Obfuscation Resilient Search through Executable Classification

Fang-Hsiang Su
Computer Science
Columbia University, USA
mikefhsu@cs.columbia.edu

Jonathan Bell
Computer Science
George Mason University, USA
bellj@gmu.edu

Gail Kaiser
Computer Science
Columbia University, USA
kaiser@cs.columbia.edu

Baishakhi Ray
Computer Science
Columbia University, USA
rayb@cs.columbia.edu

Abstract

Android applications are usually 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 and/or modifying program structures. It is challenging to search for executables relevant to an obfuscated application for developers to analyze efficiently. Prior approaches toward obfuscation resilient search 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 are not broken by obfuscators; others have assumed that control flow graphs maintain their structures. Both approaches can be easily defeated by a motivated obfuscator. We present a new approach, Macneto, to search for programs relevant to obfuscated executables leveraging deep learning and principal components on instructions. Macneto makes few assumptions about the kinds of modifications that an obfuscator might perform. We show that it has high search precision for executables obfuscated by a state-of-the-art obfuscator that changes control flow. Further, we also demonstrate the potential of Macneto to help developers understand executables, where Macneto infers keywords (which are from relevant un-obfuscated programs) for obfuscated executables.

MAPL'18, June 18, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5834-7/18/06…$15.00
https://doi.org/10.1145/3211346.3211352

CCS Concepts • **Security and privacy** → *Software reverse engineering*; • **Theory of computation** → *Program analysis*; • **Computing methodologies** → *Supervised learning by classification*;

Keywords executable search, bytecode search, obfuscation resilience, bytecode analysis, deep learning

ACM Reference Format:
Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Baishakhi Ray. 2018. Obfuscation Resilient Search through Executable Classification. In *Proceedings of 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3211346.3211352

1 Introduction

Android apps are typically obfuscated before delivery to decrease the size of distributed binaries and reduce disallowed reuse. Malware authors may take advantage of the general expectation that Android code is obfuscated to pass off obfuscated malware as benign 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 obfuscation resilient search: tools that can automatically find program structures (in a known codebase) likely to be similar to the original version of code that has been obfuscated.

Obfuscation resilient search can be used in various automated analyses, for instance, plagiarism detection [39] or detecting precise versions of third party libraries [25] embedded in applications, allowing auditors to identify the use of vulnerable libraries. Similarly, obfuscation resilient search can search among un-obfuscated apps to recover identifiers [11] and/or control flow [51] of an obfuscated app. Obfuscation resilient search can be useful in human-guided analysis, where an engineer inspects applications to determine security risks.

In general, searching for code likely to be similar to an obfuscated program relies on some training set of pairs of un-obfuscated code and its obfuscated counterpart to build a model. Once trained, the code search engine can match

obfuscated code to its original un-obfuscated code when both obfuscated and un-obfuscated versions are in the corpus. In the more typical use case when only the obfuscated code is at hand, and the un-obfuscated version is unknown to the analyst, the code search may be able to find known code highly likely to be similar to the un-obfuscated version.

For example, some code search "deobfuscation" tools rely on the structure of an application's control flow graph. However, they are susceptible to obfuscators that introduce extra basic blocks and jumps to the application's code and can be slow to use, requiring many pair-wise comparisons to perform their task [16, 46]. Using another approach, DeGuard [11] is a state-of-the-art deobfuscator that builds a probabilistic model for identifiers based on the co-occurrence of names. While this technique can be very fast to apply (after the statistical model is trained), it may be defeated by obfuscators that introduce new fields among classes.

We present a novel approach for automated obfuscation resilient search for Android apps, using deep learning: Macneto, which searches at bytecode level, instead of source code. Macneto leverages a key observation about obfuscation: an obfuscator's goal is to transform how a program looks as radically as possible, while maintaining the original program semantics. Macneto works by learning lightweight (partial) executable semantics through Principal Component Analysis (PCA). These PCA 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 assumed stable by other deobfuscators). Macneto's deep PCA model is resilient to multiple obfuscation techniques [18, 44], including identifier renaming and control flow modifications.

Macneto uses deep learning to train a classifier on known pairs of un-obfuscated/obfuscated apps offline. This training process allows Macneto to be potentially applicable to various obfuscators: supporting a new obfuscator using the same kinds of obfuscations would only require a new data set of pairs of the original un-obfuscated apps and the corresponding obfuscated apps. Then, these models are saved for fast, online search where an unknown obfuscated executable is projected to principal components via deep learning, and matched to the most similar executables from the known corpus.

We evaluated Macneto on 1500+ real Android apps using an advanced obfuscator: Allatori [44]. Allatori supports name obfuscation (similar to what ProGuard does [18]) and also control flow obfuscations, e.g., it changes the standard Java constructions for loops, conditional and other branching instructions. Macneto achieves good search precision, about 80%, to retrieve relevant code given unknown obfuscated executables. It significantly outperforms two baseline approaches without deep learning.

The contributions of this paper are:

- A new approach to conduct obfuscation resilient code search leveraging deep learning and principal components (features) on bytecode.
- A new approach to automate classification of programs with similar semantics.
- An evaluation of our tool on an advanced obfuscator.
- An open source implementation of Macneto [1].

2 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. These obfuscations might include lexical transformations, control transformations, and data transformations [14]. 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). 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.

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 [31, 41]. 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. For encrypting/decrypting strings, an obfuscator can inject additional helper methods into programs [44].

In this paper we define the obfuscation resilient search problem as follows. A developer/security analyst has access to a set of obfuscated executables, and her job is to identify executables similar with its original version in the existing codebase. The developer/analyst can analyze the obfuscated executable with its similar ones to identify malware variants, which becomes a significantly easier problem. Thus, our task is similar to a search-based deobfuscator DeGuard [11].

We assume that obfuscators can make lexical, control, and data transformations to code. We do not base our search 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. We assume that there are some patterns (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 random with no pattern to the transformations that it makes, then MACNETO would be unable to apply its search model to other obfuscated apps. 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.

3 MACNETO Overview

From a set of obfuscated APKs, MACNETO intends to identify the relevant executables to the original version of a given obfuscated APK. 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 program executable are the summation of each individual method that it has. The semantics of a program executable are captured as the hidden principal components of its machine code instead of human texts such as identifier names in methods and/or descriptions of this program. By construction, an obfuscated program/application is semantically equivalent to its original, un-obfuscated version that it is based on. MACNETO assumes that the principal component vector of an obfuscated application will match those of the original application. In its learning phase, MACNETO is provided a training set of android applications (APKs), which are labeled pairs of obfuscated and original versions. Once training is complete, MACNETO can be presented with an arbitrary number of obfuscated applications, and for each return suggested applications from its codebase (that it had been trained on) that are similar to the unknown original application. In the event that the original version happens to exist in the corpus, MACNETO will match the obfuscated application with its original version.

MACNETO utilizes a four stage approach:

(i) *Computing Instruction Distribution.* For an application executable (original or obfuscated), MACNETO parses each method as a distribution of instructions. An application executable can then be represented as the summation of all of its methods, which is also a distribution of instructions. The instruction distribution of an application executable is analogous to the term frequency vector of a document, where we treat each application executable as a document.

(ii) *Principal Component Analysis.* Identifies principal components [37] from the instruction distribution of the original app. These principal components are used as a proxy for app semantics. The same PCA model is used later to annotate the corresponding obfuscated application.

(iii) *Learning.* Uses a three-layered Artificial Neural Network (ANN) [33] where the input is the instruction distribution of an application executable (original and obfuscated), and the output layer is the corresponding principal component vector of the original application. MACNETO uses this three-layered ANN as a program classifier that maps an original application and its obfuscated version to the same class represented by principal component vector. This is the training phase of the ANN model. Such model can be pre-trained.

(iv) *Obfuscation Resilient Search.* This is the testing phase of the ANN model. It operates on a set of original and obfuscated applications that form our testing set. Given an obfuscated application, the above ANN model tries to infer its principal component vector; MACNETO then finds a set of un-obfuscated applications with similar principal component vectors and ranks them as possible deobfuscated candidates.

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

Consider the example `readAndSort` program shown in Figure 2, assuming that this is an Android app that we are using to train MACNETO. To compute the instruction distribution of `readAndSort` application, MACNETO first recruits the data flow analysis to identify all possible methods which may be invoked at runtime, which are `readFile` and `sort`. The instruction distributions of these two callee methods will be incorporated into `readAndSort`. Then MACNETO moves to the next step, applying Principal Component Analysis (PCA) [37] on the instruction distributions of all applications including `readAndSort` in the training app set. The result of this step is a vector containing the value/membership that a method has toward each principal component: a Principal Component Vector (PCV).

Our insight is that while some instructions in our feature set can be correlated, PCA can help us convert these instructions into orthogonal features. Furthermore, we can also understand which components are more important to classify application executables. These components can help drastically reduce the query time of MACNETO to search for similar executables. MACNETO annotates both the original and obfuscated versions of this application with this same PCV. This annotation process allows our learning phase to predict similar PCVs for an application and its obfuscated version, even their instruction distributions are different.

4 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

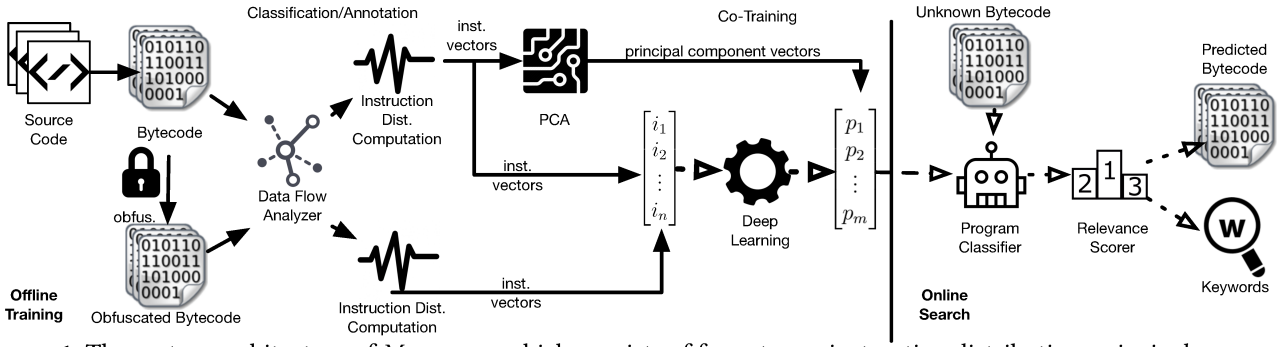

Figure 1. The system architecture of MACNETO, which consists of four stages: instruction distribution, principal component (feature) analysis on bytecode, deep-learning for classifying programs and online scoring, to deobfuscate Android executables.

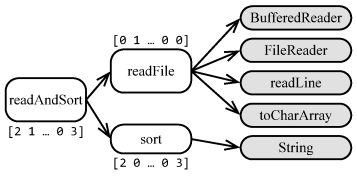

Figure 2. The `readAndSort` application and two of its methods, `readFile` and `sort`.

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.

4.1 Computing Instruction Distribution

We use the instruction distribution (ID) of an application to begin to approximate its behavior. The ID is a vector that represents the frequencies of important instructions. We use a bottom-up approach to compute the ID of an application. Given an application executable A_j, its ID is the summation of all possible methods invoked at runtime. For computing the possible invoked methods of an application, we use FlowDroid [7], a state-of-the-art tool that uses context-, flow-, field-, and object-sensitive android lifecycle-aware control and data-flow analysis [7]. For a method M_j^k that belongs to A_j, its instruction distribution can be represented as $ID(M_j^k) = [freq_{I_1}, freq_{I_2}, ...freq_{I_n}]_j^k$, where n represents the index of an instruction and $freq_{I_n}$ represents the frequency of the n_{th} instruction in the method M_j^k. The ID of A_j can then be computed by $ID(A_j) = \sum_k ID(M_j^k)$. The `readAndSort` program in Figure 2 can be an example: the ID of `readAndSort` is the summation of two possibly invoked methods `readFile` and `sort`. This step is similar to building the term frequency vector for a document.

MACNETO considers various bytecode operations as individual instructions (e.g., adding or subtracting 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.

Table 1. MACNETO'S INSTRUCTION SET.

Opcode	Description
`xaload`	Load a primitive/object x from an array
`xastore`	Store a primitive/object x to an array
`arraylength`	Retrieve the length of an array.
`xadd`	Add two primitives of type x on the stack.
`xsub`	Subtract two primitives of type x on the stack.
`xmul`	Multiply two primitives of type x on the stack.
`xdiv`	Divide two primitives of type x on the stack.
`xrem`	Compute the remainder of two primitives x on the stack
`xneg`	Negate a primitive of type x on the stack.
`xshift`	Shift a primitive x (type integer/long) on the stack.
`xand`	Bitwise-and two primitives of type x (integer/long) on the stack.
`xor`	Bitwise-or two primitives of type x (integer/long) on the stack.
`x_xor`	Bitwise-xor two primitives x (integer/long) on the stack.
`iinc`	Increment an integer on the stack.
`xcomp`	Compare two primitives of type x on the stack
`ifXXX`	Represent all conditional jumps.
`xswitch`	Jump to a branch based on stack index.
`android_apis`	The APIs offered by the Android framework

To avoid over-specialization, MACNETO groups very similar instructions together and represents them with a single word, as shown in Table 1. 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.

To calculate these instruction distributions, MACNETO uses the ASM Java bytecode library [36], and Dex2Jar [12]. This allows MACNETO to search for 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 [5]. Including these Android APIs, MACNETO observes 252 types of instructions in total.

4.2 Principal Component Analysis on Executable

Principal Component Analysis [37] is a statistical technique that project data containing possibly correlated components

into an orthogonal feature space. While we believe that some instructions may have dependencies, PCA helps us project the IDs of all executables onto an orthogonal space, where each dimension is a principal component. MACNETO uses PCA to convert the ID of each application to a principal component vector (PCV):

$$PCV(A_j) = [Val_{P_1}, Val_{P_2}...Val_{P_m}]_j \quad (1)$$

, where A_j represents the j_{th} application executable in the application set and P_m represents the m_{th} principal component. Val_{P_m} represents the value that the application executable A_j has of the principal component P_m.

PCA accelerates the search time of MACNETO compared with a naïve approach using the ID directly to search for similar applications. More details regarding the performance of MACNETO can be found in Section 5. To the best of our knowledge, MACNETO is the first system to identify principal components of programs from machine code.

In MACNETO, we define 32 principal components ($m = 32$) and have 252 types of instructions ($n = 252$) as we listed in Table 1. Using these 32 principal components, we generate unique principal component vector (PCV). Note that, the dimension of each PCV is the same as the principal component number, i.e. 32, although the number of PCV can be potentially infinite due to different feature values (see Eq. 1). Thus, an application A_j can have a unique $PCV(A_j)$ that encodes the value of the application belonging to each principal component. $PCV(A_j)$ becomes the semantic representation of both A_j and its obfuscated counterpart A_j^{ob}. We annotate each original and its obfuscated application with the corresponding PCV and use them to train our ANN based classifier, which will be discussed in Section 4.3. To compute PCV, we use the scikit-learn [38] library on machine code.

In the next two steps, MACNETO aims to search for relevant executables to the original version of an obfuscated executable using a ANN based deep learning technique. In the training phase, the ANN learns the semantic relationship between a original and its obfuscated executable through their unique PCV. Next, in the testing (obfuscation resilient search) phase, given an obfuscated application executable, ANN retrieves a set of candidate applications having similar PCVs with the obfuscated application. MACNETO then scores these candidate applications and outputs a ranked list of relevant un-obfuscated applications with similar PCVs.

4.3 Learning Phase

In this step, MACNETO uses an ANN based deep learning technique [45] to project the low-level features (Instruction Distributions) of applications to a relevant vector of principal components (PCV). MACNETO treats PCV as a proxy for program semantics, which should be invariant before and after obfuscation. Thus, PCV can serve as a signature (i.e., class) of both original and obfuscated applications. Given a training application set T, MACNETO attempts to project each application $A_j \in T$ and its obfuscated counterpart A_j^{ob} to the same PCV, i.e., $A_j \rightarrow PCV(A_j) \leftarrow A_j^{ob}$.

Similar deep learning technique is widely adopted to classify data. However, most of data comes with pre-annotated classes to facilitate learning. For example, Socher et al. [45] 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 and co-trains a classifier on both obfuscated and original application executables with MACNETO knowing the mapping from each training application to its obfuscated counterpart. The learning phase of MACNETO is relevant to the Deep Structured Semantic Models (DSSM) [20], which projects two related corpuses having the same concepts, e.g., queries and their corresponding documents, onto the same feature space. DSSM can then maximize the similarity based on the relationship between a query and its corresponding document, which can be constructed by clickthrough rate data. DSSM offers MACNETO a future direction, where we can project the origin version of an execution and all of its obfuscated versions onto the same feature space.

MACNETO characterizes each application A_j and A_j^{ob} by the same principal component vector $PCV(A_j)$, allowing it to automatically tag each application for training program classifiers. Given an unknown obfuscated application, MACNETO can first classify it to relevant PCV, which helps quickly search for similar and/or original applications.

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

$$J(\Theta) = \sum_{A_j \in T} \|PCV(A_j) - l(\theta^{(3)} \cdot g(\theta^{(2)} \cdot f(\theta^{(1)} \cdot A_j)))\|^2 + \|PCV(A_j) - l(\theta^{(3)} \cdot g(\theta^{(2)} \cdot f(\theta^{(1)} \cdot A_j^{ob})))\|^2 \quad (2)$$

, where T is a training application set, $PCV(A_j) \in \mathbb{R}^n$ (because MACNETO defines n principal components) and $\Theta = (\theta^{(1)}, \theta^{(2)}, \theta^{(3)})$ defines the weighting numbers for each hidden layer. For hidden layers, MACNETO uses *relu* function, where $f(.)$ is the first layer containing 128 neurons, $g(.)$ is the second layer containing 64 neurons and $l(.)$ is the third layer containing 32 neurons. MACNETO uses the Adam [24] solver to solve this objective function. MACNETO build the ANN on the frameworks of tensorflow [2, 50] and keras [23].

4.4 Obfuscation Resilient Search

Taking an obfuscated application executable as a query, MACNETO attempts to locate which un-obfuscated application(s) in the codebase are mostly similar with it. The ANN in MACNETO can effectively infer the principal component vector

(PCV) of an unknown obfuscated executable and then locate a set of un-obfuscated candidates having similar PCVs measured by the cosine similarity. Given two application executables, their cosine similarity is defined as

$$similarity = \frac{PCV(A_i) \cdot PCV(A_j)}{\|PCV(A_i)\| * \|PCV(A_j)\|} \quad (3)$$

, where A_i and A_j are two application executables.

Before we detail the procedure of obfuscation resilient search, we define the terminology that we will use as follows.

- Tr: The training application executable set.
- Te: The testing application executable set.
- Tr_{or}: The training original application set, which is a subset of Tr. This is also the search space in our evaluation.
- Tr_{ob}: The training obfuscated application set, which is a subset of Tr and the obfuscated counterpart of Tr_{or}.
- Te_{or}: The testing original application set, which is a subset of Te.
- Te_{ob}: The training obfuscated application set, which is a subset of Te and the obfuscated counterpart of Te_{or}.
- A_j: The j_{th} application executable.
- A_j^{ob}: The obfuscated counterpart of A_j.
- $\sim A_j$: A similar application executable of A_j, where the similarity is measured by their PCV via Eq. 3.
- $\{\sim A_j\}$: A list of similar application executables sorted by their cosine similarity with A_j.

In our evaluation, we split all Android application executables and their obfuscated counterparts into a training set Tr and a testing set Te. For the training purpose, both Tr_{or} and Tr_{ob} are used to construct the ANN, where only Tr_{or} is recruited for building the PCA model. For the testing purpose, we use an A_j^{ob} in Te_{ob} as a query to search for n ($n = 10$ in this paper) similar application executables $\{\sim A_j^{ob}\}$ in Tr_{or}.

The original version of A_j^{ob}, A_j in Te_{or} is not in the search space, Tr_{or}. Thus, to verify the efficacy of Macneto, we use A_j as a query to search for the closest executable $\sim A_j$ in Tr_{or}, as the groundtruth. We then check the ranking of $\sim A_j$ in $\{\sim A_j^{ob}\}$. By this procedure, we evaluate the search performances of three systems including Macneto, which will be discussed in Section 5.

5 Evaluation

To evaluate the performance of Macneto, we design two large scale experiments to address two research questions based on an advanced obfuscator Allatori [44] as follows.

- **RQ1** Executable search: Given an unknown application executable that is obfuscated using lexical, control and data transformation, how accurately can Macneto search for relevant un-obfuscated executables?
- **RQ2** Executable understanding: Given an unknown application executable without source code and text description, can Macneto infer meaningful keywords for developers/program analyst to understand its semantics?

We selected the Allatori obfuscator based on a recent survey of Android obfuscators for its complex control and data-flow modification transformations [9]. We performed our evaluation on the most recent version at time of experimenting: Allatori 6.5. To judge Macneto's precision for obfuscation resilient search, we needed a benchmark of plain apps (that is, not obfuscated) from which we could construct training and testing sets. We used the 1,559 Android apps from the F-Droid repository as experimental subjects [17].

We first split these apps into a training set and a testing set and then systematically obfuscate each of them. Both the original and obfuscated training sets are used to train the program classifier using the first three steps outlined in Section 4. To evaluate the obfuscation resilient search precision of Macneto, we follow the procedure in Section 4.4 to compare the search results given an obfuscated application A_j^{ob} as a query and its original version A_j as a query.

As a baseline, we compare Macneto with two approaches as follows: (1) Naïve approach: Calculates the similarity between two applications based on their instruction distributions (IDs) described in Section 4.1 without PCA and deep learning. (2) Pure PCA approach: Calculates the similarity between two applications based on their PCV computed solely by PCA without deep learning.

The major difference between Macneto and the pure PCA approach is that while Macneto uses Tr_{or} to build a PCA model and use it to annotate both Tr_{or} and Tr_{ob} for deep learning, the pure PCA approach use the whole training set ($Tr_{or} + Tr_{ob}$) to build a PCA model without deep learning to transform ID of an application executable to PCV. The key insight here is that we believe an application A_j and its obfuscated counterpart A_j^{ob} should share the same semantic classification (PCV), but the pure PCA approach cannot guarantee this invariance. The PCVs of A_j and A_j^{ob} can be different by pure PCA, because their IDs can be different. This is why Macneto first uses PCA to compute the PCV for A_j and use the same PCV to annotate A_j^{ob}. The power of deep learning can then help Macneto recognize and search for similar application executables given an unknown obfuscated application. In our evaluations, we observe that the pure PCA approach can provide good search precision, but Macneto with deep learning can achieve even better precision.

5.1 Evaluation Metrics

We use two metrics to evaluate Macneto's performance to conduct obfuscation resilient search: Top@K and Mean Reciprocal Rank (MRR). By this procedure, we evaluate the search performances of three systems including Macneto, which will be discussed in Section 5.

- $Search(A_j, Tr_{or}, n)$: Given an application executable A_j, $Search(.)$ retrieves the most n ($n = 10$ in this paper) similar application executables in the search space, which is the training original application set Tr_{or} in this paper.
- $Best(A_j, Tr_{or})$: $Best(.)$ is a specialized version of $Search(.)$ to retrieve the most similar application executable ($n = 1$) in the search space.
- $Rank(A_k, \{A_l\}, K)$: Given an application executable A_k, $Rank(.)$ return 1 if the ranking of A_k in the application list $\{A_l\}$ is higher than or equal to K. If A_k is not in $\{A_l\}$ or its ranking is lower than K, $Rank(.)$ returns 0.

The definition of Top@K can then be

$$Top@K = \frac{\sum_{j \in Te_{or}} Rank(Best(A_j), Search(A_j^{ob}, n), K)}{|Te_{or}|} \quad (4)$$

, where A_j is an application executable, A_j^{ob} is its obfuscated counterpart. The search space is Tr_{or} for both $Search(.)$ and $Best(.)$, so we ignore it to simplify the definition. In our experiments, we use $K = \{1, 5, 10\}$ to evaluate the system performance.

The definition of MRR is

$$MRR = \frac{1}{|Te_{or}|} \sum_{j \in Te_{or}} \frac{1}{Rank(Best(A_j), Search(A_j^{ob}, n))} \quad (5)$$

, where the $Rank(.)$ here returns the ranking of $Best(A_j)$ in $Search(A_j^{ob}, n)$ directly. If $Best(A_j)$ is not in $Search(A_j^{ob})$, $Rank(.)$ returns 0.

5.2 Executable Search

In this section, we answer the research question: **RQ1.** Given an unknown application executable that is obfuscated using lexical, control and data transformation, how accurately can Macneto search for relevant un-obfuscated executables?

Compared with lexical obfuscators like ProGuard [18] that 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 to search for relevant application executables against such advanced obfuscations, we conduct a K-fold analysis ($K = 8$) on the Android application set we have. We first split the 1559 Android application executables we have into 8 folds. For each experiment, we use 7 folds to train an obfuscation resilient search model and use the other as the testing set (queries). In total, we trained 8 models for 8 experiments, where each application executable will be in the testing set for once and in the training set for 7 times. This K-fold analysis can help us verify the robustness of Macneto, because each executable will be tested.

The overall results of the 8 models trained by Macneto and two baseline approaches can be found in Table 2. In Table 2, the "Exp" column represents the experiment ID, where we have 8 folds (experiments) in total. The "Training APKs' and "Testing APKs" columns represent the numbers of training and testing apps, respectively. Note that the size of training apks does not matter to the naïve approach, because it simply relies on instruction distributions to search for programs. The reason that we offer two numbers, e.g., 1359 + 1359, for the training apks is that the training phase includes both original apks in Tr_{or} and their obfuscated counterparts in Tr_{ob}. The "Training Methods" and "Testing Methods" columns show the number of methods including in the training apks and testing apks, respectively. Note that while Macneto includes all possible invoked methods of executables (apks) to compute instruction distributions, our training and testing works at executable level. The "System" column shows which system we evaluate, where "PCA" and "Naïve" are the two baseline approaches we discussed in Section 5. The "Training Time" and "Query Time" column shows the time consumed by each system to complete the training and testing (query) in seconds. The "Top@K" columns, where $K = \{1, 5, 10\}$, and the "MRR" column is self-explanatory. The "Boost@1" column shows the improvement of Macneto and the pure PCA approach against the naïve approach.

We observe three key findings of Macneto in Table 2:

1. Good effectiveness of obfuscation resilient research: Macneto can achieve 80 + % Top@1 for most experiments.
2. Effectiveness of the executable classifier trained by deep learning: Compared with the naïve and the pure PCA approach without deep learning, Macneto achieves 17.76% and 8.72% enhancements of Top@1, respectively. In our experiment, PCA is an effective technique to extract application semantics, because the pure PCA approach already offers 8.31% enhancement of Top@1 over the naïve approach. Deep learning helps Macneto understand executables further (17.76% enhancement) with PCA.
3. Query performance by PCA: Compared with the naïve approach which searches for similar applications based on 252 types of instructions, Macneto and the pure PCA approach searches only by 32 principal components. This leads to great runtime performance of both systems to search for similar application executables: while the naïve approach needs 65.09 seconds in average to process 200 queries, Macneto and the pure PCA approach only need 24.09 and 20.13 seconds in average.

Result 1: Macneto can achieve up to 84% precision (Top@1) for searching for similar application executables given unknown and obfuscated executables as queries. It significantly outperforms two baseline approaches in precision and MRR.

5.3 Executable Understanding

In addition to searching for similar executables, we are interested in exploring the potential of Macneto to support developers quickly understanding an unknown application executable without human descriptions. We will answer the research question: **RQ2.** Given an unknown application executable without source code and text description, can

Table 2. Obfuscation resilient search results of Allatori-obfuscated code.

Exp	#Training APKs	#Training Methods	#Testing APKs	#Testing Methods	System	Training Time (sec)	Query Time (sec)	Top@1	Top@5	Top@10	MRR	Boost@1
#1	1359 + 1359	1.14M	200 + 200	152K	Macneto	2786.72	24.66	0.825	0.96	0.965	0.89	+22.22%
					PCA	0.0357	20.49	0.78	0.925	0.97	0.85	+15.56%
					Naïve	N/A	66.55	0.675	0.91	0.94	0.78	N/A
#2	1359 + 1359	1.08M	200 + 200	214K	Macneto	2791.56	24.88	0.82	0.94	0.955	0.87	+18.84%
					PCA	0.0357	20.73	0.75	0.925	0.95	0.86	+8.7%
					Naïve	N/A	67.5	0.69	0.91	0.94	0.79	N/A
#3	1359 + 1359	1.16M	200 + 200	137K	Macneto	2826.57	24.76	0.8	0.915	0.955	0.86	+22.13%
					PCA	0.0352	20.72	0.73	0.94	0.975	0.82	+11.45%
					Naïve	N/A	66.57	0.655	0.905	0.935	0.77	N/A
#4	1359 + 1359	1.15M	200 + 200	143K	Macneto	2848.41	24.68	0.755	0.905	0.925	0.82	+16.15%
					PCA	0.0349	20.71	0.715	0.915	0.95	0.80	+10%
					Naïve	N/A	66.7	0.65	0.865	0.9	0.79	N/A
#5	1359 + 1359	1.14M	200 + 200	157K	Macneto	2842.08	24.72	0.84	0.945	0.96	0.89	+15.86%
					PCA	0.0336	20.7	0.765	0.93	0.94	0.80	+5.5%
					Naïve	N/A	67.37	0.725	0.89	0.935	0.79	N/A
#6	1359 + 1359	1.14M	200 + 200	152K	Macneto	2866.28	24.77	0.795	0.95	0.96	0.86	+16.05%
					PCA	0.0342	20.65	0.72	0.95	0.97	0.85	+5.1%
					Naïve	N/A	66.64	0.685	0.865	0.935	0.77	N/A
#7	1359 + 1359	1.13M	200 + 200	166K	Macneto	2866.14	24.81	0.82	0.915	0.935	0.86	+29.13%
					PCA	0.0347	20.65	0.735	0.94	0.95	0.82	+15.75%
					Naïve	N/A	66.68	0.635	0.87	0.915	0.74	N/A
#8	1400 + 1400	1.13M	159 + 159	167K	Macneto	2958.32	20.192	0.79	0.90	0.93	0.84	+4.2%
					PCA	0.039	16.94	0.73	0.91	0.93	0.81	-3.3%
					Naïve	N/A	54.5	0.75	0.90	0.93	0.82	N/A

Column Description: Exp: Experiment ID; Train APKs and Test APKs: numbers of training and testing APKs, respectively; Training Methods and Testing Methods: denote the method numbers belonging to Training APKs and Testing APKs, respectively; System: system under evaluation; Training Time: the time that each system spends to training the model for each experiment; Query Time: the time that each system spends to search for relevant executables given an unknown and obfuscated executable; Top@K: the percentage of queries (executables), where their original versions are retrieved and ranked by each system at or better than K_{th} position; MRR: Mean Reciprocal Ranking of each system; Boost@1: the enhancements achieve by Macneto and the pure PCA approach over the naïve approach on precision (Top@1).

Macneto infer meaningful keywords for developers/program analyst to understand its semantics?

We crawled the F-Droid repository [17] to extract the description for each un-obfuscated APK. Then we follow the search procedure in Section 4.4 to retrieve 10 most similar application executables in Tr_{or}, given an obfuscated executable A_j^{ob}. Among these 10 similar application executables, we use a TF-IDF [40] model to extract a set of keywords from their descriptions. From this set of keywords, we select top 10 keywords having the highest TF-IDF values. These selected keywords become the human semantic that we predict for an application executable A_j^{ob} without human descriptions. To verify the correctness of these keywords, we compare with the real description of A_j, the original version of A_j^{ob}.

To conduct this experiment, we randomly select 1,539 application executables as the training set and use the rest 20 as the testing set. We then manually compare the predicted keyword set of A_j^{ob} with the real description of its original version A_j. Due to the page limitation, we are not able to offer all the results. Instead, we list some interesting cases that Macneto precisely infers meaningful words to describe an unknown and obfuscated executable without any descriptions.

- `com.platypus.SAnd`: The partial description of this executable is "Use your phones sensors...to show your current **orientation**, height and air pressure...". Given the obfuscated version of this executable as a query, Macneto infers a relevant keyword **coordinates**, where the pure PCA and the naïve approaches fail to offer meaningful keyword.
- `se.danielj.geometridestroyer`: This application executable is a game app [15], but its description does not mention any words relevant to "game": "Remove the green objects but don't let the blue objects touch the ground". However, Macneto predicts two relevant words of "game", **game** and **libgdx** (which is a framework to develop Android game app) to describe this executable. The naïve approach predicts these two keywords as well, where the pure PCA approach does not offer any relevant words.
- `net.bierbaumer.otp_authenticator`: This application executable offers a two-factor authentication functionality to users, which users can scan QR code to log in. The description of this application is "OTP Authenticator is a two-factor **authentication**...Simply scan the **QR** code...". While the naïve approach predicts a relevant word **privacy**, Macneto infers two relevant words, **QR** and **security**, which precisely describe this app. The pure PCA approach fails to offer any relevant words.

While Macneto is able to provide at least one meaningful keyword for 14/20 obfuscated executables in the testing set, the naïve approach and the pure PCA approach can only achieve 7/20 and 4/20, respectively. Determining the

relevance of a keyword to an executable can be subjective, so we plan to conduct user studies to examining the efficacy of Macneto on the executable understanding in the future.

Result 2: Macneto has the potential to infer meaningful human words for developers/program analysts to understand an unknown executable, even it is obfuscated and has no human descriptions.

5.4 Discussions

We discuss the limitations of Macneto and the potential solutions as follows. In this paper, we have shown that Macneto can search for relevant executables, even they are obfuscated by control flow transformation and anonymization. However, while Macneto relies on instruction distributions (ICs) to search for relevant executables, adding noisy instructions that change the IC dramatically without affecting the functionality of an executable may not be handled by Macneto. If two executables have totally different functionalities, but their ICs become similar after adding some noises, Macneto will falsely detect them as similar programs. A potential solution is to leverage data flow analysis at instruction level to collect useful instructions that can influence outputs of executables to compute ICs for Macneto. Another relevant concern is that while we believe that Macneto can be a generic approach to tackle various obfuscators, we only discuss one in this paper. We plan to collect more obfuscators to conduct further experiments to prove the efficacy of Macneto in general.

While optimizing the hyper parameters, such as the layer number, of deep learning technique is out of the scope of this paper, we want to discuss the trade-off between precision and training time of Macneto. Adding more layers in Macneto can possibly enhance the search precision, but this will also increase training time. For deciding the layer number, we follow [45] to start from fewer layers (3 in this paper), which facilitate us verifying the effectiveness of Macneto in a timely fashion. In the future, after we collect more obfuscators, we believe that Macneto would definitely need more layers to classify executables.

In this paper, we use principal components as a proxy (representation) of executable behaviors. We plan to explore more techniques, such as autoencoders [10], to extract and represent executable behaviors.

6 Related Work

Although in a programming language identifier names can be arbitrary, real developers usually use meaningful names for program comprehension [29]. Allamanis et al.[3] 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 [3, 6, 13, 27, 32, 48]. Among the identifiers, a good method name is particularly helpful because they often summarize the underlying functionalities of the methods [4, 19]. Using a rule-based technique, Host et al. [19] 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 [4]. Although these two studies can suggest better method names in case of naming bugs, they do not look at the obfuscated application executables that can even change the structure of the program.

JSNice [42] and DeGuard [11] 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 may not be applied to obfuscators that modify program structure or control flow.

While Macneto uses PCA as a proxy for application behavior, a variety of other systems use input/output behavior [21, 22, 47], call graph similarity [16, 46], or dynamic symbolic execution [26, 28, 35] at method level. Macneto is 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 [8, 30, 34, 43, 49, 52]. 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 APIs and instruction mix, allowing it be an effective executable search engine.

7 Conclusion

We present Macneto, which leverages deep learning and PCA techniques at the executables level (bytecode) to search for programs (in a known corpus) similar to a given obfuscated executable. In a large scale experiment, we show that Macneto can achieve up to 84% precision to search for relevant Android executables even when the query executable is obfuscated by anonymization and control flow transformation. Compared with a naïve approach relying on instruction distribution to search for relevant executables, Macneto improves search precision by up to 29%. We also show the potential of Macneto to infer meaningful keywords for unknown executables without human descriptions.

Acknowledgements

We would like to thank all of our reviewers and our shepherd, Alvin Cheung, for their valuable comments and suggestions. This work was done while Fang-Hsiang Su was a PhD candidate at Columbia University. The Programming Systems Laboratory is funded in part by NSF CNS-1563555.

References

[1] Macneto. 2018. Macneto repository. (2018). https://github.com/Programming-Systems-Lab/macneto_release

[2] Martín Abadi, Michael Isard, and Derek G. Murray. 2017. A Computational Model for TensorFlow: An Introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017)*. ACM, New York, NY, USA, 1–7. https://doi.org/10.1145/3088525.3088527

[3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 281–293. https://doi.org/10.1145/2635868.2635883

[4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

[5] Android Open Source Project. 2018. Android Codenames, Tags, and Build Numbers. (2018). https://source.android.com/source/build-numbers

[6] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. 2016. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Software Engineering* 21, 1 (2016), 104–158.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. 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 (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[8] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 426–436. http://dl.acm.org/citation.cfm?id=2818754.2818808

[9] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 356–367. https://doi.org/10.1145/2976749.2978333

[10] Pierre Baldi. 2011. Autoencoders, Unsupervised Learning and Deep Architectures. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27 (UTLW'11)*. JMLR.org, Washington, USA, 37–50. http://dl.acm.org/citation.cfm?id=3045796.3045801

[11] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *23rd ACM Conference on Computer and Communications Security (CCS 2016)*. ACM, New York, NY, USA, 343–355. https://doi.org/10.1145/2976749.2978422

[12] Bob Pan. 2018. Dex2Jar Tools to work with android .dex and java .class files. (2018). https://github.com/pxb1988/dex2jar

[13] Simon Butler, Michel Wermelinger, and Yijun Yu. 2015. Investigating naming convention adherence in Java references. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, Washington, DC, USA, 41–50.

[14] Christian S. Collberg and Clark Thomborson. 2002. Watermarking, Tamper-proffing, and Obfuscation: Tools for Software Protection. *IEEE Trans. Softw. Eng.* 28, 8 (Aug. 2002), 735–746.

[15] Daniel "MaTachi" Jonsson. 2013. Geometri Destroyer. (2013). https://github.com/matachi/geometri-destroyer

[16] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, Reston VA, 1–15. https://www.internetsociety.org/sites/default/files/blogs-media/discovre-efficient-cross-architecture-identification-bugs-binary-code.pdf

[17] F-Droid Limited and Contributors. 2018. F-Droid. (2018). https://f-droid.org/

[18] GuardSquare. 2018. ProGuard. (2018). https://www.guardsquare.com/en/proguard

[19] Einar W. Høst and Bjarte M. Ostvold. 2009. Debugging Method Names. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Genoa)*. Springer-Verlag, Berlin, Heidelberg, 294–317. https://doi.org/10.1007/978-3-642-03013-0_14

[20] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning Deep Structured Semantic Models for Web Search Using Clickthrough Data. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management (CIKM '13)*. ACM, New York, NY, USA, 2333–2338. https://doi.org/10.1145/2505515.2505665

[21] Lingxiao Jiang and Zhendong Su. 2009. Automatic Mining of Functionally Equivalent Code Fragments via Random Testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/1572272.1572283

[22] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2010. Code Similarities Beyond Copy & Paste. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR '10)*. IEEE Computer Society, Washington, DC, USA, 78–87. https://doi.org/10.1109/CSMR.2010.33

[23] Keras. 2018. Keras: The Python Deep Learning library. (2018). https://keras.io/

[24] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014), 1–15. arXiv:1412.6980 http://arxiv.org/abs/1412.6980

[25] Vadim Kotov and Michael Wojnowicz. 2018. Towards Generic Deobfuscation of Windows API Calls. In *Workshop on Binary Analysis Research*, Vol. abs/1802.04466. Network and Distributed System Security Symposium (NDSS), San Diego CA, 1–11. arXiv:1802.04466 http://arxiv.org/abs/1802.04466

[26] D. E. Krutz and E. Shihab. 2013. CCCD: Concolic code clone detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, Piscataway NJ, 489–490. https://doi.org/10.1109/WCRE.2013.6671332

[27] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. What's in a Name? A Study of Identifiers. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC '06)*. IEEE Computer Society, Washington, DC, USA, 3–12. https://doi.org/10.1109/ICPC.2006.51

[28] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. 2016. Measuring Code Behavioral Similarity for Programming and Software Engineering Education. In *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16)*. ACM, New York, NY, USA, 501–510. https://doi.org/10.1145/2889160.2889204

[29] Ben Liblit, Andrew Begel, and Eve Sweetser. 2006. Cognitive Perspectives on the Role of Naming in Computer Programs. In *Proceedings of the 18th Workshop on the Psychology of Programming Interest Group*. University of Sussex, Brighton, UK, 53–67.

[30] Mario Linares-Vásquez, Collin Mcmillan, Denys Poshyvanyk, and Mark Grechanik. 2014. On Using Machine Learning to Automatically Classify Software Applications into Domain Categories. *Empirical Software Engineering* 19, 3 (June 2014), 582–618. https://doi.org/10.1007/s10664-012-9230-z

[31] Douglas Low. 1998. Protecting Java Code via Code Obfuscation. *Crossroads* 4, 3 (April 1998), 21–23.

[32] Robert C Martin. 2009. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, White Plains NY.

[33] Warren S. McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5, 4 (01 Dec 1943), 115–133. https://doi.org/10.1007/BF02478259

[34] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. 2012. Detecting Similar Software Applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 364–374. http://dl.acm.org/citation.cfm?id=2337223.2337267

[35] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan. 2016. Semantic Modelling of Android Malware for Effective Malware Comprehension, Detection, and Classification. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 306–317. https://doi.org/10.1145/2931037.2931043

[36] OW2 Consortium. 2017. ASM Framework. http://asm.ow2.org/index.html. (2017).

[37] K. Pearson. 1901. On Lines and Planes of Closest Fit to Systems of Points in Space. *Philos. Mag.* 2 (1901), 559–572. Issue 6.

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

[39] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. 2016. Similarity of Source Code in the Presence of Pervasive Modifications. In *16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Raleigh NC, 117–126. https://doi.org/10.1109/SCAM.2016.13

[40] Anand Rajaraman and Jeffrey David Ullman. 2011. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA.

[41] V. Rastogi, Y. Chen, and X. Jiang. 2014. Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Transactions on Information Forensics and Security* 9, 1 (2014), 99–108.

[42] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

[43] David Schuler, Valentin Dallmeier, and Christian Lindig. 2007. A Dynamic Birthmark for Java. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 274–283. https://doi.org/10.1145/1321631.1321672

[44] Smardec Inc. 2018. Allatori Java Obfuscator. (2018). http://www.allatori.com/

[45] Richard Socher, Milind Ganjoo, Christopher D. Manning, and Andrew Y. Ng. 2013. Zero-shot Learning Through Cross-modal Transfer. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'13)*. Curran Associates Inc., USA, 935–943. http://dl.acm.org/citation.cfm?id=2999611.2999716

[46] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code Relatives: Detecting Similarly Behaving Software. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 702–714. https://doi.org/10.1145/2950290.2950321

[47] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. 2016. Identifying Functionally Similar Code in Complex Codebases. In *24th IEEE International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, Washington, DC, USA, 1–10. http://dx.doi.org/10.1109/ICPC.2016.7503720

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

[49] Haruaki Tamada, Masahide Nakamura, and Akito Monden. 2004. Design and Evaluation of Birthmarks for Detecting Theft of Java Programs. In *Proceedings of the IASTED International Conference on Software Engineering*. IASTED, Calgary Canada, 569–575.

[50] TensorFlow. 2018. TensorFlow An open-source machine learning framework for everyone. (2018). https://www.tensorflow.org/

[51] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 674–691. https://doi.org/10.1109/SP.2015.47

[52] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. 2015. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 303–313. http://dl.acm.org/citation.cfm?id=2818754.2818793