Robust Detection of Obfuscated Strings in Android Apps

Alireza Mohammadinodooshan
Ulf Kargén
Nahid Shahmehri
alireza.mohammadinodooshan@liu.se
ulf.kargen@liu.se
nahid.shahmehri@liu.se
Department of Computer and Information Science, Linköping University
Linköping, Sweden

ABSTRACT

While string obfuscation is a common technique used by mobile developers to prevent reverse engineering of their apps, malware authors also often employ it to, for example, avoid detection by signature-based antivirus products. For this reason, robust techniques for detecting obfuscated strings in apps are an important step towards more effective means of combating obfuscated malware. In this paper, we discuss and empirically characterize four significant limitations of existing machine-learning approaches to string obfuscation detection, and propose a novel method to address these limitations. The key insight of our method is that *discriminative* classification methods, which try to fit a decision boundary based on a set of positive and negative samples, are inherently bound to generalize poorly when used for string obfuscation detection. Since many different string obfuscation techniques exist, both in the form of commercial tools and as custom implementations, it is close to impossible to construct a training set that is representative of all possible obfuscations. We instead propose a *generative* approach based on the Naive Bayes method. We first model the distribution of natural-language strings, using a large corpus of strings from 235 languages, and then base our classification on a measure of the *confidence* with which a language can be assigned to a string. Crucially, this allows us to *completely eliminate* the need for obfuscated training samples. In our experiments, this new method significantly outperformed both an n-gram based random forest classifier and an entropy-based classifier, in terms of accuracy and generalizability.

CCS CONCEPTS

• **Computing methodologies** → **Supervised learning by classification**; **Machine learning approaches**; • **Security and privacy** → *Malware and its mitigation.*

KEYWORDS

Android, string obfuscation detection, string encryption, machine learning, generative models, malware

ACM Reference Format:
Alireza Mohammadinodooshan, Ulf Kargén, and Nahid Shahmehri. 2019. Robust Detection of Obfuscated Strings in Android Apps. In *12th ACM Workshop on Artificial Intelligence and Security (AISec'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338501.3357373

AISec'19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6833-9/19/11…$15.00
https://doi.org/10.1145/3338501.3357373

1 INTRODUCTION

For users of mobile devices, the threat of malware is becoming increasingly prevalent, with millions of new mobile malware samples detected each year. The vast majority of mobile-malware infections happen through the use of trojanized apps, where unsuspecting victims download and install malware posing as legitimate applications on an app store. Due to the relatively unrestricted nature of the Google Play Store, where apps can be added without a manual vetting process, and also due to the availability of many third-party Android app stores, the majority of malware infections affect systems running Android.

To combat the ever-growing threat of mobile malware, many anti-malware providers offer commercial solutions to detect mobile malware. Mobile malware detection is also a very active field of research in academia, where both static [15, 21, 31] and dynamic methods [10, 35] have been proposed to detect malicious apps. Dynamic methods execute suspect apps in a sandbox environment in an effort to detect any malicious behavior. While such methods can be highly effective, they require significant computational resources when used at scale. Moreover, some malware is capable of subverting dynamic analysis by, e.g., fingerprinting the execution environment and refraining from displaying malicious behavior when executed inside a sandbox. Static methods, on the other hand, offer a cost-effective alternative by quickly detecting malware based solely on features that can be statically extracted from apps. As a countermeasure to static detection methods, however, many malware authors employ various obfuscation techniques. In a recent study by Dong et al. [14], *string encryption* stands out as an obfuscation technique that is significantly more common in malware than in benign apps. There are several reasons for the popularity of this obfuscation technique in malware. Firstly, string analysis often offers important clues to malware analysts about the inner workings of a malicious app [13, 20]. Secondly, string obfuscation is an effective means for preventing detection by common anti-malware

products. For example, in an experiment by Zheng et al. [36], string obfuscation reduced the average detection ratio of popular anti-malware engines from 94% to 51%. Several other studies have also pointed to the effectiveness of string obfuscation in circumventing malware scanners [17, 24, 27, 29, 30]. In light of these results, there is clearly a need for effective methods to detect string obfuscation in mobile apps. Such methods could, for example, be used as a pre-filter for classification methods, allowing an anti-malware engine to choose the best analysis approach based on whether obfuscation was present or not, or they could aid in feature prioritization and filtering in malware classification. An accurate method for classifying strings as obfuscated or non-obfuscated could also aid in manual analysis of malicious apps, for example by helping to quickly find all of the parts of an app's code that access obfuscated strings.

In a string-encrypted app, string material is stored in encrypted or scrambled form within the application, and additional logic is inserted into the program code of the app to dynamically reconstruct strings from the encrypted material on-the-fly. Several techniques for detecting string encryption in apps have been proposed in the literature. One approach to the problem, used by two recent works [23, 33], is to detect the presence of the deobfuscation logic that is spliced into the code of a string-encrypted app. In order to be effective, such obfuscation detectors must have up-to-date signatures to detect the specific logic added by different obfuscation tools. Such signatures can either be manually crafted, or they can be learnt using a machine learning algorithm. A significant limitation of this approach is that any updates to the obfuscation tool could potentially invalidate existing signatures. Since several obfuscation tools also have many configurable options, collecting signatures that cover all possible configurations could be challenging. Moreover, custom string-encryption implementations cannot be detected at all with this approach.

Instead of attempting to detect artifacts of specific obfuscators, a more general approach is to directly detect the presence of obfuscated strings in an app. There are two existing approaches to solving this problem. The first is to classify strings based on some general indicators of "randomness", such as the Shannon entropy. The other approach is to train a machine-learning model to identify differences between obfuscated and non-obfuscated strings by supplying both regular strings (i.e., negative samples) and their obfuscated counterparts (i.e., positive samples) as training data for the model. The recent AndrODet system by Mirzaei et al. [25] use the former approach, whereas the work by Dong et al. [14] utilizes the latter. While more general than signature-based methods, both of these existing works have several limitations that reduce their effectiveness. First, both methods can only classify entire apps as obfuscated, and not individual strings (*Limitation 1*). This is a significant limitation, as any method that performs classification based on the frequency or prevalence of certain traits or features of strings in the *entire* app (for example, the average string entropy) will be highly sensitive to the *ratio* of obfuscated strings to all strings in the app. While studying the recently published AMD malware dataset [34], we observed that many kinds of Android malware contain both encrypted and unencrypted strings, and that the encryption ratio differs significantly between different malicious apps. Therefore, in practice, robust detection of string obfuscation in apps cannot be based on measuring compound properties of all app string material. Instead, each string must be classified individually. Final triaging of an app as obfuscated or not can then be performed based on, for example, a threshold on either the percentage or the absolute number of obfuscated strings. By necessity, however, such a threshold must be selected on a case-by-case basis, since a universally valid threshold simply cannot be determined. Moreover, a general obfuscation classifier that works on individual strings could have a wider range of application, for example in detecting network data exfiltration [18, 22, 28] or social media bots [11, 12, 16]. The second main limitation of existing methods that we have identified concerns the methods' ability to generalize. The first source of limited generalizability that we have identified stems from using strings from a collection of real apps as training data for an obfuscation detector. Since such training material is typically highly biased towards a small number of common natural languages, the trained model will likely perform poorly on strings from less common languages (*Limitation 2a*). For example, the F-Droid [7] app repository used by Dong et al. for training their model only contains apps using a handful of different natural languages. Therefore, if a non-obfuscated app uses a less common language, it may be more likely to be misclassified as obfuscated. A more fundamental limitation of using an off-the-shelf machine learning method for string obfuscation classification is the inherent challenge in finding representative training samples of obfuscated strings (*Limitation 2b*). Obfuscation methods can range from using full-fledged strong cryptography (e.g., AES, RC4, etc.) to simple scrambling methods, such as rotation ciphers or encoding schemes (e.g., Base64 encoding). A truly general obfuscation classifier must be able to distinguish a human-generated string[1] from *any* other type of string. This introduces a significant asymmetry-problem in training-data selection, as the space of all possible strings is enormous in comparison to the space of human-generated strings. We found in our experiments that when training an off-the shelf classifier using different types of "random-looking" strings as positive (i.e., obfuscated) training samples, the final model invariably tended to overfit to characteristics of the particular type of training samples used, and therefore did not generalize well to different kinds of obfuscation methods.

The approach used in the AndrODet system partially avoids Limitations 2a and 2b by using features specifically selected to capture the randomness of strings, namely the Shannon entropy, in addition to several features that are relevant for specific encoding schemes, such as the frequency of certain special characters. While the encoding-specific features do not generalize to all types of string obfuscation, the entropy is a general measure of randomness. However, we found in our experiments that the entropy is not a sufficiently informative feature for classifying strings with high accuracy, especially when strings are fairly short (*Limitation 3*). Moreover, some obfuscation methods, such as simple rotation or substitution ciphers, do not increase the entropy of a string at all.

To address these limitations, in this paper we propose a novel method for determining whether or not a string is obfuscated. Our goal has been to design a method that can classify individual strings with good accuracy, even when strings are fairly short (Limitation

[1] Here, the term "human-generated strings", denotes natural-language strings, or strings that closely resemble natural language, such as, e.g., file names, identifiers in source code, or URLs.

1). In light of Limitation 3, we opted to use string n-grams as our principal features and train a machine-learning model to classify strings, similar to the work by Dong et al. To avoid Limitation 2a (language bias), instead of training directly on strings extracted from apps, we train our model on the WiLI dataset [32], which contains natural-language strings from 235 languages. The main novelty of our method, however, lies in the way we avoid Limitation 2b (overfitting to positive training samples) when employing machine learning. Most popular classification methods are *discriminative*, i.e., given a set of samples of each class, they attempt to find classification boundaries in feature space that separate classes optimally. Such methods would suffer significantly from Limitation 2b, for the reasons discussed above. A *generative* classification approach instead attempts to learn the distribution of samples in each class, and base its classification on this. Instead of relying on a discriminative classifier, which needs a representative set of positive and negative samples, we propose a generative approach that *entirely* avoids using positive (i.e., obfuscated) samples. For each language in the WiLI dataset, we train a separate Naive Bayes model to compute the likelihood of a string belonging to that language. Obfuscation classification is done using a statistical distance measure based on the *confidence* with which a unique natural language can be assigned to a string. The intuition behind our approach is that, for natural-language strings (or at least for strings containing fragments of natural language) we will be able to assign a language to the string with high confidence, whereas for obfuscated strings the likelihood distribution is more uniform across all 235 languages, as the string will not closely resemble any language.

In summary, the main contributions of our work are as follows:

- We have identified four main limitations of existing string obfuscation classification methods:
  - **Limitation 1**: A string obfuscation detector must work on individual strings rather than on entire apps, as the fraction of strings that are obfuscated cannot be assumed to be the same for all apps.
  - **Limitation 2a**: If an obfuscation classifier is trained to identify differences between obfuscated and non-obfuscated strings, it is important for training data to be representative of as many natural languages as possible, in order to avoid poor performance on less common languages.
  - **Limitation 2b**: A discriminative classifier that is trained to identify obfuscated strings is fundamentally limited by the lack of representative training data for all possible kinds of obfuscation, and will generalize poorly.
  - **Limitation 3**: While entropy is a generalizable measure of string randomness, it does not have sufficient discriminative power to allow high-accuracy obfuscation classification of strings.
- We perform a thorough empirical study of the impact of Limitations 2b and 3 on string obfuscation detection.
- We propose a novel approach to detecting obfuscated strings, and show that it has superior accuracy and generalizability, compared to both a classical discriminative classification approach, and to an entropy-based classifier.

While we have focused our discussion in this work on Android string obfuscation detection, our results, and the proposed new obfuscation-detection method, are general enough to also be of interest for string obfuscation detection in other domains.

The remainder of the paper is structured in the following way: In Section 2 we provide some background on string obfuscation techniques and describe the aforementioned string obfuscation detection techniques by Dong et al. and Mirzaei et al. Section 3 describes the design of our string obfuscation detection method. In Section 4 we begin with an empirical demonstration of the impact of the limitations we describe above, and then we present the evaluation results for our method. In Section 5, we discuss our results and outline directions for future work. We discuss related work in Section 6, and finally conclude the paper in Section 7.

2 BACKGROUND

In this section, we first provide some background on Android string obfuscation techniques, and then we discuss the existing approaches by Mirzaei et al. [25] and Dong et al. [14] for detecting the use of string obfuscation in apps.

2.1 Android String Obfuscation

String obfuscation (also known as string encryption) is a simple way to complicate reverse-engineering of apps by denying an analyst the semantic insight provided by constant strings embedded in the app. It may also be combined with Java reflection to hide API calls from static analysis tools, by obfuscating the names of API methods and classes that are invoked reflectively. Due to its effectiveness in slowing down reverse-engineering, it has become a popular way to prevent intellectual property theft among app developers. Consequently, most commercial app obfuscation tools support string obfuscation. However, since string obfuscation is also a simple yet effective way to transform a malicious app to evade signature-based malware detection or complicate manual analysis, string obfuscation is also frequently employed in malware.

String obfuscators work by removing constant strings from an app, and instead storing them in encrypted form within the app. Special decryption routines are inserted in the code, so that strings are decrypted just prior to their use. In addition to commercial obfuscators, many malware authors use custom string encryption implementations, such as simple substitution ciphers (e.g., rotation ciphers) [34].

Android apps are shipped in the form of Android application packages (APKs), which in turn contain one or more DEX files. A DEX file holds metadata about all classes and methods (such as their name and signature), in addition to the compiled Dalvik bytecode of all methods contained in the DEX file. Each DEX file also has a *string section*, holding all constant strings used by the contained classes. Most obfuscation tools store encrypted strings in this string section. Examples of popular obfuscators using this approach are DashO [4], DexProtector [6], and Allatori [1]. A notable exception is the DexGuard [5] tool, which instead stores encrypted string material as static byte-arrays. Obfuscators that store encrypted strings in the string section must, however, make sure that all strings are still valid according to the Modified UTF-8 format stipulated by the DEX file format specification [3]. In particular, strings must never contain unencoded zero-bytes, as a single byte with value zero is used as a string delimiter in DEX files.

2.2 Existing Approaches to Detecting String Obfuscation

Here, we describe in more detail the works on string obfuscation classification by Dong et al. and Mirzaei et al. To the best our knowledge, these are the only works that specifically address the problem of detecting string obfuscation in apps by analyzing extracted strings. Even though the focus of our work has been different from theirs, in that we aim to classify *individual strings* as obfuscated, it is still important to understand the design of these earlier methods, as they highlight some of the limitations of existing string-obfuscation detection techniques.

Dong et al. [14] used a support vector machine (SVM) model to detect string obfuscation, with the goal of characterizing the prevalence of different obfuscation techniques among both benign and malicious apps. To obtain a set of obfuscated apps for training, they applied the DashO and DexProtector obfuscation tools to apps from the F-Droid [7] dataset. For each app (and its obfuscated counterparts) a feature vector was created, containing the 3-gram frequencies for the entire set of strings in the app. An SVM model was finally trained on a random sample of 500 original and 500 obfuscated apps. When evaluated on 100 apps of each class (obfuscated and original), which had been excluded from the training set, their method could classify apps as obfuscated and unobfuscated with 98.5% accuracy, according to the paper. The authors did not indicate if the obfuscation tools were configured to encrypt all strings in an app, or if only a subset of strings were encrypted. The documentation of most tools recommends the latter approach, as encrypting all of the strings in an app often leads to excessive runtime overhead. If the classifier was trained on apps in which all strings were encrypted, it is highly likely that its accuracy would be reduced if used to classify in-the-wild apps, since these would contain a mix of encrypted and unencrypted strings (Limitation 1). Furthermore, if all strings in the obfuscated training set are encrypted, a potential concern is that a machine learning algorithm could be prone to memorizing common string fragments that exist in almost all apps, such as mandatory parts of URLs or common file-name endings, and base classification on the presence of such string fragments in an app. Such a model would obviously fail to generalize to apps where not all strings are encrypted.

Mirzaei et al. present the AndrODet system [25] for detecting different types of obfuscation in Android malware. Their system is based on on-line learning. In contrast to the more common batch-learning approach, in which a model is trained on a fixed set of samples, on-line training instead works on a stream of samples, where the model is continuously updated as new samples arrive. They, however, also show results of their approach with batch-learning for comparison. In our setting, the batch-learning results are more interesting to discuss, as on-line learning often yields less accurate models. Also, there is not a clear use case for on-line learning in our context. For on-line learning, Mirzaei et al. used the Leveraging Bagging machine-learning method with decision trees as the base classifier. For the batch-learning experiments, they used an SVM model, similar to Dong et al. In contrast to Dong et al., however, they use features specifically selected to capture the average "randomness" of all strings in an app. Specifically, they compute the average per-string entropy, average string length and word size[2], average number of several special characters (equals, dashes, slashes, pluses), and the average number of repeated characters. It can be noted that all these features except the entropy are based on assumptions about specific obfuscation implementations. (For example, many Base64 implementations use the equals sign as a padding character.) The accuracy of AndrODet was evaluated on the AMD [33] and PraGuard [24] malware datasets. For on-line learning, a balanced subset of the AMD dataset was used[3], so that there were an equal number of apps with and without string obfuscation. The average accuracy for on-line learning was reported to be 81.4%. For batch-learning, the SVM model was trained on the AMD dataset and evaluated on the PraGuard dataset, with a reported accuracy of 81.2%.

AndrODet shares Limitation 1 with the work by Dong et al., as their model implicitly learns the ratio of obfuscated to non-obfuscated strings in the particular dataset used for training. Also, the only generalizable feature used in AndrODet is the entropy, which has limited discriminative power, as we will show in Section 4.1 (Limitation 3).

3 METHOD DESIGN

The key insight, on which the design of our method is based, is that accurately modeling the apparent "randomness" of obfuscated strings in a generalizable way is an extremely difficult problem. Both of the approaches described in the previous section suffer fundamentally from that same limitation: The AndrODet system uses features specifically engineered to capture the "randomness" of strings. However, crafting a set of features that can accurately separate *any* obfuscated string from *any* natural-language string is notoriously hard. In practice, such attempts will be based on observations of existing obfuscation schemes, and therefore suffer from potential bias. With the approach taken by Dong et al., rather than using pre-engineered features, an off-the-shelf machine learning method is used to try to learn the differences between natural-language and obfuscated text, using some "raw" text features. However, when selecting positive (i.e. obfuscated) training samples, one is faced with the problem depicted in Figure 1. Since the set of natural-language strings is extremely small in comparison to all possible obfuscated strings, it is difficult to compile a corpus of positive samples that captures characteristics of all possible obfuscations. One possibility would be to create the positive samples with a limited selection of encoding/obfuscation schemes. As depicted in Figure 1a, this may lead to, e.g., an obfuscated string being misclassified as non-obfuscated because it has been created with a scheme that was not represented in the training set. If instead one attempts to create a maximally general set of positive samples by creating strings using random character sampling, then one is faced with the futility of trying to cover the entire space of obfuscated strings, as depicted in Figure 1b.

Taking the aforementioned problems with existing approaches into consideration, we have proposed a new method for classifying

[2]What the authors refer to as the *word size* is actually, according to their released source code, the size in memory of a Python string object, which depends both on the total string length and the Unicode character set of the string, but also on the underlying OS and the particular Python implementation.
[3]It is not specified in the paper how this subset was selected.

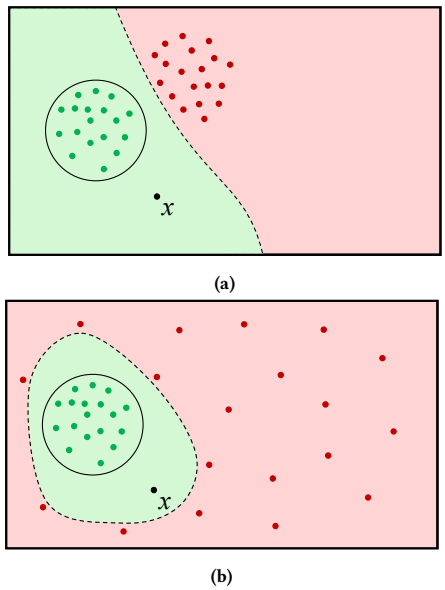

Figure 1: Illustration of the challenges of training a discriminative classifier to separate obfuscated strings from non-obfuscated. The rectangle represents the space of all possible strings, while the circle represents human-generated (i.e., non-obfuscated) strings. The green dots within the circle represent negative samples, while the red dots outside the circle represent positive samples. When attempting to find a decision boundary that closely approximates the circle, one is faced with either (a) overfitting to a particular type of obfuscation, or (b) a lack of representative samples of *all* possible kinds of obfuscation. In both cases the obfuscated string x will be misclassified.

a string as either (closely resembling) natural language, or non-natural language. Due to the challenges of using specific features (such as entropy, cf. Limitation 3) to capture "randomness", we opted instead for using an n-gram based model, similar to Dong et al. However, to avoid biasing the classifier towards the limited number of natural languages present in a particular app dataset (Limitation 2a), we instead train our model on the WiLI [32] dataset, consisting of two sets of 500 strings each for 235 languages, sampled from Wikipedia. The dataset has been created with language-identification applications in mind, where the first set of strings is intended for model training and the second set for testing. An additional benefit of using a standalone all-natural language dataset is that we avoid potential biasing towards, or memorizing of, strings from, e.g., certain popular app libraries.

The most challenging hurdle to overcome is Limitation 2b, i.e., to create a classifier that generalizes, to the largest degree possible, to *any* obfuscation scheme. We address this challenge by using a generative classification approach rather than a discriminative one. Using the WiLI dataset, we construct a statistical model for each of the 235 languages, which can be used to estimate the *likelihood distribution* of languages given a particular string. When classifying an individual string as obfuscated or not, we apply each model to the string and rank all the likelihoods. The maximum likelihood can then be used to compute a statistical *distance measure*. A distance threshold can be defined in a second training phase by computing this distance for a collection of natural language strings, and selecting a threshold to achieve, for example, a certain minimum accuracy or a maximum false positive rate. The main benefit of this approach, which is also the key to overcoming Limitation 2b, is that we *completely eliminate* the need for obfuscated training samples. Instead of determining a decision boundary based on a set of positive and negative samples, cf. Figure 1b, we approximate the optimal decision boundary (represented by the circle in the figure) by learning the distribution of natural-language strings. Next, we describe our method in more detail.

3.1 Model Construction

Our approach to detecting string obfuscation is based on the Naive Bayes method. We have chosen this method for its simplicity and scalability. Due to the simplicity of the method we achieve a high degree of resilience to overfitting and memorizing, which may be a problem for more complex machine learning methods. For each language in the first string set of WiLI, we extract all n-grams up to a specified n from the strings of that language and record their relative frequency in a table. This frequency table is used to look up the estimated likelihood $P(w|L)$ of encountering n-gram w while scanning through a text written in language L. To compute the likelihood $P(L|x)$ that a particular string x belongs to language L, we can extract all n-grams $w_1^x \ldots w_m^x$ from x and compute the compound probability

$$P(L|x) = \prod_{i=1}^{m} P(w_i^x|L)$$

However, to avoid precision loss during computation due to fixed-width floating point arithmetic, in practice we compute the log-likelihood

$$\log P(L|x) = \sum_{i=1}^{m} \log P(w_i^x|L)$$

For n-grams in x that are not present in the training data for L (and therefore do not have an entry in the frequency table), we use the lowest n-gram frequency of that language: $\arg\min_w P(w|L)$.

The aforementioned distance measure, which describes how close a string is to any natural language (present in our training set), is computed as the relative (log) likelihood of the most likely language $P_{\max}(x) = \arg\max_L P(L|x)$ and the "average" language. We call this quantity the *confidence* $C(x)$. We compute $C(x)$ in the following way:

$$C(x) = \log P_{\max}(x) - \frac{\sum_{i=1}^{N} \log P(L_i|x)}{N} \quad (1)$$

where N is the total number of languages. Note that

$$C(x) = \log P_{\max}(x) - \log \sqrt[N]{\prod_{i=1}^{N} P(L_i|x)} = \log \frac{P_{\max}(x)}{\sqrt[N]{\prod_{i=1}^{N} P(L_i|x)}}$$

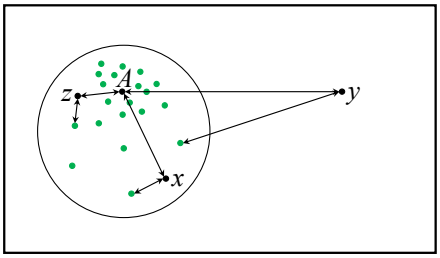

Figure 2: An illustration of the intuition behind our statistical distance measure, which is based on the relative similarity to the closest language compared to the "average" language. Non-obfuscated samples (x, z) are significantly closer to some natural language (a green dot) than to the "centroid" A of all languages. For an obfuscated sample (y), there is a smaller difference between the distance to A and the distance to the closest language.

where the denominator $\sqrt[N]{\prod_{i=1}^{N} P(L_i|x)}$ is the *geometric mean* of the likelihoods for all languages, i.e., $C(x)$ is a measure of how much more likely it is that x belongs to the best-matching language than to the "average" language. The intuition behind this definition is that, for most natural-language strings, the likelihood of the best match $P_{\max}$ is significantly higher than the geometric mean, yielding a high C, while for an obfuscated (and therefore more "random looking") string, the distribution of $P(L_1|x)...P(L_N|x)$ will be closer to uniform, therefore yielding a lower C. A geometric analogy is shown in Figure 2. The circle again denotes the actual boundary between natural-language strings and all other strings, while the green dots inside the circle denote the "centroids" of the languages in our training set. The point labeled A represents the centroid of all languages, i.e., the "average" language. Consider the non-obfuscated sample denoted x. The distance between x and the "closest" language is much shorter than the distance from x to A. Hence, the confidence $C(x)$ is high. The obfuscated sample y, however, is about as far away from its closest language as it is to A, yielding a low $C(y)$.

Initially, we considered simply basing classification on a threshold on $P_{\max}$, but found the approach shown above to work better. One reason for this may be that quantifying a value of $P_{\max}$ that is "close enough", without using obfuscated samples for comparison, is quite hard. Many non-obfuscated strings may still be relatively dissimilar from any natural language string. However, even in cases where a string is not very similar to any language, if it is considerably *more* similar to one language, this is a good indication that the string is not obfuscated.

For the final classification of a string, we must select a threshold for how distant the string is allowed to be from the natural languages without being considered obfuscated. However, we must take into account that the level of confidence in assigning a language to a string will generally be higher for longer strings. Also, strings of some languages are inherently more similar to each other than to other languages. For example, strings from languages based on the Latin alphabet are relatively much more similar to each other than to languages using a different alphabet. Returning to our geometric analogy in Figure 2, this can be understood as some languages being closer to the centroid A, i.e., languages are not uniformly distributed around the "average" language. For example, the ratio between the distance to A and the closet language is much smaller for z than for x. Therefore, we cannot simply use a global threshold on this ratio to determine if a string is sufficiently close to the natural languages. To address this problem, we compute C for each distinct language and string length[4], using the second string set in WiLI[5]. We generate strings of all lengths by "chopping" the original WiLI strings at the word boundaries, to generate non-overlapping substrings. Based on this, we can, for each language/string-length combination, select a threshold. Currently we use the 5th percentile as threshold, i.e. a value C_T such that 95% of all strings of the given length and language have $C(x) > C_T$. When classifying a string, we determine the language with the highest likelihood ($P_{\max}$), and lookup C_T for that language, for the given string length. This method of computing the threshold has the benefit of both having an intuitive interpretation, and being easily tunable to achieve a desired trade-off between precision and recall. Note that, when computing the table of thresholds, we always base our computation on $P_{\max}$, as shown in Equation (1). That is, even in cases where the most likely language does not match the actual language of the string, we still use $P_{\max}$ to compute C. The rationale for this is that, in cases where several languages are roughly equally good matches, the likelihood of the true language is about the same as $P_{\max}$ anyway.

3.2 String Preprocessing and Feature Extraction

All strings, both for model construction as well as for classification, are preprocessed by removing all punctuation and delimiters, and replacing them with a space. This step is necessary since the use of punctuation differs significantly between languages. For example, words are delimited using a space in English, while in, e.g., Chinese and Japanese it is common to use delimiters that are not whitespace. Delimiters are removed by scanning through each string and replacing all characters that are not in the "letter", "mark", or "number" categories of the Unicode specification [9] with a space. Any instances of consecutive whitespace are then collapsed into a single space. Furthermore, all characters are converted to their lowercase equivalent, when applicable.

After preprocessing, we extract all character n-grams from the string. We have chosen to use n-grams up to size 3. Strings are split into words (i.e. sequences of characters separated by a space), and all 1-, 2-, and 3-grams of each word are extracted. (That is, n-gram extraction respects word boundaries, so that n-grams never span several words.) We only consider strings that have length 3 or more after preprocessing.

3.3 Implementation

Our method is implemented in Python using the `CountVectorizer` class of scikit-learn 0.20.3. String preprocessing is done using the

[4]We consider string lengths up to 80 characters. For strings longer than this, we use the values computed for a length of 80.
[5]That is, the "testing" string set of WiLI.

built-in UTF support in Python. The built-in Python function `casefold` is used for converting strings to lowercase.

4 EXPERIMENTAL RESULTS

In this section, we first present empirical results to highlight the limitations of discriminative machine learning methods for detecting obfuscated strings. We then show how our generative classification method performs in the same experimental setting. In all of our experiments, the multi-language WiLI dataset was used for training. For testing, we first extracted all strings (about 4.3 million) from the 1830 apps in the F-Droid [7] repository of open-source apps, and then randomly sampled 400,000 strings to create our testing dataset.

4.1 Discriminative Methods

An n-gram based classifier. In our first experiment, we evaluated the performance of an n-gram based classifier using the random forest implementation in scikit-learn. We chose random forest because of its versatility and ability to find complex non-linear patterns in data. Because the number of features (i.e., n-grams) is very high, we set the depth of trees to 1000. The number of trees was left at the default setting of 10. We used the same preprocessing and feature extraction procedure as described in Section 3.2. Strings extracted from the apps in the F-Droid repository were converted from the Android Modified UTF-8 format to standard UTF-8 before preprocessing.

For each non-obfuscated string in the training and testing sets, we randomly generated four synthetic strings of the same length, each intended to be representative of a distinct type of string obfuscation found in malware. We chose to use synthetic strings rather than extracting real obfuscated strings from apps for two reasons. First, since our goal has been to devise a maximally general string obfuscation detector, the main objective of our experiments has been to evaluate our method's ability to generalize to different kinds of string obfuscation. With synthetic strings, we are certain to have a set of sufficiently diverse classes of "obfuscated" strings with known properties, which would not be the case if we were to use a collection of closed-source obfuscation tools to generate strings. The second reason for using synthetic strings is to have accurate ground truth for our tests. Since, in many cases, not all strings in an obfuscated app are actually encrypted, we would need to manually check all strings that had been extracted from real obfuscated apps. Our four types of synthetic "obfuscated" strings are described below:

- **Base64** (B64). Base64 encoding of strings is commonly used in, e.g., obfuscated malware [34]. An encoding scheme such as Base64 can be used both as an obfuscation scheme in itself, or to encode strings encrypted using, e.g., standard block ciphers or XOR encryption. (Note that strings subjected to an off-the-shelf standard encryption method must always be encoded somehow, to ensure that they contain only valid UTF-8 characters.) Synthetic strings were generated by randomly sampling from the typical Base64 range of ASCII characters (a–z, 0–9). Note that we randomly generate *Base64-like* strings, rather than applying actual Base64 encoding to the corresponding non-obfuscated string. This is to make sure that the lengths of a non-obfuscated string and its "obfuscated" counterpart are always the same. Otherwise, we might inadvertently end up training a model to classify strings based on their length rather than their content.
- **Rotation** (ROT). Various types of string permutation methods are also commonly used for obfuscation in malware [34]. Rotation ciphers constitute a simple class of permutation ciphers that is interesting to study, as it does not add any entropy to the original string. To produce synthetic strings of this type, the original string is transformed by adding a small constant to each Unicode code point.
- **Fully uniform random sampling** (FU). Strings are generated by random sampling (with replacement) from the entire Unicode range. This scheme is intended to be representative of string encryption methods used by commercial obfuscation tools. Based on examples given in documentation [2, 8], as well as the results of independent investigation [19, 26], many tools appear to generate strings from the entire range (or at least a large range) of Unicode characters.
- **Language-uniform random sampling** (LU). This is a variant of the FU scheme, which addresses the shortcoming of fully random sampling that "obfuscated" strings will be biased towards languages with a large number of letters (such as, for example, Chinese). For each language in the WiLI dataset, we first compile the set of all characters that are present at least once in that language. When generating random strings, we first pick a random language in the WiLI dataset, and then randomly pick a letter from that language with uniform probability.

The results of our experiment are shown in Figure 3a. The labels on the rows of the matrix denote the type of strings used as positive (obfuscated) strings during training, and the labels of the columns denote the type of strings used for testing. Each cell of the matrix shows the accuracy as a percentage as well as the F-score, for the respective training/testing combination. In addition, each cell is color-coded based on the accuracy. The values are averages over five training/testing runs with randomly generated synthetic strings[6]. As can be seen from the figure, the classification accuracy tends to be significantly better when the same types of "obfuscated" strings were used both for training and testing (cf. Figure 1a). The training bias problem is most evident for the B64 case, where almost all of the more high-entropy strings in the LU and FU categories are misclassified as non-obfuscated, as indicated by the F-score of 0. The classifier trained on fully random strings (FU) appears to generalize the best. However, the accuracy for the B64 and ROT cases is much lower than for FU and LU test strings (cf. Figure 1b).

We also tried "forcing" the classifier to focus on features of the non-obfuscated strings (in a similar vein to our method), by limiting the feature set to only the n-grams that were present in the natural-language strings. The results are shown in Figure 3b. Here, we see that the classifiers trained on ROT strings perform significantly better. Interestingly, the ROT classifier performs better on FU strings than on ROT strings. The LU and FU classifiers also appear

[6]For the ROT case, we used the offsets 1–5 for the respective runs, rather than random offsets. The rationale for this was to preserve the same character set as the original language. For testing we used the training offset "rotated" 3 steps (i.e., training offset $+2 \mod 5 + 1$).

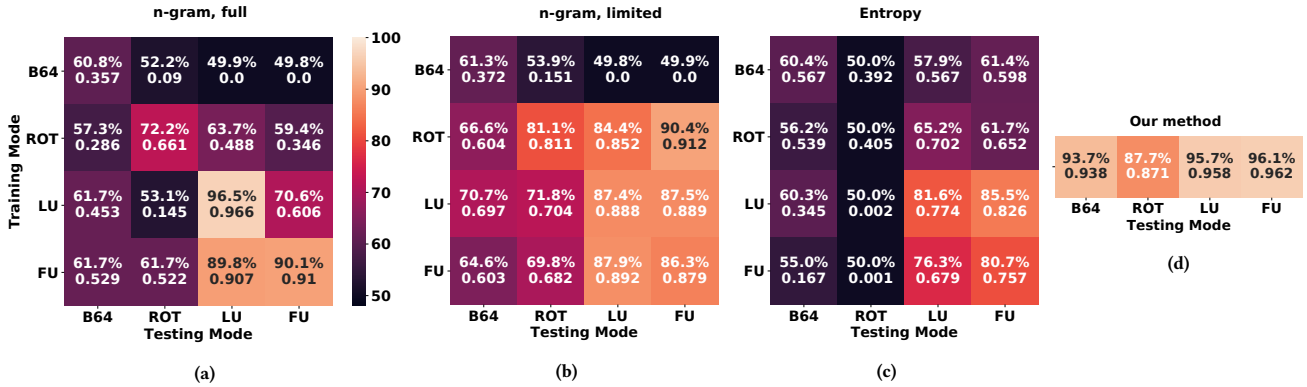

Figure 3: Accuracy and F-score for different training (rows) and testing (columns) sets. The matrices show results for (a) an n-gram based discriminative classifier, (b) the same classifier with the feature set limited to just n-grams from the WiLI dataset, (c) an entropy-based discriminative classifier, and (d) our generative classifier.

to generalize a bit better, however, at the cost of lower accuracy on the LU and FU test cases.

In summary, the n-gram based discriminative classifiers in our experiments showed a significant tendency to overfit to the positive samples used during training, and did not generalize well to detect different types of obfuscated strings, i.e., they suffer from Limitation 2b. This problem persisted even when we limited the feature set to force the learning algorithm to focus on traits of the negative (non-obfuscated) samples.

An entropy-based classifier. In our second experiment, we evaluated the use of entropy-based classification, using the same training and testing data as in the previous experiment. We also used the same machine learning method and hyperparameters as before (a forest of 10 trees with maximum depth 1000). Since the length of a string will affect the value of the Shannon entropy, we also included the string length as a feature, in addition to the byte-level entropy of strings. The results are presented in Figure 3c When trained on B64 strings, the entropy-based classifier shows a better balance between false positives and false negatives than the n-gram based classifier, but at a similarly low level of accuracy. The classifiers trained on high-entropy samples perform relatively well on the high-entropy test cases (FU and LU), albeit less well than the n-gram based classifier, despite completely random strings being the theoretically optimal case for an entropy-based classifier. As expected, all classifiers here fail completely on the ROT test cases, with an accuracy of 50% (i.e., no better than random guessing). This is because a rotation cipher will not affect the entropy of a string. The classifiers trained on high-entropy positive samples also perform very poorly on the lower-entropy B64 test cases, where many of the "obfuscated" strings are misclassified as non-obfuscated, resulting in a low F-score.

To further investigate the discriminative power of entropy for string obfuscation classification, we plotted the sample density as a function of entropy, for the different types of strings in our test set. The ROT case is left out, since rotation-encrypted strings have identical entropy to the original string. To make densities comparable, we chose strings of a fixed length, specifically the median string length, which was 18 for the strings from the F-Droid dataset. The

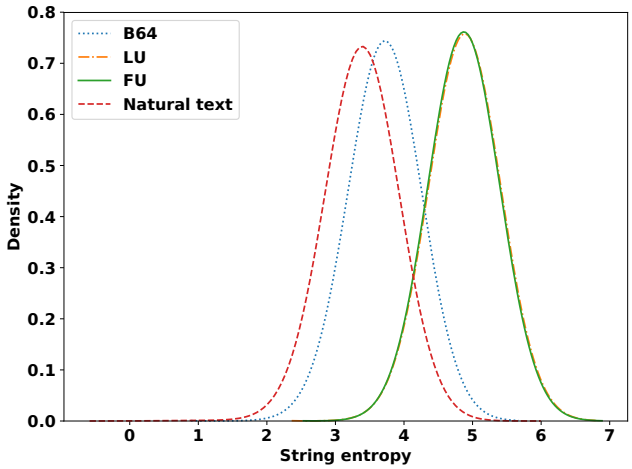

Figure 4: Smoothed sample density as a function of entropy for strings with length 18.

plot is shown in Figure 4. There is a clear discriminative gap between non-obfuscated strings and the high-entropy types of strings (FU and LU). (Note that there is an almost perfect overlap of the FU and LU curves.) However, there is a significant overlap between non-obfuscated strings and B64 strings, meaning that an entropy-based classifier will be unable to accurately distinguish between these two classes. (Note that, since we use randomly generated strings from the Base64 range of characters, our B64 strings have higher entropy than a real Base64 encoding of a natural language string. Therefore, the difference in entropy of natural language and Base64 strings may be even smaller than shown in this plot.)

In conclusion, our experiments confirm Limitation 3: Shannon entropy does not have sufficient discriminative power to allow string obfuscation classification with high accuracy.

4.2 Our Method

Our method was evaluated on the same test set as the discriminative classifiers above. As shown in Figure 3d, our method significantly outperformed the discriminative classifiers in terms of accuracy. Also, in contrast to the discriminative classifiers, the F-score of our method is very close to the accuracy for all four cases, indicating a good balance between false positives and false negatives. Our method performs the best on the FU and LU cases, but also very well for the B64 case. It is noteworthy that the performance does not differ significantly between the FU and LU cases, indicating that a bias towards character sets of large alphabets does not influence the accuracy of our method. The lowest accuracy is observed for the ROT case, most likely because these types of strings are the closest to natural language out of the four types considered.

Another important factor is how the length of a classified string affects the performance of different models. Figure 5 shows plots of the accuracy as a function of string length for our three types of classifiers. For the n-gram based random forest classifier, we show results when using the full feature set, cf. Figure 3a. For these plots, we considered only the cases where the same types of strings were used in training and testing (i.e., the diagonals in Figures 3a and 3c). Note that this is the most advantageous case for the discriminative classifiers, since, in this case, they are not hampered by their limited generalizability. Despite this, our method clearly outperforms the others for string lengths below 10, regardless of "obfuscation" type. Out of the three classifier types, the entropy-based one performs the worst on short strings. For the FU and LU cases, the other classifiers perform slightly better than ours for long strings. However, since our method performs significantly more consistently across various string lengths it still achieves a better overall accuracy. It should be noted that many of the real strings from the F-Droid apps are fairly short, with the most common length being 6. This explains why our method has a much better overall accuracy than the other classifiers, and also highlights the need for accurate obfuscation detection on short strings. Interestingly, we observe two cases with better performance on shorter strings than on longer ones, which is unexpected. We see better performance on short strings for the entropy-based classifier in the B64 case, and likewise for the n-gram based classifier in the ROT case. We suspect that this is due to some specific property being more common for shorter strings, which somehow makes them easier to distinguish from obfuscated strings for some types of classifiers. We intend to look more closely into this phenomenon in future work.

5 DISCUSSION AND FUTURE WORK

In this section, we discuss some limitations of our approach to detecting obfuscated strings, and the method used for evaluating it. We also outline directions for future work.

One potential threat to the validity of our results is the dataset used for evaluation. In order to ensure that strings are representative of real apps, we use strings from open source apps in the F-Droid repository. However, the vast majority of F-Droid strings are from the English language. While our training on the entire WiLI dataset ensures that models are not biased towards specific languages, we cannot rule out the possibility that our method could perform differently depending on the language of non-obfuscated strings. Therefore, further evaluation is needed to investigate this. Another limitation of using the F-Droid strings is that a small percentage of these strings do not actually resemble natural language. This puts an upper bound on the achievable accuracy of any method in our evaluation. A few representative examples (after preprocessing) are shown below:

```
tls ecdhe ecdsa with rc4 128 sha
yyyy mm dd hh mm ss
651fff
```

The first string mostly consists of technical acronyms, whose letter frequencies do not resemble that of any natural language, while the second string denotes a time-stamp format, and the third one appears to be an ASCII representation of a hexadecimal number. We also manually analyzed some of the false negatives produced by our method (synthetic strings misclassified as non-obfuscated). For the B64 and ROT cases, these mostly consisted of short strings that by chance happened to resemble real words. For the FU and LU cases, most false negatives consisted mostly of Chinese letters. We speculate that this is because Chinese has a much larger set of letters, and that our method therefore has a harder time distinguishing which letter combinations (i.e., n-grams) are legal for that language. It may therefore be necessary to use a larger set of training strings for languages with a larger alphabet. We intend to investigate this further in future work.

A third problem with our evaluation methodology is that all methods are tested on synthetic "obfuscated" strings, due to the aforementioned challenges with getting accurate ground truth for a set of real obfuscated strings. While the main aim of our experiments in this paper has been to evaluate the ability of different methods to *generalize* to different types of obfuscated strings, it would of course be of great interest in future work to test our method's effectiveness on various in-the-wild obfuscation schemes.

While the 235 languages in the dataset we used for training covers the most commonly spoken languages, our method is readily extensible by using a training set with more languages. An interesting topic of future work is to study how this influences the accuracy of the method. Another direction for future work is to make our method more resilient against advanced obfuscation schemes. A potential attack against our current design is to use a substitution cipher, where individual characters are replaced with common 3-grams of whole dictionary words from some language. While this would result in greater overhead due to a significant blowup in string size, it would be a fairly straightforward approach to circumvent our method. This type of obfuscation could, however, potentially be defeated by using a more complex language model than our current Naive Bayes approach, such as a deep neural network model. (The use of deep learning would likely, however, require a significantly larger training set to avoid potential problems with memorizing or overfitting, which are naturally avoided by our current Naive Bayes model.)

We also acknowledge that our method will not work if a string obfuscation technique of the type used by DexGuard is employed by an app, since in that case, encrypted strings are stored as byte arrays rather than actual strings in the string section of a DEX file.

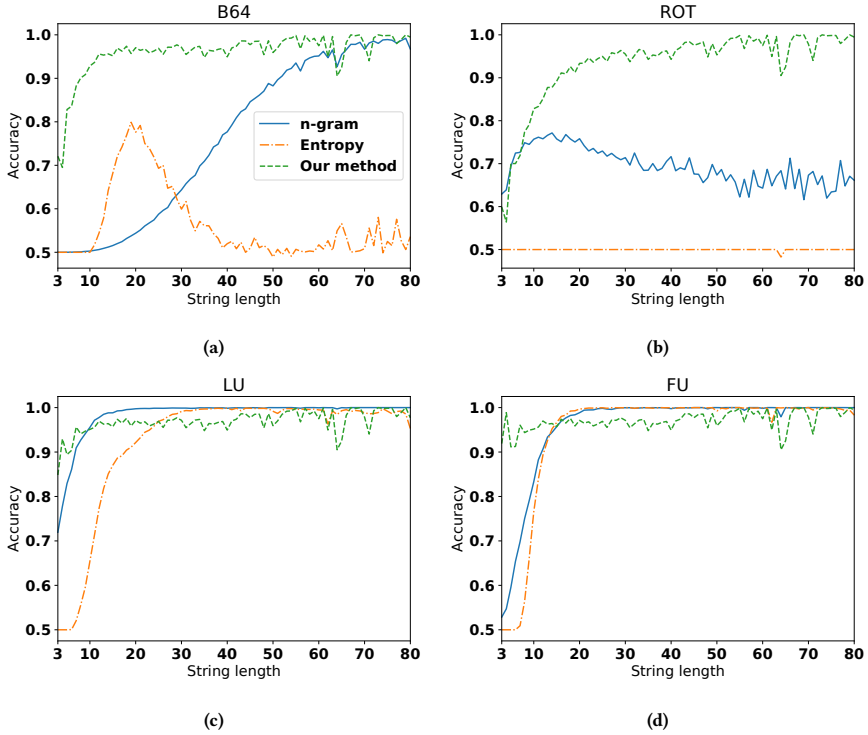

Figure 5: Accuracy as a function of string length when a classifier is trained and tested on positive samples from the respective classes B64 (a), ROT (b), LU (c), and FU (d). (For our method, the positive samples are used for testing only.)

Other methods, such as signature-based approaches, must be used to detect these kinds of obfuscated apps.

Finally, we believe that our method may have applications outside the field of Android obfuscation detection. As future work, we intend to investigate the potential range of application in other fields. In the next section, we discuss some problems in other fields, where our method might find potential use.

6 RELATED WORK

In addition to the work by Mirzeai et al. [25] and Dong et al. [14] that we discussed in Section 2.2, the works by Wang and Rountev [33] and Kaur et al. [23] both aim to detect obfuscated apps. Both of these latter works, however, attempt to detect signatures of the *obfuscation tool* used. Wang and Rountev extract strings with specific semantics (file names, package names, class names, etc.) from each DEX file, and train a linear SVM classifier. They report 97% accuracy in determining which obfuscator was used. Kaur et al. propose a similar system based on pattern detection within a raster-image visualization of an android app.

Others have also considered similar problems in other domains. Kartaltepe et al. [22] use decision trees to detect encoded botnet C&C communication in tweets. Similarly to us, they also found that the classifier had trouble generalizing when different encoding schemes were used. Henderson et al. [18] explored using different types of classifiers for the same task, and found an n-gram based Naive Bayes classifier (i.e., what we base our method on) to work best. In contrast to our work, however, they still rely on both positive and negative samples, and are therefore susceptible to Limitation 2b. Qi et al. [28] propose a mechanism based on bigram frequencies to detect DNS tunnels. Freeman [16] proposed a method for detecting bot accounts in social media by their name, using an n-gram based Naive Bayes model. Beskow et al. [11, 12] also describe methods to detect bots using account user names.

7 CONCLUSION

In this paper, we have studied techniques for detecting obfuscated strings in Android apps. We have identified and empirically verified several limitations of existing machine-learning based approaches to detecting obfuscated strings, and we have proposed a novel method to address these limitations. By employing a generative classification approach, our method does not rely on positive (i.e., obfuscated) samples at all for training. Our experiments indicate that this allows our method to achieve superior generalizability to different obfuscation schemes, compared to using existing off-the-shelf machine learning approaches.

REFERENCES

[1] [n.d.]. Allatori. http://www.allatori.com/.
[2] [n.d.]. Allatori — String Encryption. http://www.allatori.com/features/string-encryption.html.
[3] [n.d.]. Dalvik Executable format. https://source.android.com/devices/tech/dalvik/dex-format.
[4] [n.d.]. DashO. https://www.preemptive.com/products/dasho/overview.
[5] [n.d.]. DexGuard. https://www.guardsquare.com/en/products/dexguard.

[6] [n.d.]. DexProtector. https://dexprotector.com/.
[7] [n.d.]. F-Droid. Retrieved 2019-03-02 from https://f-droid.org
[8] [n.d.]. KlassMaster — Java String Encryption. https://www.zelix.com/klassmaster/featuresStringEncryption.html.
[9] 2019. The Unicode Standard Version 12.0, Chapter 4. https://www.unicode.org/versions/Unicode12.0.0/ch04.pdf.
[10] B. Amos, H. Turner, and J. White. 2013. Applying machine learning classifiers to dynamic Android malware detection at scale. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*. 1666–1671.
[11] David M Beskow and Kathleen M Carley. 2018. Using random string classification to filter and annotate automated accounts. In *International Conference on Social Computing, Behavioral-Cultural Modeling and Prediction and Behavior Representation in Modeling and Simulation*. Springer, 367–376.
[12] David M. Beskow and Kathleen M. Carley. 2019. Its all in a name: Detecting and labeling bots by their name. *Computational and Mathematical Organization Theory* 25, 1 (2019), 24–35.
[13] Justin Del Vecchio, Feng Shen, Kenny M Yee, Boyu Wang, Steven Y Ko, and Lukasz Ziarek. 2015. String analysis of Android applications (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 680–685.
[14] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, XiaoFeng Wang, and Kehuan Zhang. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In *Security and Privacy in Communication Networks*. Springer International Publishing, 172–192.
[15] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 576–587.
[16] David Mandell Freeman. 2013. Using Naive Bayes to Detect Spammy Names in Social Networks. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security (AISec '13)*. ACM, 3–12.
[17] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A Large-scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-malware Products. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 421–431.
[18] Jette L Henderson, Daniel J Frazee, Nick P Siegel, Cheryl E Martin, and Alexander Y Liu. 2016. Evaluating Methods for Distinguishing Between Human-Readable Text and Garbled Text. In *The Twenty-Ninth International Flairs Conference*.
[19] k00dr. 2016. Cracking the Allatori string encryption. https://forum.moparisthebest.com/t/cracking-the-allatori-string-encryption/177659.
[20] Andi Fitriah Abdul Kadir, Natalia Stakhanova, and Ali Akbar Ghorbani. 2015. Android Botnets: What URLs are telling us. In *International Conference on Network and System Security*. Springer, 78–91.
[21] Hyunjae Kang, Jae wook Jang, Aziz Mohaisen, and Huy Kang Kim. 2015. Detecting and Classifying Android Malware Using Static Analysis along with Creator Information. *International Journal of Distributed Sensor Networks* 11, 6 (2015).
[22] Erhan J. Kartaltepe, Jose Andre Morales, Shouhuai Xu, and Ravi Sandhu. 2010. Social Network-Based Botnet Command-and-Control: Emerging Threats and Countermeasures. In *Applied Cryptography and Network Security*, Jianying Zhou and Moti Yung (Eds.). 511–528.
[23] R Kaur, Y Ning, H Gonzalez, and N Stakhanova. 2018. Unmasking Android Obfuscation Tools Using Spatial Analysis. In *2018 16th Annual Conference on Privacy, Security and Trust (PST)*. 1–10.
[24] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers and Security* 51 (2015), 16–31.
[25] O Mirzaei, J M de Fuentes, J Tapiador, and L Gonzalez-Manzano. 2019. AndrODet: An adaptive Android obfuscation detector. *Future Generation Computer Systems* 90 (2019), 240–261.
[26] Yoni Moses and Yaniv Mordekhay. 2018. Android app deobfuscation using static-dynamic cooperation. https://www.virusbulletin.com/virusbulletin/2019/03/vb2018-paper-android-app-deobfuscation-using-static-dynamic-cooperation/. In *VB2018*.
[27] Mila Dalla Preda and Federico Maggi. 2017. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques* 13, 3 (2017), 209–232.
[28] Cheng Qi, Xiaojun Chen, Cui Xu, Jinqiao Shi, and Peipeng Liu. 2013. A Bigram based Real Time DNS Tunnel Detection Approach. *Procedia Computer Science* 17 (2013), 852 – 860. First International Conference on Information Technology and Quantitative Management.
[29] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2013. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ACM, 329–334.
[30] 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.
[31] A-D Schmidt, Rainer Bye, H-G Schmidt, Jan Clausen, Osman Kiraz, Kamer A Yuksel, Seyit Ahmet Camtepe, and Sahin Albayrak. 2009. Static analysis of executables for collaborative malware detection on android. In *2009 IEEE International Conference on Communications*. IEEE, 1–5.
[32] Martin Thoma. 2018. The WiLI benchmark dataset for written language identification. *arXiv preprint arXiv:1801.07779* (2018).
[33] Yan Wang and Atanas Rountev. 2017. Who changed you?: Obfuscator identification for Android. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 154–164.
[34] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, Cham, 252–276.
[35] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. 569–584.
[36] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. 2013. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 82–101.