt{read}, \texttt{sort}, \texttt{decrypt}\}$. Thus, the $\Delta readAndSort = \{\texttt{decrypt}\}$. The instruction distribution of `decrypt`, $ID_{decrypt}$, is recorded by MACNETO as a pattern. If the support of such pattern is higher than a threshold (0.03 in this paper), MACNETO reports it as a significant pattern and uses it to filter out superfluous method calls in the testing app set.

Note that the Graph Diff step is optional, but can have a positive impact to recover methods obfuscated by an advanced obfuscator. We evaluated the impact of including the Graph Diff step in detail in Section V.

B. Machine Topic Modeling

Topic modeling [14] is a generative model that identifies the probabilities/memberships that a document has to (hidden) topics or groups. Topic modeling has been widely used in software engineering literature to understand programs and detect anomalies [15], [16]. Most existing approaches apply topic modeling on human-readable program elements (*e.g.,* method names, comments or texts in source code) or program documents to identify hidden topics. These approaches treat each program as plain text and use topic modeling to identify topics in programs, as labeled by human words. However, these human words may be noisy and inadequate to describe program behavior. Further, an obfuscated or anonymized program may not have a meaningful method name or other identifiers, and would not have any comments left. Existing approaches reliant on human words to cluster or classify programs by topics may fail to process such programs.

In contrast, in this paper, we propose the concept of *Machine Topics*, where we attempt to identify the probability distribution that a method belongs to multiple topics from *machine code*. We further include machine code of the callee methods to retrieve the contextual semantics as well. Modeling instructions (machine code) as terms and methods as documents, MACNETO uses Latent Dirichlet Allocation [14] to extract hidden machine topics in methods. We extract topics from the original method since some noisy instructions such as `nop` might be injected into an obfuscated method by an obfuscator. To the best of our knowledge, MACNETO is the first system to identify topics of programs from machine code.

LDA models each document as a sequence of words, where each document can exhibit a mixture of different topics. Each topic is subsequently modeled as a vector of words. Here, we model instruction distribution of a method as a document where each instruction is a word. Thus, following the concept of LDA, a *machine topic* becomes a vector of instructions (machine words) and can be represented as:

$$MT_i = [Pr_i^{I_1}, Pr_i^{I_2}...Pr_i^{I_a}] \tag{5}$$

, where MT_i is the i_{th} machine topic and $Pr_i^{I_a}$ represents the probability of the instruction I_a belonging to the topic in MT_i. Each method can also be modeled in terms of machine topic vector (MTV):

$$MTV(M_j) = [Pr_j^{MT_1}, Pr_j^{MT_2}...Pr_j^{MT_b}] \tag{6}$$

, where M_j represents the j_{th} method and MT_b represents the b_{th} machine topic. $Pr_j^{MT_b}$ represents the probability/membership of the method M_j belonging to the topic MT_b.

In MACNETO, we define 35 machine topics ($b = 35$) and have 252 types of instructions ($a = 252$) as we listed in Table I. While optimizing the topic number is out of the scope of this paper, we observe that 35 machine topics can split all methods in a reasonable way. Using these 35 machine topics, we generate unique machine topic vector (MTV). Note that, the dimension of each MTV is same as the topic number, *i.e.,* 35, although the number of topic vectors can be potentially infinite due to different probability values (see Equation 6). Thus, a unique method M_j can have a unique topic vector $MTV(M_j)$ that encodes the probability of the method belonging to each machine topic. $MTV(M_j)$ becomes the semantic representation of both M_j and its obfuscated counter part M_j^{obfus}. We annotate each original and its obfuscated method with the corresponding machine topic vector and use them to train our RNN based classifier, which will be discussed in Section IV-C. To compute machine topics and topic vector for each method, we use the Mallet library [17].

In the next two steps, MACNETO aims to deobfuscate an obfuscated method using a RNN based deep learning technique. In the training phase, the RNN learns the semantic relationship between a original and obfuscated method through their unique MTV. Next, in the testing (deobfuscating) phase, given a obfuscated method, RNN retrieves a set of candidate method having similar MTVs with the obfuscated method. MACNETO then scored these candidate methods and outputs a ranked list of original methods with similar MTVs.

C. Learning Phase

In this step, MACNETO uses a RNN based deep learning technique [18] to project the low-level features (Instruction Distributions) of methods to a relevant distribution of machine topics (Machine Topic Vector). MACNETO treats MTV as a proxy for program semantics, which should be invariant before and after obfuscation. Thus, MTV can serve as a signature (*i.e.,* class) of both original and obfuscated methods. Given a training method set T, MACNETO attempts to project each method $M_j \in T$ and its obfuscated counterpart M_j^{obfus} to the same MTV, *i.e.,* $M_j \rightarrow MTV(M_j) \leftarrow M_j^{obfus}$.

Similar deep learning technique is widely adopted to classify data [18]. However, most of these data comes with pre-annotated classes to facilitate learning. For example, Socher et al. [18] uses deep learning to classify images to relevant wordings. Such work has benchmarked images accompanied with correct descriptions in words to train such classifiers, MACNETO does not have any similar benchmarks. However, MACNETO *does* have available sets of applications, and has access to obfuscators. Hence, MACNETO builds a training set by co-training a classifier on obfuscated and deobfuscated methods (with MACNETO knowing the mapping from each training method to its obfuscated counterpart).

MACNETO characterizes each method M_j and M_j^{obfus} by the same machine topic vector $MTV(M_j)$, allowing it to automatically tag each method for training program classifiers. Given an unknown obfuscated method, MACNETO can first classify it to relevant machine topics (a MTV), which helps quickly search for similar and relevant original method. Only these original methods sharing similar MTVs with the unknown method will be scored and ranked by MACNETO, which enhances both system performance and effectiveness of deobfuscation.

To train such projection/mapping, MACNETO tries to minimize the following objective function

$$\begin{aligned} J(\Theta) = \sum_{M_j \in T} & \left\| MTV(M_j) - g(\theta^{(2)} \cdot f(\theta^{(1)} \cdot M_j)) \right\|^2 \\ & + \left\| MTV(M_j) - g(\theta^{(2)} \cdot f(\theta^{(1)} \cdot M_j^{obfus})) \right\|^2 \end{aligned} \tag{7}$$

, where T is a training method set, $MTV(M_j) \in \mathbb{R}^b$ (because MACNETO defines b machine topics), $\Theta = (\theta^{(1)}, \theta^{(2)})$, $\theta^{(1)} \in \mathbb{R}^{h \times a}$ and $\theta^{(2)} \in \mathbb{R}^{b \times h}$. For hidden layers, MACNETO uses $tanh$ function ($f(.)$) as the first layer and uses $logistic$ function ($g(.)$) as the second layer. MACNETO uses the technique of stochastic gradient descent (SGD) to solve this objective function.

As we discussed in Section IV-B, there can be infinite classification (MTV) in MACNETO, which may result in the un-convergence of our classifier learning. Thus in this learning phase, we select those methods having high memberships (> 0.67) toward specific machine topics. Our experiment result shows that the classifier built on these high-membership methods actually work on all methods (see Section V).

D. Deobfuscating

Taking an obfuscated method as a query, MACNETO attempts to locate which original method in the codebase have the lowest distance from it. The RNN in MACNETO can effectively infer the machine topic vector (MTV) of an unknown obfuscated method and then locate a set of original candidates having similar MTVs measured by the cosine similarity.

To further score and rank candidates, we develop a scoring model, which takes both semantic information and structural information of programs [19], [20] into account. For semantic

information, we use the instruction distribution as the feature. For structural information, we select the features of methods on callgraphs, which include centrality (PageRank in this paper), in-degree (how many other methods call this method) and out-degree (how many other methods this method calls) of the method. We then use a linear combination to compute distance between two methods:

$$Dist(M_k, M_l, W) = w_{inst} * Dist_{kl}^{inst} + w_c * Dist_{kl}^{c} + w_{in} * Dist_{kl}^{in} + w_{out} * Dist_{kl}^{out} \quad (8)$$

, where k and l are method indices, and $W = \{w_{inst}, w_c, w_{in}, w_{out}\}$. For computing $Dist_{kl}^{inst}$, we apply cosine similarity, while for the other three features, we compute the absolute differences between two methods.

Because our objective is to maximize the precision of the deobfuscation, we can formalize our objective function as

$$\underset{W}{\operatorname{argmax}} \sum_i I(M_i, Deob(M_i^{obfus}, T, Dist(M_k, M_l, W))) \quad (9)$$

, where T is a training method set, $Deob(.)$ returns the nearest neighbor method of M_i^{obfus} based on the distance function and $I(.)$ returns 1 if the results from $Deob(.)$ is M_i else return 0. To solve this function, we apply Simulated Annealing [21] to MACNETO for optimizing the weighting numbers W.

V. EVALUATION

We evaluated the performance of MACNETO on two popular obfuscators: ProGuard [4] and Allatori [5]. For each obfuscator, we gave MACNETO the task of recovering the original version of each obfuscated method. We selected these obfuscators based on a recent survey of Android obfuscators, selecting ProGuard for its widespread adoption and Allatori for its complex control and data-flow modification transformations [22]. We performed our evaluation on the most recent versions of these tools at time of writing: ProGuard 5.3.2 and Allatori 6.0. In particular, we answer the following two research questions:

- **RQ1:** How accurately can MACNETO deobfuscate methods transformed by a lexical obfuscator?
- **RQ2:** How accurately can MACNETO deobfuscate methods that are obfuscated using control and data transformation?

To judge MACNETO's precision for method deobfuscation, we needed a benchmark of plain apps (that is, not obfuscated) from which we could construct training and testing sets. For each app, we applied both ProGuard and Allatori, each of which output a mapping file (to aid in debugging) between the original (not obfuscated) method, and the obfuscated equivalent. Hence, we used the $1,611$ Android apps from the F-Droid open-source repository of Android apps as experimental subjects [23]. In our experiments, we vary the numbers of apps included in training and testing app sets and then randomly select apps into both sets.

We first split these apps into a training set and a testing set and then obfuscate each of them. Both the original and obfuscated training sets are used to train the program classifier using the first three steps outlined above. To evaluate the deobfuscation precision of MACNETO, we use methods in each app in the obfuscated testing set as a query to see if the original versions of these obfuscated methods can be retrieved by MACNETO from the original testing set. In our training and testing phase in this paper, we filter out the trivial methods having very few instructions (< 30) or very few types of instructions (< 10), because they may not offer sufficient information for MACNETO to deobfuscate. The constructor methods `<init>` and `<clinit>` are also excluded, because their functionality is usually setting up fields in classes/objects without too much logic.

We compare our results directly with the state-of-the-art Android deobfuscator *DeGuard* [3]. While DeGuard can support inferring other obfuscated information, such as field names and data types in programs in addition to method names, we only compare MACNETO's capability to recover method names. Note that the evaluate suite we used (from F-Droid) matches the same suite used in Bichsel et al.'s evaluation of DeGuard [3]. The size of full app set from F-Droid we use is slightly different with DeGuard: we have $1,611$ apps but DeGuard has $1,784$. This is because about 170 apps cannot be processed by the Allatori obfuscator or use some 3_{rd} party libs that are not included in app, which are detected by MACNETO. DeGuard's approach is not applicable to obfuscators that transform control flow (such as Allatori), and hence, we only include DeGuard results for the ProGuard experiments.

As a baseline, we also compare MACNETO to a naïve approach that simply calculates the distance between two methods using the feature-scoring equation presented in the previous section (equation 8). This baseline does not include MACNETO's topic modeling classifier and weighting number optimization for the scoring equation.

A. Evaluation Metrics

We use two metrics to evaluate MACNETO's performances to deobfuscate programs: precision and Top@K. We first define a testing method set as $\{M_i | i \in \mathbb{R}\}$, and its obfuscated counterpart as $\{M_j^{obfus} | j \in \mathbb{R}\}$, where i and j are the method indices. The definition of precision is

$$precision = \frac{\sum_j I(M_j, Deob(M_j^{obfus}))}{\left|\{M_j^{obfus}\}\right|} \quad (10)$$

, where $Deob(.)$ returns the deobfuscation result of M_j^{obfus} by a system and $I(.)$ returns 1 if the result of $Deob(.)$ is the same with the real original version M_j, else returns 0.

As we discussed in Section I, we model the deobfuscation problem as a nearest neighbor search problem. Thus, we also use Top@K, which is widely adopted to measure the performance of search systems, as an evaluation metric. Top@K is

TABLE II: DEOBFUSCATION RESULTS ON PROGUARD.

Train	Test	System	Top@1	Top@3	Top@10
1501	110	MACNETO	96.29%	99.31%	99.86%
		DeGuard	66.59%(80%)	N/A	N/A
		Naïve	53.84%	74.79%	90.79%

a generalized version of precision, if we replace $Deob(.)$ in Eq. 11 by a ranking function:

$$Top@K = \frac{\sum_j I(M_j, Rank(M_j^{obfus}, \{M_i\}, K))}{\left|\{M_j^{obfus}\}\right|} \quad (11)$$

, where $Rank(.)$ first computes the distance between M_j^{obfus} and each method in the testing method set $\{M_i\}$ by a distance function (Eq. 8 in this paper), and then return K methods having the shortest distances from $\{M_i\}$. $I(.)$ returns 1 if the original version M_j is in the K methods returned by $Rank(.)$, else returns 0. Precision, then, is Top@1. In our experiments, we use $K = \{1, 3, 10\}$ to evaluate the system performance. For DeGuard, because it only predicts the best answer ($K = 1$) and we cannot access their source code after contacting the authors, we only evaluate DeGuard by precision.

RQ1. How accurately can MACNETO deobfuscate methods transformed by a lexical obfuscator?

To compare MACNETO with DeGuard, we use the same testing and training data sets as used to evaluate DeGuard [3], which includes 110 Android apps for testing. A direct comparison between the two systems is complicated: MACNETO only considers methods that are reachable from any entry point (e.g. those on a callgraph rooted by standard Android entry points), whereas DeGuard considers all methods in an app for deobfuscation (including those that could never be called). Because the source code of DeGuard is not freely available online (or from the authors), we were not able to modify DeGuard to fit our evaluation. For completeness, we include the comparison results with this caveat.

We use the rest $1,501$ apps as the training data set to train MACNETO and deobfuscate these same 110 apps used in the evaluation of DeGuard. Our evaluation results on three systems can be found in Table II. On this task, we found that the precision (i.e. Top@1) of MACNETO was 96.29%. Top@3 and Top@10 of MACNETO are 99.31% and 99.86%, respectively.

While Bichsel et al. report the overall precision (80%) of DeGuard on all program properties including method names, field name, data type, etc., the exact precision for each specific program property is not reported (except graphically) [3]. To determine the precision of DeGuard on method names alone, we used the DeGuard web interface [24] to attempt to deobfuscate these same 110 apps. In this experiment, we found DeGuard's precision on deobfuscating method names to be 66.59% by our evaluation, matching the results in Figure 6 of the original DeGuard paper [3].

MACNETO achieves almost perfect deobfuscation on ProGuard in our evaluation. ProGuard renames identifiers (human words) in programs without further program transformation, such as changing control flow. Thus, the instruction distributions and structural information of each method are similar before and after ProGuard's obfuscation. The program classifier and the scoring function in MACNETO can resolve such identifier renaming, because the information in the machine code mostly stays similar.

We also apply the naïve approach developed by us to deobfuscate these apps. The precision is 53.84%, 74.79% and 90.79% for Top@1, Top@3 and Top@10, respectively. Comparing MACNETO with the naïve approach, we can find that our deep learning based program classifier can help enhance the precision of deobfuscation. The naïve approach also relies on the instruction distribution to deobfucate methods, but this approach seems to be too sensitive. Its precision (Top@1) is only 53.84%, even though its Top@10 is reasonable: $\sim 91\%$ of methods can be ranked in the top 10 positions.

Result 1: MACNETO can deobfuscate lexically obfuscated methods with 96% precision. It outperforms a naïve approach and previous tool DeGuard by 42 and 30 percentage points respectively.

RQ2. How accurately can MACNETO deobfuscate methods that are obfuscated using control and data transformation?

Compared with ProGuard, which mainly focuses on renaming identifiers in programs, Allatori changes control flow and encrypts/decrypts strings via inserting additional methods into programs. To demonstrate the performance of MACNETO against such advanced obfuscations, we trained 6 deobfuscation models (varying several parameters such as the sizes of training and testing app sets of MACNETO) and report the precision and Top@K. We consider building and applying each model to two types of testing methods: 1) all methods that have been obfuscated, and 2) only *significant* methods that have been obfuscated, as determined by those that have relatively high memberships (> 0.67) toward any machine topic. This leads to 12 results (6 models $*$ 2 types of method set) in our evaluation.

The overall results of the 6 models on 2 types of method sets as well as the comparison results between MACNETO and the naïve approach can be found in Table III. In Table III, the "ID" column represents the configuration ID, where we have 12 configurations in total. The "Train apps" and "Test apps" columns represent the numbers of training and testing apps, respectively. Note that the size of training apps does not matter to the naïve approach, because it simply relies on instruction distributions and the vanilla weighting numbers to deobfuscate programs. We randomly split all apps from F-Droid into our training and testing app sets. Note that the configuration 4 with $\{Train = 1501, Test = 110\}$ matches the exact same configuration used in the original evaluation of DeGuard. The "Method Selection" column denotes the two types of method selection based on the filtration criteria of methods' membership towards any topic as discussed above, and the "Method" column records the number of testing method. The "System" column shows which system we evaluate, and the "Gdiff" column shows if our graph diff module discussed in Section IV-A is enabled or not.

As discussed in Section IV-D, MACNETO uses the program

TABLE III: RESULTS OF DEOBFUSCATING ALLATORI-OBFUSCATED CODE.

ID	#Train APKs	#Test APKs	Method Selection	Methods	System	Gdiff.	Worst comp.	Real comp.	Filter	Top@1	Top@3	Top@10	Boost@1
#1	1501	110	0.67	1,932	MACNETO Naïve	✓	3.92 M	2.04 M	47.85%	91.10% 29.76%	96.95% 33.59%	98.81% 35.92%	206% ↑
#2	1501	110	N/A	12,690	MACNETO Naïve	✓	174.30 M	11.70 M	93.29%	81.33% 43.56%	89.55% 48.79%	92.36% 58.52%	87% ↑
#3	1501	110	0.67	2,148	MACNETO Naïve		4.81 M	2.77 M	42.35%	75.42% 22.77%	90.27% 26.16%	95.58% 30.30%	231% ↑
#4	1501	110	N/A	12,695	MACNETO Naïve		174.37 M	11.28 M	93.53%	68.96% 34.21%	82.47% 40.48%	89.37% 51.45%	102% ↑
#5	1111	500	0.67	12,533	MACNETO Naïve	✓	213.60 M	46.82 M	78.08%	69.62% 40.12%	82.37% 45.83%	92.25% 51.36%	74% ↑
#6	1111	500	N/A	97,578	MACNETO Naïve	✓	11203.32 M	500.01 M	95.54%	69.86% 29.16%	81.84% 33.26%	89.61% 37.71%	140% ↑
#7	1111	500	0.67	10,798	MACNETO Naïve		161.68 M	38.78 M	76.02%	52.15% 33.11%	65.91% 39.78%	78.16% 46.33%	58% ↑
#8	1111	500	N/A	97,567	MACNETO Naïve		11202.06 M	464.81 M	95.85%	50.30% 20.19%	63.14% 23.91%	73.52% 27.86%	149% ↑
#9	611	1000	0.67	30,102	MACNETO Naïve	✓	1130.81 M	117.40 M	89.62%	63.99% 27.90%	78.11% 32.79%	87.09% 42.52%	129% ↑
#10	611	1000	N/A	152,817	MACNETO Naïve	✓	26991.00 M	984.13 M	96.35%	69.56% 33.43%	81.37% 38.01%	88.63% 43.97%	108% ↑
#11	611	1000	0.67	25,064	MACNETO Naïve		805.63 M	79.81 M	90.09%	35.59% 16.53%	53.79% 20.30%	66.57% 25.85%	115% ↑
#12	611	1000	N/A	152,817	MACNETO Naïve		26991.00 M	1082.85 M	95.99%	48.51% 23.65%	60.42% 27.83%	69.56% 32.25%	105% ↑

Column Description: ID: Conguration ID; Train APKs and Test APKs: numbers of training and testing APKs, respectively; Method Selection: denotes a method's membership towards a machine topic; Method: total number of testing methods; System: system under evaluation; Gdiff: whether graph diff module is enabled or not; Worst comp.: total number of method comparisons *without* program classifier; Real comp.: total number of method comparisons with the program classifier; Filter: the percentage of unnecessary comparisons that is saved by MACNETO; Boost@1: enhancement achieve by MACNETO over the naïve approach on precision (Top@1).

classifier to filter out those methods that do not share the similar classification (MTV) of a given obfuscated method. The "Worst comp." column shows the total comparison numbers at the scale of million methods in the deobfuscation stage (see Section IV-D) *without* the program classifier that MACNETO would have to perform, and the "Real comp." column reports the total comparison number with the program classifier. The "Filter" column represents the percentage of unnecessary comparisons that can be saved by MACNETO with the program classifier. The "Top@K" columns, where $K = \{1, 3, 10\}$, are self-explanatory. The "Boost@1" column shows the enhancement achieve by MACNETO over the naïve approach on precision (Top@1).

There are three key findings we observe in Table III

1) Good deobfuscation performance: In general, MACNETO can achieve $80 + \%$ Top@10 under most configurations. As the number of the training apps grows, MACNETO can even achieve $\sim 99\%$ Top@10.
2) Effectiveness of the program classifier trained by deep learning: The filter rate of unnecessary comparisons is usually higher than 80% for most configurations. Such filtering achieved by our program classifier also enhances the system performance of MACNETO to deobfuscate programs, which will be discussed in the following paragraphs.
3) Effectiveness of *graph diff*: Given the same training-testing app set, MACNETO can have up to 20% enhancement on Top@10 by enabling graph diff. Allatori encrypts string literals in programs, so it injects additional methods into programs to decrypt these strings, which changes the program structure and instruction distribution. Our graph diff. module precisely identifies these injected decryption. With the graph diff. module, MACNETO achieves roughly 90% Top@10 when the size of the training data is small.

The Boost column shows the precent increase in precision from the naïve approach to MACNETO —note that this improvement is more than 100% in most models. As the number of testing apps increases, which means that the number of training apps decreases (this only influences MACNETO, since the naïve approach does not have a training phase), the performance of both system drops, even though MACNETO still outperforms the naïve approach. MACNETO offers a relatively stable precision when the size of testing apps ($152K+$ methods in 1000 apps) is large and the size of training apps (611 apps) is small: $\sim 70\%$ Top@10, while the naïve approach drops to about 32%. By enabling the graph diff. module, MACNETO can achieve $88 + \%$ Top@10, which does not even drop significantly due to the increase of the search range (testing set).

The same finding can also be observed when we evaluate both systems on the significant methods having memberships > 0.67 toward specific machine topics in the testing app set. While the performance of the naïve approach does not improve significantly, MACNETO has about 10% improvement on the

precision compared with testing on all methods in Table III, when the size of test apps is 110. This results in MACNETO achieves 200% boost over the naïve approach.

In general, the deobfuscation precisions of MACNETO steadily increase as the size of the training/testing set increases/decreases, while the precisions of naïve approach do not enhance significantly as the size of search range (testing set) decreases. For most of the configurations, MACNETO can rank the correct original versions of 80% of the obfuscated methods in the top 10 position. Even though the training app set is small (611 apps), the Top@10 offered by MACNETO is still about $\sim 70\%$.

Result 2: MACNETO can deobfuscate up to 91% precision. It significantly outperforms a naiïve approach in all the tested configurations.

B. Threats to Validity

In our evaluation, we collect 1600+ apps from an open-source repository. It is possible that such app set is not representative enough. Also, we apply MACNETO to recover programs obfuscated by two obfuscators in our evaluation. Some threat models and transformation techniques adopted by other obfuscators may not be evaluated in this paper. In the future, we plan to collect more apps and obfuscators to test and enhance the system capability of MACNETO.

When we split the apps into training set and testing set, we use a random approach. It is possible that MACNETO may not perform well on some testing sets, but we only have 3 in our evaluation. To enhance the confidence on MACNETO, we can conduct a K-fold cross validation [25] to make sure the performance of MACNETO is stable.

C. Limitations

Currently, MACNETO relies on the callgraph of the app to understand the semantic of each method and merge instruction distributions. Thus, only the methods in the callgraph can be deobfuscated by MACNETO. These methods are potentially the ones that will be executed in runtime.

The graph diff. module in MACNETO can identify the methods frequently injected by the obfuscator. By using the instruction distribution as the pattern, MACNETO can compute which patterns appear frequently in the training apps after obfuscation and then filter out the methods having the same instruction pattern in the testing apps. One possible obfuscation that MACNETO may not be able to handle is to randomly generate methods, where each method has different instruction distribution. In this way, MACNETO may not be able to determine which methods are intentionally injected by the obfuscator. The other possibility is that some methods having the similar instruction distributions with the detected patterns can be falsely filtered by MACNETO. Further, because MACNETO recursively merges instructions from callee methods into their caller methods, even if a single method is falsely filtered, a large portions of method can be affected in a negative way. One potential solution for these two limitations is to use data flow analysis [10] in MACNETO to determine which callees may not influence the result of the current method so that they can be safely removed.

VI. Related Work

Although in a programming language identifier names can be arbitrary, real developers usually use meaningful names for program comprehension [26]. Allamanis *et al.* [27] reported that code reviewers often suggest to modify identifier names before accepting a code patch. Thus, in recent years, naming convention of program identifiers drew significant attention for improving program understanding and maintenance [27]–[32]. Among the identifiers, a good method name is particularly helpful because they often summarize the underlying functionalities of the corresponding methods [33], [34]. Using a rule-based technique, Host *et al.* [33] inferred method names for Java programs using the methods' arguments, control-flow, and return types. In contrast, Allamanis *et al.* used a neural network model for predicting method names of Java code [34]. Although these two studies can suggest better method names in case of naming bugs, they do not look at the obfuscated methods that can even change the structure of the program.

JSNice [35] and DeGuard [3] apply statistical models to suggest names and identifiers in JavaScript and Java code, respectively. These statistical models work well against so called "minimizers" — obfuscators that replace identifier names with shorter names, without making any other transformations. These approaches can not be applied to obfuscators that modify program structure or control flow.

While MACNETO uses topic models as a proxy for application and method behavior, a variety of other systems use input/output behavior [36]–[38], call graph similarity [1], [2], or dynamic symbolic execution [39]–[41]. MACNETO is perhaps most similar to systems that rely on software birthmarks, which use some representative components of a program's execution (often calls to certain APIs) to create an obfuscation-resilient fingerprint to identify theft and reuse [42]–[47]. One concern in birthmarking is determining which APIs should be used to create the birthmark: perhaps some API calls are more identifying than others. MACNETO extends the notion of software birthmarking by using deep learning to identify patterns of API calls and instruction mix, allowing it be an effective deobfuscator. Further, MACNETO extends the notion of birthmarks by considering not just the code in a single method, but also instructions called by it.

VII. Conclusion

Deobfuscation is an important technique to reverse-engineer an obfuscated program to its original version, which can facilitate developers understand and analyze the program. We present MACNETO, which leverages topic modeling on instructions and deep learning techniques to deobfuscate programs automatically. In two large scale experiments, we apply MACNETO to deobufscate 1600+ Android APKs obfuscated by two well-known obfuscators. Our experiment results show that MACNETO achieves 96 + % precision on recovering obfuscated programs by ProGuard, which renames identifiers

in APKs. MACNETO also offers great precision on recovering programs obfuscated by an advanced obfuscator, Allatori, which changes control flow and inserts additional methods into APKs in addition to renaming identifiers. Compared with a naïve approach relying on instruction distribution of the obfuscated method to search for the most similar original version in the codebase, MACNETO has up to 200% boost on the deobfuscation precision.

REFERENCES

[1] F.-H. Su, J. Bell, K. Harvey, S. Sethumadhavan, G. Kaiser, and T. Jebara, "Code Relatives: Detecting Similarly Behaving Software," in *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, November 2016, pp. 702–714, artifact accepted as platinum. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950321

[2] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in *23rd Annual Network and Distributed System Security Symposium (NDSS)*, February 2016. [Online]. Available: https://www.internetsociety.org/sites/default/files/blogs-media/discovre-efficient-cross-architecture-identification-bugs-binary-code.pdf

[3] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical Deobfuscation of Android Applications," in *23rd ACM Conference on Computer and Communications Security*, ser. CCS 2016. New York, NY, USA: ACM, October 2016, pp. 343–355. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978422

[4] "Proguard." [Online]. Available: https://www.guardsquare.com/en/proguard

[5] "Allatori." [Online]. Available: http://www.allatori.com/

[6] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proffing, and obfuscation: Tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 735–746, Aug. 2002.

[7] D. Low, "Protecting java code via code obfuscation," *Crossroads*, vol. 4, no. 3, pp. 21–23, Apr. 1998.

[8] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, 2014.

[9] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: Taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 318–329. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931044

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, pp. 259–269.

[11] "Asm framework," http://asm.ow2.org/index.html.

[12] "Dex2jar." [Online]. Available: https://github.com/pxb1988/dex2jar

[13] "Android build numbers." [Online]. Available: https://source.android.com/source/build-numbers

[14] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, March 2003.

[15] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk, "Topicxp: Exploring topics in source code using latent dirichlet allocation," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10, 2010, pp. 1–6.

[16] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, pp. 1025–1035.

[17] "Mallet: Machine learning for language toolkit." [Online]. Available: http://mallet.cs.umass.edu/

[18] R. Socher, M. Ganjoo, C. D. Manning, and A. Y. Ng, "Zero-shot learning through cross-modal transfer," in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, ser. NIPS'13, 2013, pp. 935–943.

[19] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[20] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 37:1–37:30, Oct. 2013.

[21] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[22] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016, pp. 356–367.

[23] "The f-droid repository." [Online]. Available: https://f-droid.org/

[24] "The website of deguard." [Online]. Available: http://apk-deguard.com/

[25] S. Arlot and A. Celisse, "A survey of cross-validation procedures for model selection," *Statistics Surveys*, vol. 4, pp. 40–79, 2010.

[26] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proceedings of the 18th annual psychology of programming workshop*, 2006.

[27] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 281–293.

[28] S. Butler, M. Wermelinger, and Y. Yu, "Investigating naming convention adherence in java references," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, 2015, pp. 41–50.

[29] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Whats in a name? a study of identifiers," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006, pp. 3–12.

[30] 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.*, vol. 4, no. 3, pp. 143–167, 1996.

[31] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: What they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2016.

[32] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[33] E. W. Høst and B. M. Østvold, "Debugging method names," in *European Conference on Object-Oriented Programming*, 2009, pp. 294–317.

[34] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 38–49.

[35] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 111–124. [Online]. Available: http://doi.acm.org/10.1145/2676726.2677009

[36] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09, 2009, pp. 81–92.

[37] F.-H. Su, J. Bell, G. Kaiser, and S. Sethumadhavan, "Identifying functionally similar code in complex codebases," in *Proceedings of the 24th IEEE International Conference on Program Comprehension*, ser. ICPC 2016, 2016.

[38] E. Juergens, F. Deissenboeck, and B. Hummel, "Code similarities beyond copy & paste," in *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, ser. CSMR '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 78–87. [Online]. Available: http://dx.doi.org/10.1109/CSMR.2010.33

[39] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of android malware for effective malware comprehension, detection, and classification," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 306–317. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931043

[40] S. Li, X. Xiao, B. Bassett, T. Xie, and N. Tillmann, "Measuring code behavioral similarity for programming and software engineering education," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16, 2016, pp. 501–510.

[41] D. E. Krutz and E. Shihab, "Cccd: Concolic code clone detection," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 489–490.

[42] H. Tamada, M. Nakamura, and A. Monden, "Design and evaluation of birthmarks for detecting theft of java programs," in *Proc. IASTED International Conference on Software Engineering*, 2004, pp. 569–575.

[43] D. Schuler, V. Dallmeier, and C. Lindig, "A dynamic birthmark for java," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 274–283. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321672

[44] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012, pp. 364–374.

[45] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 303–313.

[46] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *2015 International Conference on Software Engineering (ICSE)*, ser. ICSE '15, 2015, pp. 426–436.

[47] M. Linares-Vásquez, C. Mcmillan, D. Poshyvanyk, and M. Grechanik, "On using machine learning to automatically classify software applications into domain categories," *Empirical Softw. Engg.*, vol. 19, no. 3, pp. 582–618, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-012-9230-z