Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy

Preprint · February 2020

Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy

Leonid Glanz*, Patrick Müller*, Lars Baumgärtner*, Michael Reif*, Sven Amann*, Pauline Anthonysamy+, and Mira Mezini*
<glanz,mueller,baumgaertner,reif,amann,mezini>@cs.tu-darmstadt.de,anthonysp@google.com
*Technical University of Darmstadt, +Google Inc.

Abstract

String obfuscation is an established technique used by proprietary, closed-source applications to protect intellectual property. Furthermore, it is also frequently used to hide spyware or malware in applications. In both cases, the techniques range from bit-manipulation over XOR operations to AES encryption. However, string obfuscation techniques/tools suffer from one shared weakness: They generally have to embed the necessary logic to deobfuscate strings into the app code.

In this paper, we show that most of the string obfuscation techniques found in malicious and benign applications for Android can easily be broken in an automated fashion. We developed StringHound, an open-source tool that uses novel techniques that identify obfuscated strings and reconstruct the originals using slicing.

We evaluated StringHound on both benign and malicious Android apps. In summary, we deobfuscate almost 30 times more obfuscated strings than other string deobfuscation tools. Additionally, we analyzed 100,000 Google Play Store apps and found multiple obfuscated strings that hide vulnerable cryptographic usages, insecure internet accesses, API keys, hard-coded passwords, and exploitation of privileges without the awareness of the developer. Furthermore, our analysis reveals that not only malware uses string obfuscation but also benign apps make extensive use of string obfuscation.

Keywords string (de-)obfuscation, Android apps, slicing

1 Introduction

Obfuscation protects applications against abusive practices (e.g., repackaging) but also hides malicious intent (e.g., malware) [17, 37]. It significantly impedes the analysis of apps to check their compliance with privacy regulations or to inspect them for detecting malware [36, 45]. In particular, *string obfuscation*, which is applied by most existing obfuscators [12, 16, 40, 52, 62], can hide paths, URLs, and intents to track user activities, thus compromising the user's privacy, or opening shells for remote command execution to execute a malicious payload.

Opposing prior work [17, 37, 56], which have stated that strings are often not obfuscated in the wild, in this paper, we provide strong empirical evidence (cf. Section 4.2.1) that it is widely used in both malicious and benign apps. The usage of string obfuscation in the benign apps is to a significant extent due to integrated ad libraries – hence, even the app developer may not be aware of their presence. Under these conditions, approaches that analyze plain strings [21, 34, 39, 41, 60, 63, 65] are deemed to be ineffective, and techniques for automatically uncovering obfuscated strings are highly needed.

Given that the deobfuscation logic usually is part of the application [58], an analyst can try to debug or to run the application with a monkey script. However, such "brute-force" testing has serious drawbacks. First, given that there is no guarantee that all execution paths are covered and that the appropriate execution point to deobfuscate a string is unknown, the latter maybe not triggered. Second, obfuscated applications could detect the debugging mode and avoid to execute the deobfuscation [45] since deobfuscation functionality is often protected by guards trying to defend against artificial runtime environments [54].

To address obfuscation, several dedicated approaches [5, 14, 15, 26, 45, 49, 61, 64] have been proposed. But, they suffer from limited scalability and generality. To circumvent defenses and to force the execution of all branches, many existing approaches [5, 45, 61] typically alter if statements of the target program and then run the code with all combinations of values. Given that many obfuscators perform automatic string obfuscation on millions of apps, the above approaches are not suited for large-scale analyses. The approach by Zhou et al. [64] slightly reduces the number of executions, but at the cost of generality, as its emulator is fitted to string operations only. In fact, to the best of our knowledge, all works lack a systematic analysis of existing automatic obfuscators and their scope.

To address the above problems with the state-of-the-art, we propose StringHound, a novel string deobfuscation technique for automatically obfuscated strings in Java bytecode. StringHound generalizes to different string obfuscations and executes only the code necessary for their deobfuscation to ensure scalability.

To ensure generality, we performed a comprehensive study of existing obfuscation techniques and used the gained insights to guide the design of StringHound. We systematically studied how strings are obfuscated in ad libraries

(cf. Section 2). These libraries often employ string obfuscation [10, 13, 46, 50, 51] and are, hence, a good source for systematically surveying string obfuscation techniques used in the wild. To ensure that only code necessary for deobfuscation is executed, StringHound (a) locates the usage of obfuscated strings within the application code, (b) extracts the deobfuscation logic alongside with all the context it needs to perform, and (c) executes the extracted code directly on a Java Virtual Machine (JVM) to yield the plain-text versions of obfuscated strings.

For a fast location, we propose two classifiers, one that uses decision trees [44] to identify potentially obfuscated strings (String Classifier), and another one that uses Spearman correlation [38] to identify code of deobfuscation methods (Method Classifier). For the extraction of the deobfuscation logic, we propose a specially targeted method-slicing that includes all program statements that affect the state of an obfuscated-string sink located within a given method. Additionally, StringHound extracts the execution context of the deobfuscation logic and injects the slice into it. Through the injection of the slice, countermeasures, potentially introduced by obfuscators, are rendered ineffective. Finally, StringHound executes the resulting slice within the extracted context to obtain deobfuscated strings.

We evaluated StringHound and four available state-of-the-art deobfuscation tools [14, 15, 26, 49] by applying them to a set of apps that we obfuscated with 21 different techniques. The evaluation shows that StringHound outperforms the other tools by orders of magnitude (2.5% vs. 73.9%). We also applied StringHound to four sets of benign and malicious real-world apps: (a) a random sample of 100,000 apps, (b) popular apps based on AndroidRank Top 500 [2], (c) malware from Contagio [9], and (d) apps from the Google Play Store in 2018 classified as malicious by VirusTotal. StringHound's classifiers were key to enabling a study of more than 100,000 apps by using them to filter out apps that do not contain any obfuscated strings to avoid unnecessary slicing and deobfuscation steps. A brute-force approach that tries to deobfuscate each string in all apps would be infeasible, given that an app such as, WhatsApp [59] contain 17,176 strings.

Our study shows that string obfuscation is used at least 12 times more often than previous studies stated [17, 37, 56]. We categorize our findings and give insights on how string-obfuscation is used in different kinds of apps. Besides expected results, e.g., obfuscated URLs and commands in malware sets, we surprisingly found that 76% of the 100,000 apps contain obfuscated strings. An in-depth analysis revealed that several strings are commonly found in ad libraries integrated into apps. Moreover, we identified two apps in the Top 500 set that conceal suspicious behavior through string obfuscation. They collect sensitive information from a user's mobile phone, such as call logs and location information, to build a user profile for tracking. Furthermore, they also check for the `SuperUser.apk`, which grants root access to the mobile phone. These apps are installed over 20 million times and are not flagged as malicious by VirusTotal [55].

Table 1. String Obfuscation Techniques in Ad Libraries

Ad Library	Cipher	Enc.	Countermeasures
com.chamspire		B64	
com.intentsoftware		B85	
com.ironsource		custom	
com.youmi		custom	
com.adcolony		URL	
a.a.a	AES		SO
com.google.android	AES&Bit	B64	SI
cn.pro.sdk	Bit		BA
br.com.tempest	Bit		Key Changed by SW
com.applovin	Bit		Key in BA
br.com.tempest	Bit		Key is SC
com.tnkfactory	Bit		OI
br.com.tempest	Bit		SC
com.google.android	Bit		ST
br.com.tempest	Bit		SW
com.apptracker	Bit		TK
com.adlib	Bit		TM
com.mnt	Bit	B64	Key is Index of BA
com.waystorm.ads	Bit	B64	KMC
com.vpon.adon	DESede		
com.mt.airad	DESede	B64	

In summary, this work makes the following contributions:

1. A study which identified 21 unique string obfuscation techniques used by state-of-the-art obfuscators (Section 2).
2. Two novel techniques for locating string obfuscation (Section 3.1.2 & 3.1.3).
3. StringHound [47], an novel open-source string deobfuscator that integrates the proposed classifier and slicing techniques. StringHound outperforms publicly available deobfuscators by orders of magnitude (2.5% vs. 73.9%), effectively rendering current string obfuscation techniques ineffective (Section 3).
4. A study of string obfuscation in four real-world data sets (Section 4.2.2) containing more than 100,000 apps and providing valuable insights into the prevalence of obfuscation usage in the wild.

2 Existing String Obfuscation Techniques

We systematically analyzed string obfuscation in ad libraries, since these libraries have been shown to use it in various forms and quantities [10, 13, 46, 50, 51] The knowledge gained from the following analysis was used as a basis for designing our approach and conducting controlled experiments for evaluation purposes.

Methodology. As there is no publicly available set of ad libraries that use string obfuscation, we sampled our own collection of ad libraries by analyzing apps which integrate

them. First, we collected a list of package names of frequently used ad libraries [29] and a list of URLs of ad networks [3]. We reversed the internet domain name [11] (e.g., youmi.net ⇒ `net.youmi`) of the URLs to guess package names of ad libraries. Next, we searched for code with the respective package names by analyzing 100,000 randomly sampled apps from AndroZoo [25] and found 640 unique ad libraries distributed across 81,008 individual apps.

To identify string obfuscation techniques, we manually inspected obfuscated strings and analyzed their flows until they reached methods that are not modifiable by the obfuscator (e.g., `System.println`). Nevertheless, we did not focus only on string constants because, in the obfuscated form, they are often also stored in byte arrays [48]. Hence, we considered any data structure which can be used to hide string representations and refer to such data structures in the following as obfuscated strings.

During our analysis, we classified a string as not obfuscated, if it flows, without any modification, into an unmodifiable method [1]. Additionally, if the string contains multiple words found in a dictionary or matches a known format (e.g., XML), it is not classified as obfuscated. In the case of other data structures, we considered all bit operations that are performed on constant values to be an indication for string obfuscation. For each obfuscated string, we then manually analyzed the code that deobfuscates the string to determine the used technique.

Overview of identified techniques. Using our methodology, we identified 21 unique string obfuscation techniques shown in Table 1. Among the identified techniques are also those of the state-of-the-art obfuscation tool manufacturers [12, 16, 40, 52, 62]. For each technique, we show the cipher, the encoding, and countermeasures used to make detection by static/dynamic analyses more difficult. The used ciphers are bit manipulations such as XOR operations (Bit), DESede, AES, and the combination of bit manipulation and AES. The encodings consist of Base64 (B64), URLEncoder (URL), Base85 (B85), or custom encodings (custom), e.g., using a BigInteger with base 33, or splitting a string and concatenating the characters at the beginning and the end of the new string. We identified the following countermeasures:

Serialized Object (SO): One technique loads a serialized object at runtime that implements a deobfuscation method. Subsequently, it must be called through reflection to deobfuscate a string. This technique evades deobfuscators that rely exclusively on identifying and executing deobfuscation methods.

Static Initializer (SI): The static initializer computes the deobfuscation key. This practice evades deobfuscators who extract the logic of only one particular method for execution.

Byte Arrays (BA): Two of the analyzed techniques use byte arrays to hide the representation of obfuscated strings and, thus, evade deobfuscators that rely on this representation.

Switch Statements (SW): Two techniques use a switch statement in a loop to deobfuscate a different string in each loop iteration. Both store the resulting strings in an array, and each method accesses this array. These techniques evade deobfuscators that search for an explicit deobfuscation method.

Stack Calls (SC): Two techniques hard-code the calling context (e.g., method name and class name) of the deobfuscation method. While one technique checks the calling context in a conditional statement, the second one uses the calling-context information as part of the deobfuscation key. Both techniques evade deobfuscators that execute the deobfuscation logic without a specific context. However, only the second one enforces the extraction of the context for slicing approaches because it is a direct part of the deobfuscation.

Object Initializer (OI): One technique deobfuscates strings by inserting a specific class whose constructor initializes the deobfuscation key. Subsequently, a method of the constructed object deobfuscates all strings which were obfuscated with the initialized key. This technique evades deobfuscators that execute only static methods.

Stream Transfer (ST): Hidden channels are used to transfer obfuscated strings to deobfuscation methods. For instance, one obfuscator transfers the obfuscated string via input/output streams to its deobfuscation method. This technique evades deobfuscators that track obfuscated strings and would, therefore, miss data flows arising from streams.

Two Keys (TK): Two different keys are used for string deobfuscation. This usage evades deobfuscators that try brute force guessing of one key to uncover obfuscated strings.

Two Methods (TM): Two methods are used for string deobfuscation. This usage evades deobfuscators that execute only one deobfuscation method to uncover obfuscated strings.

Key Management Calls (KMC): One technique initializes deobfuscation keys directly before their usage by using object fields. This technique hinders deobfuscators that do not handle such initialization.

As depicted in Table 1, different combinations of ciphers, encodings, and countermeasures are used as techniques for string obfuscation. We refer to these combinations as obfuscation schemes. Some of the techniques are used in state-of-the-art commercial obfuscation tools, and developers most commonly use these tools to obfuscate strings in Android and Java apps. The findings of this study are surprising as none of the identified techniques requires a broader focus than the one described above.

> **Observation 1.** *All analyzed obfuscation schemes initialize the deobfuscation within the same class that encloses the deobfuscation methods. This fact suggests that no heavyweight inter-procedural analysis is necessary.*

[1] Most obfuscators produce strings with unreadable symbols and, therefore, contain no words.

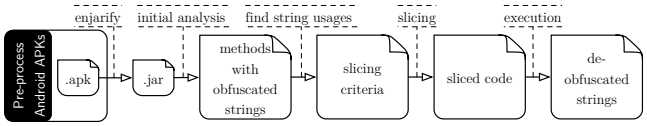

Figure 1. Overview of StringHound's Approach

Next, we will use the gained knowledge of these identified schemes to circumvent and reverse them all with our approach. To obfuscate samples with the schemes, we either acquired a tool using them or re-implemented them ourselves.

3 The StringHound Approach

StringHound processes Java bytecode in five steps. Figure 1 shows a high-level view of this process. First, when we analyze an Android Package (APK), we transform its Dalvik bytecode to Java bytecode and process the result with our analysis. Second, to reveal obfuscated strings, we need to identify the methods that potentially use them. For locating usages of obfuscated strings, we propose two complementary techniques: a classifier for identifying potentially obfuscated strings (String Classifier), and a classifier for identifying deobfuscation methods (Method Classifier). The string classifier operates on characteristics of obfuscated strings by using decision trees. The method classifier matches distributions of instruction-based tokens from known deobfuscation methods with the Spearman correlation. Third, we find the starting point for the slicing (slicing criterion) in the methods that contain the usage of the obfuscated strings. Forth, we use a specially targeted method-slicing that computes all program statements that affect the state of a given slicing criterion. Finally, StringHound injects the slice into the execution context of the deobfuscation logic. Afterward, it executes the resulting slice to obtain deobfuscated strings. The injection of the slice into the context renders countermeasures introduced by obfuscators ineffective. Our detailed description of StringHound shows the design decisions taken to address the obfuscation schemes presented in Section 2.

3.1 Classifiers

In this section, we present our classifiers and evaluate their precision and recall. In Section 4.2.1, we provide empirical evidence that they are both needed.

3.1.1 Training & Evaluation Data Set

For the training and evaluation of our classifiers we downloaded the newest versions of all 1,879 apps from F-Droid [20]. We chose F-Droid because it only consists of open-source software. Hence, string obfuscation does not make sense and is also not applied. This data set is used as ground truth of plain strings and methods which do not contain any deobfuscation logic. These two properties enable the obfuscation without dealing with the influence of previously existing obfuscation artifacts. Other works [6, 37, 57] also used the F-Droid store because of these two properties.

3.1.2 String Classifier

To train the classifier, we generated a data set of 1,918,687 obfuscated from the same amount of non-obfuscated strings. The set of non-obfuscated strings was extracted from the F-Droid data set (cf. Section 3.1.1). To obtain obfuscated strings, we applied the 21 obfuscation schemes identified in our case study (Section 2) to non-obfuscated strings of F-Droid applications. This effort yields 32,379 obfuscated apps[1] from which we extracted all strings. As a result, we acquired significantly more obfuscated strings than plain strings. To avoid a bias towards obfuscated strings, we took all strings from the plain apps and randomly selected the same number of strings from the obfuscated ones.

Approach We extracted 49 different features from the collected strings and trained a model using Weka's REPTree. This enables a fast comparison of the features by building a regression/decision tree using the most discriminatory features to check the most relevant ones first. However, REPTree considers all these features before it comes to a final decision for a given string. The authors checked that the classifier considers all features for its final decision by manually traversing the decision tree. We used the following features to train the string classifier (A detailed list is in Section A.4):

Format: In the study presented in Section 2, we observed that obfuscated strings contain special characters. Nevertheless, we cannot classify a string as obfuscated just because it contains special characters – plain strings of certain formats may also use special characters. To avoid matching such plain strings, we use various patterns to discern format usages such as XML (e.g. `</th>`) and HTML colors (e.g. `#FFAE40`) in the feature vector. These flags are used to give the model a hint that the analyzed string might not be obfuscated. However, these hints should not be confused with filtering, as they are only a part of the classifier's decision.

Statistical Tests: Previous statistical analyses of encryption mechanisms [27, 32, 42] show that obfuscated strings often have a random (close to equal) distribution of characters. We use random distribution as a discriminating feature to distinguish between obfuscated and other strings with special characters. To check whether the distribution of the characters in a string is random, we use three different measures because each one is suited for different scenarios we encountered. With the *Chi-squared* test, we measure the deviation of the characters from the equal distribution of these characters, since randomized characters are often equally distributed. With the deviation from the *average distribution*, we measure whether the given string belongs to a language or whether

[1] We were not able to obfuscate every app with every obfuscator due to version incompatibilities between obfuscators and APKs to be obfuscated.

the characters were only rotated (e.g. caesar cipher [27]). *Normalized entropy* was previously [32] used to identify encrypted malware. We reuse it to identify encrypted strings.

AndroDet: We use the number of equals, number of dashes, number of slashes, number of pluses, and the sum of repetitive characters from the feature list of AndroDet [37] which are used to identify if an app uses string obfuscation. However, AndroDet averages these features over all strings in an app and is therefore not able to classify individual strings.

Compression rate: Obfuscated strings may be confused with compressed data, such as images compressed using JPEG and stored in strings. To identify those strings, we compressed them and compared the resulting length against the original length; the resulting length changes if the original content is not already compressed [42].

Cryptographic libraries: Cryptographic libraries use byte-encoded strings to initialize their algorithms, and this may cause false positives because they are similar to obfuscated strings. To avoid matching such encoded strings, we check if a string usage is contained in a known cryptographic library.

Dictionary words: The study in Section 2 revealed that obfuscated strings contain few or no words. We use a dictionary to check whether a string contains words or identifiers [19].

String characteristics: Finally, we extract eight features related to character distributions, e.g., character counts, digits.

Evaluation We use 80% of our data set for training and testing and 20% for validation. To train and test the model, we use a 10-fold cross-validation measure. Our validation data revealed a precision of 98.79% and a recall of 89.75%. We identified two root causes for false negatives. The first cause is that obfuscated strings accidentally contain valid words (this is exacerbated by languages where a single character can be a valid word, e.g. Chinese). The second, more prevalent cause is obfuscated strings that consist of digits since these frequently occur in plain text strings as well.

3.1.3 Classifier for Deobfuscation Methods

The string classifier may miss obfuscated strings that are hidden in other data types, e.g., strings represented as byte arrays (cf. Table 1 BA). To address this problem, we train a second classifier that identifies deobfuscation methods. We use the identified methods from Section 2.

Approach We postulate that deobfuscation methods use certain tokens more often than ordinary methods. This idea is inspired by statistical analysis of English text, which, e.g., contains a high number of the character 'e' [42]. Likewise, deobfuscation methods may use the XOR token more frequently than ordinary methods. To this end, we extract all tokens used in deobfuscation methods of the identified schemes (cf. Section 2). The extraction of the tokens is performed using the *Structure-preserving Representation (SPR)* [22]. This representation preserves the structure of a method's code but abstracts away information that gets changed in obfuscated code and, thus, would produce noise for the classification, e.g., all name and type information that does not occur in the Android standard library is removed. We compare the SPR-token distribution of our set of deobfuscation methods with the ones found in apps using Spearman's correlation to identify similar methods. This comparison enables the method classifier to identify not only exact matches of the token distribution but also variations of it. Furthermore, we limit our token extraction to those tokens occurring in known deobfuscation methods; as a result, our method classifier is also able to identify in-lined deobfuscation logic.

Evaluation Recall that the primary purpose of the method classifier is to locate deobfuscation schemes that represent obfuscated strings in other data structures. As reported in Section 2, only two such schemes exist, and these are the ones that generate variations of the deobfuscation logic. Nevertheless, to assess the precision and recall of the method classifier, we use not only the schemes which generate variations of known deobfuscation methods as a ground truth but the methods of all the obfuscation tools acquired in Section 2. We use methods generated by all tools since the method classifier discriminates all kinds of deobfuscation methods, not only those that handle other data structures than strings.

The two mentioned tools vary the logic of the deobfuscation methods in different ways [43]. First, they use random numbers as obfuscation keys. Second, they permute the order of formal parameters or change the method's signature. Third, they alter the position of code blocks, whose execution order does not matter. Finally, deobfuscation methods may also depend on the context of string usages. For instance, if a string is used only once in a class, one tool in-lines the deobfuscation logic at the string usage site; in other cases, this logic is extracted into a separate called method.

To measure the precision of the classifier and recall for these variations, we applied the all obfuscators to the F-Droid data set (cf. Section 3.1.1). We were able to generate 2,127 obfuscated apps, at least 1,000 apps for each obfuscator [2]. The deobfuscation methods in the resulting obfuscated apps constitute our ground truth for measuring recall and precision.

To extract them, we use information from the mapping files produced by the obfuscator tools for each app. Mapping files enable app developers to find the original names in the source code for crash reports using obfuscated names. Consequently, methods and fields with no entry in the mapping file must have been added by the obfuscator. We add all new methods and also methods that access newly added fields to the ground-truth list. The newly added fields are used

[2]We were not able to obfuscate every app with every obfuscator due to version incompatibilities between obfuscators and APKs to be obfuscated.

to identify in-lined deobfuscation logic, which resides in a previously existing method.

Altogether, we obtain a list of 144,190 methods that contain deobfuscation logic, either in a separate method or in-lined into previously existing methods. The comparison of this list with the method classifier's output shows that it identifies the variants of deobfuscation methods generated by the two subject obfuscator tools with a precision of 99.66% and a recall of 97.42%. We conclude that our classifier is very accurate, missing only a few deobfuscation methods. A detailed analysis revealed that these methods have in-lined obfuscation logic, but already used byte arrays before the obfuscation. These previously existing byte arrays add noise to the measured token distribution and weaken the correlation between the method under analysis and our set of known deobfuscation methods.

3.2 Slicing Relevant String Usages

A slicing criterion (s_{crit}) is any instruction within certain methods, which we call candidate methods, that produces a string value. A method m is in the set of candidate methods if (a) it contains instructions that consume a char sequence as a parameter (method calls, but also field writes, array stores, and return instructions), called *locations of interest (LoIs)*, and (b) satisfies one of following conditions: (i) the strig classifier found an obfuscated string in `m`, (ii) `m` calls a method `n`, which the method classifier classified as a deobfuscation method, or (iii) `m` is itself classified as a deobfuscation method (in-lined deobfuscation logic).

Since the classifiers identify neither *LoI*s nor slicing criteria directly, we have to search for them in the candidate methods. We use OPAL [18] to find all instructions that operate on values of type `CharSequence`, or a subtype thereof, in particular `java.lang.String`. All s_{crit} are expressions that result in strings which are afterward passed to some *LoI*. Given a candidate method that contains *LoI*s, we identify all s_{crit} while ignoring constant string expressions.

Our slicing algorithm performs backward slicing [1, 7] with forward-steps to collect all instructions necessary for the execution of other relevant instructions. For instance, if the slice contains a *new* instruction, we also collect the corresponding constructor invocation. Classical slicing algorithms inspired our algorithm (cf. Binkley et al. [7]), which we adapted to our particular needs[3]. If several potential sources for a given string parameter are present, we start the slicing separately for each of them, with each of them as a single slicing criterion.

3.3 Executing Sliced String Usages

To obtain the deobfuscated string that in the original application would flow into the *LoI*, we extend the slice by a call to a method that logs the string. This call effectively replaces the original *LoI* with the call to the logging method, which allows us to retrieve the deobfuscated string value. We add a return statement to the slice to ensure that the signature of the sliced method can remain as before. If we need to return a value, we either return `null` or the numeric value `0`—depending on the declared return type. Next, we replace the body of the original candidate method with the extended slice; this ensures that the execution context w.r.t. the name of the declaring class as well as the name and signature of the method is identical to the original code.

To execute the sliced method in its context, we have to make the class concrete, if it is abstract. Therefore, all abstract methods are made concrete by returning default values of the declared return type. We rewrite the class so that it extends a superclass that we generate, including the corresponding static initializer and super calls. With this step, we increase the likelihood that the initialization of our class containing the sliced method does not abort with an exception. Recall that we have no means to determine appropriate parameter values that we could use and, therefore, always have to use default values. The generated superclass also implements all methods transitively called by the sliced method. As previously, we return default values if required.

We set up the classpath to include all classes of the original application, except the modified one. Additionally, we add the new class as well as our new superclass to ensure that our slice can find any application class used in its code. As a replacement of the original `android.jar`, we use an artificial jar with methods stubs. Methods that have to return a value return the type's default value (e.g., `null` or `0`). All these transformations together, in combination with our slicing approach, enable StringHound to circumvent all obfuscation techniques discussed in Table 1.

Finally, we call the resulting method reflectively using default values for the parameters when necessary. The method will then call our logging method to record the deobfuscated string.[4] We chose to call the sliced methods with default values because the choice of them is simple and caused no overhead. Nevertheless, our approach does not depend on this choice and can be extended to support more advanced methods for determining the parameter values such as fuzzing.

4 Evaluation

We performed two studies (a) comparing StringHound with other string deobfuscators, and (b) assessing the performance of StringHound on real-world apps. The setup consists of a Server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPU, and 512 GB RAM. The analyses were run using OpenJDK 1.8_212 64-bit VM with 20 GB of heap memory, and a 5s timeout for a single string deobfuscation.

[3] Further details about our slicing algorithm are omitted for space reasons.

[4] We can specify a time limit for the slice execution, to cancel long-running slices.

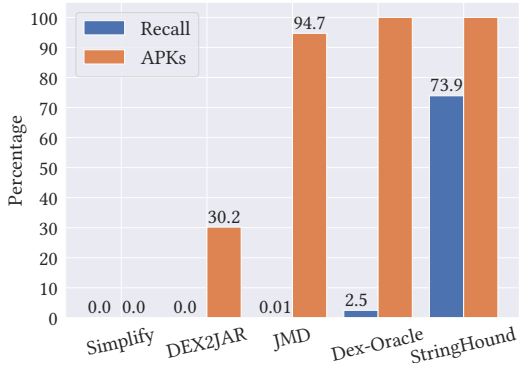

Figure 2. Recall and Successfully Processed APKs

4.1 Comparison with Other Deobfuscators

We evaluated StringHound against Dex-Oracle [14], Simplify [49], JMD [26], and DEX2JAR [15]. To the best of our knowledge, these are the only freely available deobfuscators.

We randomly picked 1,000 obfuscated apps from our F-Droid data set (cf. Section 3.1.1) as input to the deobfuscators for this experiment. Two comparison metrics are used: (a) percentage of APKs processed without termination errors; and (b) recall, which we define as the percentage of *unique deobfuscated strings* over *all unique strings* in the original apps. The precision metric is discarded since our data set contained only obfuscated strings. Therefore, there cannot be false positives (i.e., plain strings identified as obfuscated). However, StringHound's false positive rate is restricted by the false positives produced by the string classifier and the method classifier. The results are summarized in Figure 2. In the following, we discuss each deobfuscator individually.

Simplify [49] applies semantic-preserving transformations to re-engineer the APK's code, such as constant propagation and dead code removal. To enable transformations, it executes each method on a custom Dalvik virtual machine and returns a graph with all possible register and class values for every execution path. *Simplify* can be used as a deobfuscator in limited cases [49], namely for deobfuscation methods that do not depend on any state and use only constants. In such cases, constant propagation can uncover hidden information. Additionally, *Simplify* optimizes all statements also the ones which are not relevant to deobfuscate a string. Unfortunately, *Simplify*'s re-engineered APKs were completely broken and could not be analyzed to produce results; hence, Figure 2 reports 0% for both values.

DEX2JAR [15] transforms Dalvik bytecode to Java bytecode. It has a sub-module that executes methods with a certain signature for deobfuscation purposes. Similar to our approach, it executes the code in the JVM. However, unlike StringHound, DEX2JAR needs the user to provide the deobfuscation method to be executed. We applied our deobfuscation method classifier to each app and used its output as input for DEX2JAR. Providing the same deobfuscation methods as input to DEX2JAR and StringHound enables a fair comparison of the two. However, DEX2JAR processed only 30% of the APKs without errors, and it was unable to deobfuscate a single string, resulting in a 0% recall. This weak results in our empirical study are caused by DEX2JAR's assumption that deobfuscation methods are in the same class as the obfuscated string. Moreover, DEX2JAR assumes that all constant values needed for the execution of the deobfuscation method are provided before it is called, while the latter can also be the result of other accesses or computations. Unfortunately, none of the obfuscation techniques that we surveyed in Section 2 matches these conditions.

JMD [26] re-implements deobfuscation logic of known obfuscators [12, 40, 62] to execute it with directly-propagated constants. These constants are extracted from previously identified immediate callers of known deobfuscation methods. After the execution of the deobfuscation logic, the calls to this logic is replaced with the revealed strings. Unlike our approach, JMD does not consider field accesses or other ways to retrieve the propagated values. It identifies obfuscated strings by searching for a specific loading-instruction (`LDC`). Therefore, it misses almost all obfuscated strings, which would be produced by the techniques from Section 2 because they are loaded by a different instruction (`LDC_W`). Additionally, JMD uses a fixed set of method signatures without considering variations or in-lining of deobfuscation logic. Finally, the deobfuscation logic uses a constant key, but as shown in Section 3.1.3, the key varies with each string usage. JMD's limitations lead to its poor performance: while successfully processing 94% of the APKs, only 0.01% of the strings were deobfuscated.

Dex-Oracle [14] searches for deobfuscation methods and executes them in an emulator. It uses fixed method signatures to search the app code. Therefore, it misses variations of methods produced the same obfuscator and in-lined deobfuscation code. For instance, only two kinds of signatures for deobfuscation methods are processed. Whereas, one has only a `String` parameter, the other takes three `int` parameters. Both signatures return a `String`. However, some obfuscators use methods with more than three parameters, which may also have other types than `String` or `int` and return `Object` instead of `String`. Moreover, it has similar drawbacks as Simplify and JMD – wherein is it required that the inputs of the deobfuscation method call are instructions that return a constant value.

As Figure 2 shows, Dex-Oracle processed all APKs without errors but recovered only 2.5% of all obfuscated strings. Its strict assumptions match only very few deobfuscation methods found in the wild, leading to a low recall. Even those are only a coincidence because the obfuscator, which produced these deobfuscation methods, has various other templates (cf. Section 3.1.3) as also shown in Table 1 with `cn.pro.sdk`.

StringHound was able to process all APKs with a recall of 73.9%. In 26.1% of the cases, the execution environment, surrounding the sliced method, are too complex to be modeled with our default values. However, the high recall confirms the effectiveness of our approach, which does not suffer from the various limitations of the state-of-the-art deobfuscators. Unlike our approach, other deobfuscators do not 'automatically' identify obfuscated strings and deobfuscation methods. To use them, one either needs to know the deobfuscation methods beforehand or must run all methods of the app being analyzed. Such a brute-force approach does not scale to large data sets.

Observation 2. *StringHound outperforms other deobfuscators by orders of magnitude (2.5% vs. 73.9%), and none of the evaluated deobfuscators covers the variability space of existing obfuscators.*

4.2 Findings in the Wild

In this section, we use StringHound to assess how often string obfuscation is used in the wild and for what purposes. Four different sets of APKs are used for our study. The first set consists of 100,000 apps from AndroidZoo [25]. The second set consists of the Top 500 most common apps based on AndroidRank [2]. The third set consists of apps that were available on the Play Store in 2018 and were classified as malicious by at least 10 AV vendors in VirusTotal. Finally, the last set consists of 230 Android malware samples from Contagio [9], containing current and past malware families.

4.2.1 Prevalence of Obfuscated Strings in the Wild

In this section, we measure the prevalence of obfuscated strings in the wild. Therefore, we apply our approach to 100,000 apps from Section 2. To avoid false positives, we exclude all constant strings from our findings and count the remaining ones, which we refer to as newly revealed strings.

Next, we calculated the number of APKs containing newly revealed strings. Based on our study in Section 2, we discovered that only parts of the strings are obfuscated, and some obfuscators hide obfuscated strings in other data structures.

Observation 3. *StringHound enabled us to invalidate the claims of previous studies [17, 37, 56] that less than 5% of the apps contain obfuscated strings, because we discovered that 76% of the 100,000 apps contain obfuscated strings.*

During our analysis, we also measured the proportion of newly revealed strings which were found by the different classifiers. The results indicate that the string classifier detected 28%, and the method classifier 77% of the newly revealed strings.

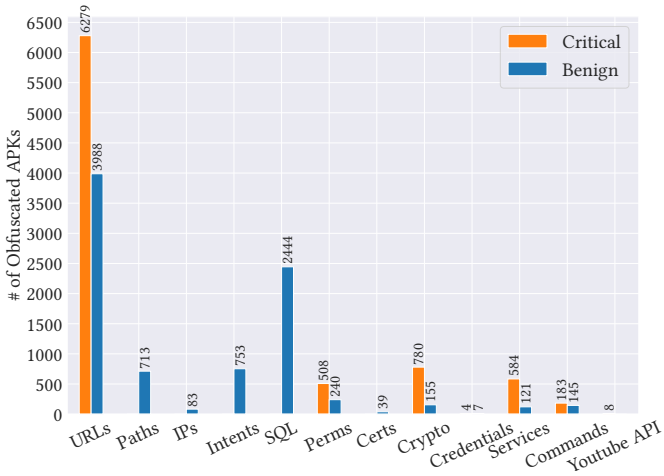

Figure 3. Categories of String Obfuscation in the 100k Apps.

Observation 4. *These findings provide empirical evidence that both classifiers are needed because they have only an overlap of 5% for newly revealed strings. The method classifier (MC) identifies most of the newly revealed strings. However, the string classifier provides at least ($SC_{new} - Both_{new} =$) 23% of newly revealed strings and, thus, it is also necessary for StringHound to achieve a higher total recall.*

4.2.2 Categorization of String Obfuscation

To understand the usage of string obfuscation in the wild, we used regular expressions to categorize all deobfuscated strings in different classes. Altogether, we defined 12 regular expressions for matching URLs, file system paths (Paths), IPs, intents, SQL statements (SQL), permissions (Perms), certificates (Certs), cryptography algorithms (crypto), credentials, system services (Services), commands, and API keys[35]. By using rather strict regular expressions, we prefer precision over recall to avoid false matches, which would lead to a wrong distribution over the categories. We applied the regular expressions to all deobfuscated strings in our data sets (discarding apps without newly revealed strings). Afterward, we counted their matches to quantify the prevalence of each kind of usage.

Figure 3 shows a categorization of the resulting deobfuscated strings in the 100,000 apps. The bar chart is divided into critical and benign apps with obfuscated strings. We identified many critical strings that we found by counting the following facts. First, we identified more HTTP requests than HTTPS for which lead to security issues [53]. Second, developers request permissions but are not aware that these permissions are also used via obfuscated strings by ad libraries to access private data. Third, insecure cryptography algorithms such as DES, AES with ECB mode, or MD2 are still used in obfuscated strings. Fourth, credentials, hidden

in obfuscated strings, are sent via HTTP GET method to login to their services. Fifth, services, requested in obfuscated strings, provide dangerous accesses (e.g., the location of the device). Sixth, rooted phones execute commands, hidden in obfuscated strings, to grant root access. Last, Youtube API keys, hidden in obfuscated strings, can be used to consume the developer's API quotas.

> **Observation 5.** *Using StringHound our analysis of the 100,000 apps reveals that critical usages of URLs, piggybacked permissions, insecure cryptography algorithms, hard-coded credentials, dangerous services, root commands, and API keys are hidden in obfuscated strings.*

4.2.3 Context Analysis of the Categories

While the 100,000 apps contain a large variety of statistical findings, we have no insights into apps that belong to the extreme fields in the Android ecosystem. Therefore, we chose three different data sets to get an understanding of these kinds of fields and the context of StringHound's findings. These data sets consist of the top 500 most installed apps in the Play store, and two malware sets to analyze current (malware 2018) and past (Contagio) obfuscated malware. Figure 4 shows a categorization of the resulting deobfuscated strings. Each bar corresponds to the percentage of APKs from a data set containing at least one deobfuscated string in the given category. Therefore, each category comprises a group of three bars, where each corresponds to one data set.

The first bar shows that 60% of the Contagio malware obfuscates strings, mostly paths (40%), URLs (12%), or intents (5%). A detailed analysis revealed absolute paths of commands trying to open a command shell or of further APK or DEX files hidden in the resources of the app containing the malware's actual payload. One path was used to establish a connection to a Command & Control server (AnserverBot [23]). Furthermore, we found paths to files on the SD card and to DHCP settings, which are exploited by the DroidKungFu2 malware [28]. Our regex for URLs matched locations of browser settings that can be used to build a profile of the underlying mobile phone - some URLs linked to services providing the geolocation of the accessing IP address. We also found URLs to ad networks that profile the user's phone. The regular expression for intents matched an action that resets the default page of the browser to either show pages of ad networks or to track user's behavior. Furthermore, the intent regex discovered an action, which queries the phone number of the mobile phone to reveal the identity of the user. Finally, we also found an action that performs phone calls.

In the malware set from 2018, 35% of APKs use string obfuscation to hide a variety of interactions with the Android operating system. We matched URLs that lead to ad networks, which track the user's interactions and build profiles of users as well as URLs that access the user's calendar and

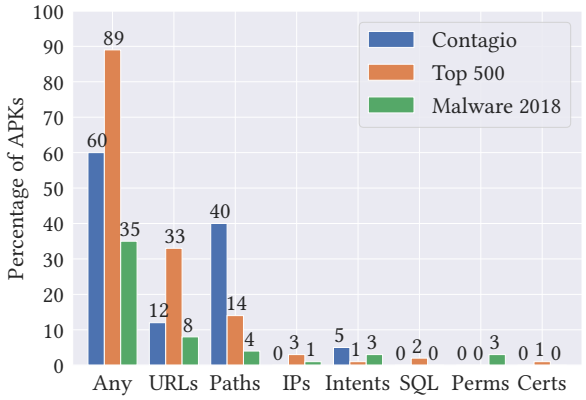

Figure 4. Categories of String Obfuscation in the Wild.

can reveal detailed information of their schedules. The tracking of interactions, in combination with profiling, violates the user's privacy. We also found hidden paths of an APK holding its malicious payload. Moreover, paths to operating-system commands, which access hardware and sensor data to profile a phone, were revealed. Findings regarding intents and permissions indicate that malware uses intents to access functionality to call or send SMS to premium numbers. Additionally, the malware tries to locate or profile a user by accessing personal calendars, accounts, or states of a phone. In comparison to malware from the Contagio set, more recent malware focuses on leakages of private data, causing financial damage to the unknowing user.

> **Observation 6.** *Current malware in the Play store makes less use of string obfuscation (35% compared to 60%) and focuses more on hiding leakages of private data. Without StringHound, one would miss information that is essential to detect remote command execution, even causing financial damage to the user, and leakages of private data in at least 35% of recent malware.*

Surprisingly, string obfuscation is more frequently used in the Top 500 apps than in the 100,000-set of apps (89% vs. 76%), even more frequently than in malware. Our evaluation shows that 33% of the apps use obfuscated URLs. Some of those URLs are used to track users' IDs and IP through an ad network.

These actions directly violate users' privacy. A detailed review of the findings showed that all ad libraries contain obfuscated URLs and paths. We also analyzed how many apps use string obfuscation only in ad- and third-party libraries[5]. This analysis revealed that 63.52% of all obfuscated strings in the Top 500 data set are contained in ad libraries, an additional 10.64% are contained in other libraries, and the remaining 25.84% are in the app itself.

[5]To this end, we filtered our findings by the list of ad-library package names from Section 2 and by a list of common libraries [29].

Observation 7. *String obfuscation is frequently used in all sorts of apps. Ad libraries are responsible for over 63% of these strings. They are used to hide the tracking of users. This result is alarming since neither the user nor the developer of the app is aware of the added functionality. With StringHound the developer could check the content of the used ad library and choose an appropriate alternative.*

Two games for children contained obfuscated privacy violations in the Top 500 data set. We manually analyzed their code and found that they collect and transmit sensitive information on the user's mobile phone. The leaked information includes build-, connectivity-, debug-, runtime-, telephony-, Android version-, and hardware data, which can be used to build a user profile. Code related to data collection is hidden in a stealthy package mixed into the integrated Android support library. The app additionally checks for the `SuperUser.apk`, a package that grants root access to the mobile phone. According to AndroidRank [2], these suspicious apps are installed on at least 20 million devices.

Observation 8. *Virus scanners do not flag suspicious privacy violations. Since, we uploaded the two apps to VirusTotal [55], which showed no findings, besides the usage of dangerous permissions. StringHound allows the analyst to search for all kinds of violations.*

5 Discussion

There are a few limitations of StringHound that will be subject of further consideration in future work.

Driven by the study of obfuscation schemes, which revealed that inter-procedural techniques are currently not used in practice, StringHound uses intra-procedural slicing to recover automatically obfuscated strings. As a result, the slice's execution may fail if it expects values, which differ from our injected defaults. However, this limitation can be addressed by fuzzing the expected values. Given a field or parameter, fuzzing guesses their values by their data-dependencies or using symbolic execution to discover possible value ranges. Obfuscators can use fields and parameters to perform inter-procedural obfuscation. However, to perform it automatically, they need to identify the call order of the fields and parameters. This call-order is not easily identifiable because of the limitation of current call graph analyses for Android. Of course, making StringHound inter-procedural is an obvious alternative, but coping with potential inter-procedural obfuscation schemes is a trade-off between soundness and performance.

If an obfuscator adds random dictionary words to a string, it can eventually evade detection by our String classifier because the proportion of content that is classified as non-obfuscated will increase. However, for this technique to be effective, more than half of a given string would need to consist of non-obfuscated words. During our analysis of obfuscation techniques, we never found more than one dictionary word in obfuscated strings.

Finally, if a new obfuscation technique for strings is used that does not share any commonalities with known techniques, we need to extend the approach with the found technique without training from scratch.

6 Related work

In this section, we discuss four approaches, which could potentially be used for deobfuscation, and studies on the usage of string obfuscation in the wild.

6.1 Potential Deobfuscation Approaches

While different slicing approaches [8, 10, 24, 36] exist that could be modified with much effort to deobfuscate strings; others can be used almost directly. Unfortunately, we could not include the other works [5, 45, 61, 64] in our empirical evaluation because they were not publicly available. We contacted all authors via e-mail, however, without any responses. Additionally, the re-implementation of their tools was also not possible because some parts cannot be reconstructed from the papers. As a result, we only discuss these approaches in the following based on their descriptions.

Harvester [45], *TIRO [61]*, *CredMiner [64]*, and *ARES [5]* combine static and dynamic analysis to extract obfuscated runtime values, including obfuscated strings, from Android malware. All these approaches execute re-bundled code on an emulator using monkey scripts. This re-bundled code is sliced, starting from a fixed set of starting points.

On the contrary, StringHound requires neither re-bundling the app nor an emulator setup that explores all paths with a monkey script. As a result, StringHound can analyze Android and Java applications without searching for the correct combination of events to trigger the deobfuscation. Additionally, our classifiers identify more than a fixed set of starting points.

6.2 Identifying Obfuscated Apps

Several approaches have been proposed to identify whether the content of an (Android) app is changed by an obfuscator [17, 37, 56, 58]. While OBFUSCAN [58] only identifies name obfuscations, the other three approaches [17, 37, 56] can identify whether the code contains obfuscated strings. Wang et al. [56] even infer the used obfuscator. Like our classifiers, all four approaches rely on machine learning techniques to identify whether code is obfuscated or not. However, unlike our approach, they can only detect string obfuscation if all strings in the app are obfuscated. Additionally, they cannot handle obfuscated strings which are represented by byte arrays. We use the token distribution [22] with the Spearman's

correlation to perform a scalable and lightweight similarity measurement. Other approaches, such as those used in clone detection [30], are not suited for obfuscated clones and would require a considerable ground truth for the training of their neuronal networks.

7 Conclusion & Future Work

This paper shows how and why string obfuscation is used in real-world Android and Java apps. We presented StringHound, our approach to identify obfuscated strings and recover their plain text. StringHound significantly improves over state-of-the-art deobfuscation tools. We also presented a large-scale study on the usage of string obfuscation in benign and malicious apps, revealing highly-relevant findings.

We provide empirical evidence that string obfuscation is commonly used across malware, 100,000 apps from Google's Play Store, and various ad libraries. This evidence invalidates statements by previous research, suggesting that string obfuscation is rarely used in practice. By undoing string obfuscation, we revealed abundant problematic string usages in the wild: Critical internet accesses, piggy-backed permissions, insecure usage of cryptography algorithms, hard-coded passwords, and available Youtube API keys. We have found not only malware concealing hidden commands and communication endpoints, but also spyware-like behavior in two apps in the Top 500 set. Our studies have shown that libraries account for a significant amount of obfuscated strings in benign apps. Many findings in the ad libraries reveal serious privacy issues.

We have already mentioned several interesting areas for future work in Section 5. In addition, we will investigate ways to improve StringHound's runtime performance by incorporating library detection [4, 22, 31, 33, 57] and extraction and/or by parallel execution of slices.

References

[1] V Aho Alfred, Sethi Ravi, and D Ullman Jeffrey. 1986. Compilers: principles, techniques, and tools. *Reading: Addison Wesley Publishing Company* (1986).

[2] Androidrank. Accessed: 2019-05-15. https://www.androidrank.org/.

[3] App Brain's Ad Networks. Accessed: 2019-05-15. https://www.appbrain.com/stats/libraries/ad-networks.

[4] 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, 356–367.

[5] Luciano Bello and Marco Pistoia. 2018. ARES: triggering payload of evasive Android malware. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft'18)*. IEEE, 2–12.

[6] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*. ACM, 343–355.

[7] David Binkley and Keith Brian Gallagher. 1996. Program slicing. *Advances in Computers* 43, 1-50 (1996), 1–2.

[8] Yi Chen, Wei You, Yeonjoon Lee, Kai Chen, XiaoFeng Wang, and Wei Zou. 2017. Mass discovery of android traffic imprints through instantiated partial execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. ACM, 815–828.

[9] Contagio Mobile Dump. Accessed: 2019-05-15. http://contagiominidump.blogspot.com/.

[10] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through Differential Analysis.. In *Proceedings of the 2017 Network and Distributed Systems Symposium (NDSS'17)*.

[11] Oracle Naming Conventions. Accessed: 2019-04-26. https://www.oracle.com/technetwork/java/codeconventions-135099.html.

[12] DashO. Accessed: 2019-05-15. https://www.preemptive.com/.

[13] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A Gunter. 2016. Free for All! Assessing User Data Exposure to Advertising Libraries on Android.. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS'16)*.

[14] Dex Oracle. Accessed: 2019-05-15. https://github.com/CalebFenton/dex-oracle.

[15] Dex2Jar Decrypt Strings. Accessed: 2019-05-15. https://sourceforge.net/p/dex2jar/wiki/DecryptStrings/.

[16] DexGuard. Accessed: 2017-10-23. https://www.guardsquare.com/en/dexguard.

[17] Dong, Shuaike and Li, Menghao and Diao, Wenrui and Liu, Xiangyu and Liu, Jian and Li, Zhou and Xu, Fenghao and Chen, Kai and Wang, Xiaofeng and Zhang, Kehuan. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. *Springer* (2018), 172–192.

[18] Michael Eichberg and Ben Hermann. 2014. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP '14)*. ACM, 1–6.

[19] Eric Enslen, Emily Hill, Lori Pollock, and K Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. *IEEE Computer Society* (2009), 71–80.

[20] F-Droid. Accessed: 2019-05-15. https://f-droid.org/.

[21] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 377–396.

[22] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. CodeMatch: obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 638–648.

[23] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. 2012. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 281–294.

[24] Johannes Hoffmann, Martin Ussath, Thorsten Holz, and Michael Spreitzenbarth. 2013. Slicing Droids: Program Slicing for Smali Code. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*. ACM, New York, NY, USA, 1844–1851.

[25] Médéric Hurier, Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. On the lack of consensus in anti-virus decisions: Metrics and insights on building ground truths of android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'16)*. Springer, 142–162.

[26] Java bytecode analysis/deobfuscation tool. Accessed: 2019-05-15. https://github.com/contra/JMD.

[27] David Kahn. 1996. *The Codebreakers: The comprehensive history of secret communication from ancient times to the internet.* Simon and Schuster.

[28] Richard Killam, Paul Cook, and Natalia Stakhanova. 2016. Android malware classification through analysis of string literals. *Text Analytics for Cybersecurity and Online Safety (TA-COS)* (2016).

[29] Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. An Investigation into the Use of Common Libraries in Android Apps. In *Technique Report*.

[30] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. IEEE, 249–260.

[31] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: scalable and precise third-party library detection in android markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE'17)*. IEEE, 335–346.

[32] Robert Lyda and James Hamrock. 2007. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy* 5, 2 (2007), 40–45.

[33] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, 653–656.

[34] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2016. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433* (2016).

[35] Michael Meli, Matthew R McNiece, and Bradley Reaves. 2019. How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories.. In *NDSS*.

[36] Luis Menezes and Roland Wismüller. 2017. Detecting information leaks in Android applications using a hybrid approach with program slicing, instrumentation and tagging. In *Security Technology (ICCST)*. IEEE, 1–6.

[37] Mirzaei, O and de Fuentes, JM and Tapiador, J and Gonzalez-Manzano, L. 2018. AndrODet: An adaptive android obfuscation detector. *Future Generation Computer Systems* (2018).

[38] Leann Myers and Maria J Sirois. 2004. Spearman correlation coefficients, differences between. *Encyclopedia of statistical sciences* 12 (2004).

[39] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. 2018. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS'18)*.

[40] Allatori Java Obfuscator. Accessed: 2019-05-15. http://www.allatori.com/.

[41] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. 2017. Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps.. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS'17)*.

[42] Practical Cryptography. Accessed: 2019-05-15. http://practicalcryptography.com/cryptanalysis/.

[43] ProGuard manual. Accessed: 2019-05-31. https://www.guardsquare.com/en/products/proguard/manual/usage.

[44] J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1, 1 (1986), 81–106.

[45] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. In *NDSS*.

[46] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, and Christian Kreibich Phillipa Gill. 2018. Apps, trackers, privacy, and regulators. In *25th Annual Network and Distributed System Security Symposium, NDSS*, Vol. 2018.

[47] Repository for Code & Data. Accessed: 2019-08-21. Will-be-available-after-acceptance.

[48] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)* 49, 1 (2016), 4.

[49] Simplify. Accessed: 2019-05-15. https://github.com/CalebFenton/simplify.

[50] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. 2016. What Mobile Ads Know About Mobile Users.. In *Proceedings of the 2016 Network and Distributed Systems Symposium (NDSS'16)*.

[51] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. 2012. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST'12)*, Vol. 10.

[52] Stringer Java Obfuscator. Accessed: 2019-05-15. https://jfxstore.com/.

[53] Protecting users with TLS by default in Android P. Accessed: 2019-11-22. . https://android-developers.googleblog.com/2018/04/protecting-users-with-tls-by-default-in.html

[54] Timothy Vidas and Nicolas Christin. 2014. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 447–458.

[55] VirusTotal. Accessed: 2019-05-15. https://www.virustotal.com/.

[56] 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*. 154–164.

[57] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. 2018. Orlis: Obfuscation-resilient library detection for Android. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft'18)*. IEEE, 13–23.

[58] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*. ACM, 222–235.

[59] WhatsApp. Accessed: 2019-05-15. https://play.google.com/store/apps/details?id=com.whatsapp.

[60] Michelle Y Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware.. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS'16)*, Vol. 16. 21–24.

[61] Michelle Y Wong and David Lie. 2018. Tackling runtime-based obfuscation in Android with TIRO. In *27th USENIX Security Symposium (USENIX Security'18)*. 1247–1262.

[62] Zelix KlassMaster. Accessed: 2019-05-15. http://www.zelix.com/.

[63] Qingchuan Zhao, Chaoshun Zuo, Giancarlo Pellegrino, and Li Zhiqiang. 2019. Geo-locating Drivers: A Study of Sensitive Data Leakage in Ride-Hailing Services.. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS'19)*.

[64] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. 2015. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, 23.

[65] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *IEEE Symposium on Security and Privacy (SP'19)*.

A Appendix

In this section, we describe our slicing algorithm, the runtime performance of StringHound, the features of our string classifier, and give a detailed list of the regular expressions which we used to categorize deobfuscated strings in the wild.

A.1 Our targeted Slicing

Our slicing algorithm (cf. Figure 5) is inspired by classical slicing algorithms (cf. Binkley et al. [7]), and implemented using OPAL with definitions (cf. Aho et al. [1]) of the functions defined in Table 2.

Given a method along with its control-flow graph (CFG), a *LoI*, and a slicing criterion s_{crit}, the algorithm initializes the worklist W (Line 2 of Figure 5) with s_{crit}. For each instruction in W (Line 6) that is not already part of the slice (Line 8), the algorithm performs the following steps:

1. Add the current instruction *currInstr* to the slice (Line 9).
2. In the *backward step* (Line 10), determine the set D of all definition sites related to *currInstr*, i.e., D consists of instructions that initialize variables used by *currInstr*.
3. Also in the *backward step*, we determine the set $cd_{currInstr}$ of instructions on which the current instruction is control dependent on (Line 11). From this set, remove instructions (cd_{crit}) that could prevent the execution of s_{crit}. This backward step adds instructions to W that (in)directly affect s_{crit}. With this, we include conditional instructions which do not control the execution of the criterion itself (e.g., Line 3 in Figure 7). This step is required to, e.g., ensure that loops that manipulate byte arrays are completely added to the slice. If the backward step adds instructions that define a new reference-typed variable, i.e., an object (e.g., Line 2 in Figure 6), we perform an additional forward step to include those instructions in W that potentially affect the state of the object after its initialization and which are relevant w.r.t. the *LoI* (e.g., Line 4 in Figure 6). Hence, we add only instructions that are still backward reachable from the *LoI*.
4. In the *forward step* (Line 12, 13), determine the set of all instructions U that use a variable defined by (*currInstr*) and which are backward reachable from the *LoI*. This step includes all instructions that potentially mutate the state of the defined variable, e.g., instructions that fill an array with actual values or call a method of the object.
5. In the last step (Line 14), update W with the set of instructions (a) on which *currInstr* is control dependent ($cd_{currInstr}$), (b) that use the defined variable (U), and (c) that initialize the variables used by *currInstr* (D).

Table 2. Definitions of Helper Functions for the Algorithm (inst=instruction)

def	$Instr \rightarrow \mathcal{P}(Var)$	variables defined by an inst
use	$Instr \rightarrow \mathcal{P}(Var)$	variables used by an inst
du	$Var \times Instr \rightarrow \mathcal{P}(Instr)$	definition-use insts
ud	$Var \times Instr \rightarrow \mathcal{P}(Instr)$	use-definition insts
cd	$Instr \rightarrow \mathcal{P}(Instr)$	transitive control dependency insts
br	$Instr \rightarrow \mathcal{P}(Instr)$	set of backwards reachable insts

Input: m a method with a body
I the Instructions of the method m
g the CFG of m where each $i \in I$ corresponds to one node $n \in N$ of g
$LoI \in I$ the location of interest
$s_{crit} \in I$ the slicing criterion
Output: $N_{slice} \subseteq I$

```
1  N_slice := {}
2  W := {s_crit}
3  cd_crit := cd(s_crit)
4  br_LoI := br(LoI)
5  while W ≠ ∅ do
6      currInstr := head(W)
7      W := W \ currInstr
8      if currInstr ∉ N_slice then
9          N_slice := N_slice ∪ { currInstr }
10         D := {d | x ∈ use(currInstr) ∧ d ∈ ud(x, currInstr)}
11         cd_currInstr := cd(currInstr) \ cd_crit
12         U := {u | x ∈ def(currInstr) ∧ u ∈ du(x, currInstr)
13                ∧ u ∈ br_LoI}
14         W := W ∪ D ∪ cd_currInstr ∪ U
15     end
16 end
```

Figure 5. Slicing Algorithm

For illustration, assume the *LoI* being processed is the `useString` call (Line 11), the only string parameter `s` (the `int` parameter is ignored) is defined by the result of the call `sb.toString()` (Line 9); hence, this call is our slicing criterion. On the contrary, instructions that load string constants are not considered as slicing criteria, e.g., Line 6 is a *LoI*, but the instruction that loads the string constant `"D"` is not a slicing criterion. The rationale is that in such cases, we are sure that no deobfuscation happens before reaching the *LoI*. In Figure 6 such *LoI*s are pointed at by dashed arrows.

For example, in the following code snippet:

```
1  String msg = simCountryIso().equals("US") ? %US() : INT();
2  invoke("+01234", msg);
```

there are two sources of *msg* in Line 2 corresponding to the two branches of the tertiary operator in Line 1, which load either `"US()"` or `"INT()"`—StringHound would start the slicing process twice.

A.2 Slicing String Usages

We illustrate the algorithm in the examples in Figure 6 and 7. Assume that the call `useString(s,2)` (Line 11 in Figure 6) is the *LoI* and `toString()` (Line 9) is the s_{crit} (Step 1).

The first slicing step (Step 2 in Figure 6) determines as part of the backward step the definition site of the object on which `toString()` is called, i.e., `sb`.

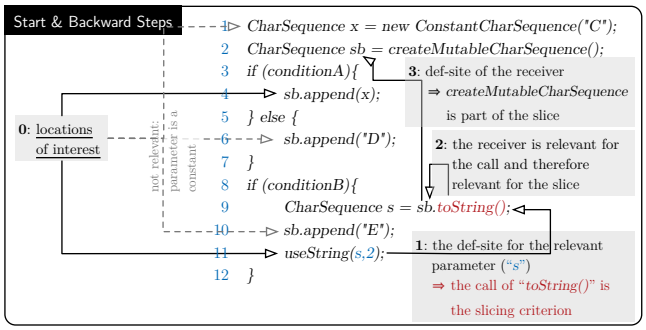

Figure 6. Example of the Slicing Process for the Parameter `s` of `useString`—Showing *LoI*s and First Backwards Step

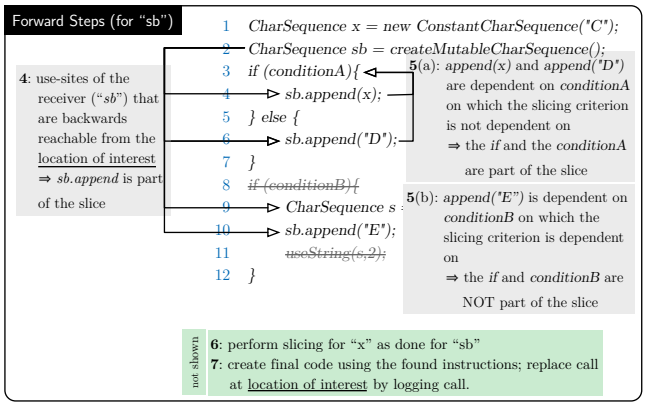

Figure 7. Example of the Slicing Process for the Parameter `s` of `useString` — Showing the Necessary Forwards Steps

Hence, `createMutableCharSequence` (Line 2, Step 3) is added to W. The if-condition (Line 8) is not added to the W because it would potentially prevent the execution of the slicing criterion (s_{crit}). Next, we perform a forward step concerning Line 2 (`createMutableCharSequence` does not use variables; hence, we do not perform a backward step.) Step 4 (cf. Figure 7) identifies all use-sites of `sb` and, since all of them (Lines 4, 6, 9, and 10) are backward reachable from the *LoI*, we add them to W. In this case, we do not add Line 10 to the slice, because the string returned at Line 9 cannot be mutated afterward in Java. However, our algorithm does not have such knowledge and therefore, conservatively adds it—as part of the forward step (Line 10).

Next, the algorithm processes the append calls in Lines 4 and 6 as follows (Step 5(a)). Given that no local variable is defined, there will not be a forward step; however, in the first `append` call (Line 4), we use `x` and, therefore, add the defining instruction (Line 1) to W. Additionally, the `if`-instruction in Line 3 is added to W, because both `appends` are control dependent on it, but not s_{crit} (Step 5(a)). When we process the `append` call in Line 10 we see that the `if`-condition (Line 8) would possibly prevent the execution of s_{crit} and thus do not add it (Step 5(b)).

To recap, the resulting slice is the entire code from Figure 7, except for Lines 8 and 11 in Figure 7 (which are crossed out). Given our special targeted slicing, we evade the techniques *Serialized Object, Byte Arrays, Switch Statements, Stream Transfer, and Two Keys* in Table 1.

Table 3. Feature List for the Detection of Obfuscated Strings

Name	Description
27 Known Formats,	User Agents, URLs, Character set of regular expressions, Network protocols (e.g., WiFi), Common OS commands, JSON format, Encodings (e.g., UTF-8), E-Mail format, DTD, HTML Colors, Class Path Format, SQL Queries, Keywords for seven programming languages, country names, XML format, IP format, HTTP state format, Multiple Date formats, Numeric formats, Cryptographic primitives, Mobile phone brands, HTML special characters (e.g., uuml), String-encoded certificate format, String-encoded Android certificate format, Private/Public key format, String signatures of social network apps, String-encoded images (e.g. JPEG),
Chi-squared Test	Tests if all chars in the given string are equally, distributed indicating a random distribution.
Average Distribution	The average distribution which is close to the Gaussian distribution for plain strings,
Normalized Entropy	The normalized entropy of the strings,
AndroDet[37]	Number of equals, Number of dashes, Number of slashes, Number of pluses, Sum of repetitive characters,
Compression Rate	The rate of the GZIP compression,
Cryptography Library	The string is used in a known crypto library,
Dictionary Words	The shortest word length, the largest word length, the number of words, the number of unique words from a multiple language dictionary.
String Characteristics	Number of vocals, Number of consonants, Number of digits, Number of characters, Number of unique characters, Number of non letters, Maximum number of consecutive characters, Maximum occurrences of the same character

A.3 Runtime Performance

To evaluate the runtime performance of StringHound, we measured the average runtime per APK and per slice, when running StringHound on the Top 500, and the two data sets from Section 4.2. While the first measure shows how long our approach needs for APKs of different sizes, the second one can be used to approximate the analysis time for a given APK. All performance measures indicate that StringHound is fast and ready for practical use.

Figure 8 shows the average runtime per APK. Thereby, each bar corresponding to one data set and is split into the time needed (a) for loading the analysis, (b) executing the String Classifier, (c) executing the Method Classifier, and (d) building and executing slices. One can see, processing the Top 500 data set needs up to 20 times more on average per APK than processing the Contagio data set. The reason for this high discrepancy is a large amount of library code in the APKs of the Top 500 data set. As mentioned in Section 4.2, 74.16% of the obfuscated strings are found in libraries, and these are up to 14 times larger in code size than APKs from the Contagio data set. The time taken to analyze such apps can be reduced by employing tools that separately analyze the library code and reuse these analysis results. Another observation is that across all three data sets, slicing consumes most of the execution time. Hence, improving the performance of the slicing would speed up the entire analysis.

We calculated the mean, median, and also the 95%-quantile for each slice of all three data sets, and all of them are below 250 ms. Thus, we conclude that building and executing a single slice takes on average less than 250ms. Given the observation that slicing consumes most of the execution time and also the execution of a single slice takes less than 250 ms, the only improvement to speed up the performance is to parallelize the building and execution of single slices.

A.4 Feature List of the String Classifier

In Table 3, all features, which are used in the String Classifier to decide whether a string is obfuscated or not, are a list. While regular expressions represent most of the features, the words we match with dictionary support must first be parsed from the strings. However, not all words can be easily matched because some languages (e.g. Chinese) do not use separators (e.g. white spaces). Further, some strings contain names of code elements with concatenated words (e.g. `getLength`). To match words without separators, we use two different word splitting approaches:

Lucene's `ICUTokenizer` for words from multiple languages that do not use separators and Samurai [19], which splits identifiers by camel case and frequent words.

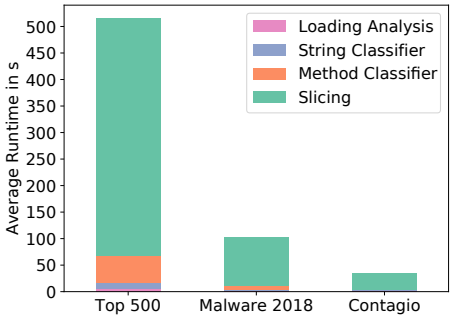

Figure 8. Average Runtime for Top 500 and the Two Malware Data Sets

Table 4. Regular Expressions for the Evaluation of Deobfuscated Strings in the Wild.

Name	Regex
URL	\w+://[^/\"]+.*
IP	.*\b([0-9]{1,3}\.){3}[0-9]{1,3}\b.*
Paths	/\w+[\./].+
Intents	android.intent\..*
SQL	.*(select.*from | update.*set | insert into | delete from | create table | drop table | truncate table).*
Certificates	MII.+
Permissions	android.permission\..*
Youtube API Key [35]	AIza[0-9A-Za-z\-_]{35}
Cryptography algorithms	MD2|MD5|SHA\-?1|ECB|DES

A.5 Pattern List for the Categorization

Table 4 shows the regular expressions we used to categorize deobfuscated strings. Our regex for URLs is not limited to the typical HTTP(S) form but also matches any scheme, such as content URLs. Furthermore, a pattern for IPs is used to match non-URL related communication. The regex for paths describes absolute paths of the Android operating system matching not only directories and file paths but also absolute paths to executables. The patterns for intents and permissions match Android's standard definitions. The regular expression for SQL statements matches strings with common keywords for querying and manipulating tables. Finally, certificates are identified by the Base64 encoded first three characters, which are used as a prefix for certificates.