DeStroid – Fighting String Encryption in Android Malware

Daniel Baier, Martin Lambertz
Fraunhofer FKIE

Abstract

String encryption is a popular technique to obfuscate the functionality, inner workings, and goals of Android apps. Especially malicious apps use this technique to thwart automatic and manual analyses. Typically, a human analyst has to manually identify the decryption routines and afterwards use these routines to decrypt the strings contained in an app. This is a time-consuming and tedious task. What is more, it has to be carried out potentially for every new malware version as the authors frequently modify their techniques.

We analyzed the Android malware corpus of Malpedia [1] and found that string encryption is used in more than half of the samples. This demonstrates that string encryption is still a prevailing obfuscation method nowadays.

In this paper we present DeStroid, an approach to fully automatically decrypt obfuscated strings from Android apps. We focus in particular on current Android malware using advanced string encryption techniques and show that DeStroid outperforms all publicly available string deobfuscation approaches.

1 Introduction

Android is not only the predominant platform when is comes to smartphones and tablets, it is also frequently used on smartwatches, TVs, and other Internet of Things (IoT) devices. Recently, Google announced that "there are over 2.5 billion active Android devices" [2]. The popularity of the Android platform is also reflected by the number of available apps: for the first quarter of 2019 the number of apps in the Google Play Store has been estimated at 2.6 million [3]. Unsurprisingly, this popularity makes the Android platform an attractive target for attackers of all kinds.

These attackers commonly use malicious apps (i.e. malware) for their purposes. Hence, the effective and efficient analysis of suspicious apps is vital from a defenders point of view. Analysts are typically interested in the functionalities and goals of a malware. One of the frequently used starting points during an analysis are the strings used in an app. Hardcoded domains names, method names used for reflection, strings used as method arguments, and so forth can already reveal certain functionalities and intents of the malware. If the strings are sufficiently unique, they would to some extent even be suitable for a detection or identification of a suspicious app.

To prevent this and to impede manual and automated analysis, malware authors employ a variety of obfuscation techniques. A prevalent technique is *string encryption* [4], which is also known as string obfuscation, value encryption, or literal encryption. Here, the strings an app uses are not present in plain text in the Android package (APK), but only in an encrypted form. To be able to use the encrypted strings, the app contains a decryption routine which decrypts the strings either right after the program start of just before a certain string is needed. This is referred to as the deobfuscation part of the APK.

Fig. 1 emphasizes how string encryption complicates the analysis of an app. Part 1a shows a decompiled excerpt of the malware *Backdoor.Android.OS.Triada 2016* which uses string encryption. The bold parts show where string encryption is used to obfuscate the keys used in the JSON object constructed in the method. In contrast, Fig. 1b shows the same excerpt but with the string encryption resolved. Obviously, the second example is much easier to understand. Even without resolving the names of the methods that fill the values, we can easily deduce which information will be contained in the final JSON object.

```
public static JSONObject m64b(Context context) {
 JSONObject jsonObject = new JSONObject();
 try {
   jsonObject.put(C0012m.m56a(C001b.f17I), C0012m.m68c(context));
   jsonObject.put(C0012m.m56a(C001b.f18J), C0012m.m71d(context));
   jsonObject.put(C0012m.m56a(C001b.f20L), C0012m.m49a());
   jsonObject.put(C0012m.m56a(C001b.f21M), C0005f.m37b(context));
   jsonObject.put(C0012m.m56a(C001b.f19K), TextUtils.isEmpty(C0012m.m61b())
                                                ? "channn" : C0012m.m61b())
                                                      ;
   jsonObject.put(C0012m.m56a(C001b.f22N), C0012m.m76f(context));
 } catch (Exception e) {}
 return jsonObject;
}
```

(a) with string encryption

```
public static JSONObject m64b(Context context) {
 JSONObject jsonObject = new JSONObject();
 try {
  jsonObject.put("imei", C0012m.m68c(context));
  jsonObject.put("imsi", C0012m.m71d(context));
  jsonObject.put("uuid", C0012m.m49a());
  jsonObject.put("net", C0005f.m37b(context));
  jsonObject.put("channel", TextUtils.isEmpty(C0012m.m61b())
                                ? "channn" : C0012m.m61b());
  jsonObject.put("existPackages", C0012m.m76f(context));
 } catch (Exception e) {}
 return jsonObject;
}
```

(b) with string encryption resolved

Figure 1: Decompiled excerpt of a sample of the *Backdoor.Android.OS.Triada 2016* malware.

To find out how often string encryption is used in real-world malware, we manually analyzed 96 different Android malware families. In $52\,\%$ of the families we found string encryption being used. Since the actual implementation of the string encryption differed between the samples, we established a taxonomy of string encryption methods used in Android malware and classified the techniques we encountered during our analysis. Moreover, we provide the labelled dataset containing the APKs of the malware samples we analyzed. The labelling includes the class of the string encryption the sample uses, the expected number of deobfuscated strings, as well as the information where to find the deobfuscation part in the decompiled code. Hereby, we provide a ground truth regarding the string encryption used in the malware samples we analyzed. This ground truth is not only used in our evaluation, but it can also be used by the scientific community for future research and evaluations.

Furthermore, we present DeStroid, our approach to fully automatically decrypt obfuscated strings from Android apps. DeStroid is designed to decrypt strings even from highly obfuscated state-of-the-art malware samples which use reflection or hide their strings in several locations within the app. Our approach combines program slicing with code generation and dynamic execution.

In our evaluation, DeStroid was able to deobfuscate 44 of the 79 samples ($55\,\%$) of the samples completely, i.e. it deobfuscated all of the expected string from the samples correctly. 10 other samples were deobfuscated to a degree of $\geq\ 50\,\%$ and $<\ 100\,\%$. Finally, in 8 cases DeStroid was able to decrypt $<\ 50\,\%$ of the strings. These decryption rates can be improved if either a specific encrypted string of interest or the decryption routine is provided. We compared the results of DeStroid with all publicly available string deobfuscation approaches and found that our approach outperforms all of them by far.

In all cases DeStroid took less than 3 minutes per app to deobfuscate the strings in an app. This makes it suitable for large-scale analyses of Android apps.

Both, the implementation of our approach and the labelled dataset of Android malware samples will be made publicly available [5, 1].

In summary, our paper makes the following contributions:

- We provide a taxonomy of string encryption implementations and their usage in current Android malware families.
- We provide a labelled dataset of Android malware samples, including the APKs, the expected deobfuscated strings, and where to find the deobfuscation part in the decompiled code.
- We propose DeStroid, an approach to fully automatically deobfuscate encrypted strings from Android apps, which outperforms all publicly available deobfuscation approaches.

The remainder of the paper is structured as follows: In Section 2 we present a taxonomy of string encryption techniques used in Android malware and the classification of the samples we analyzed with respect to this taxonomy. Section 3 summarizes the related work in the field of Android string decryption. Section 4 introduces the architecture and the concepts behind DeStroid and its components. Next on, Section 5 reports on the evaluation based on the generated ground truth and discusses its results. Then, in Section 6 the limitations of our approach and possible future works are discussed. Finally, Section 7 concludes this paper.

2 A Taxonomy of String Encryption Techniques

There are different possible ways to implement string encryption. In this section, we provide an overview of the methods we observed in the Android malware samples present in the Malpedia [1] corpus. We manually analyzed the samples and identified their string deobfuscation routines. Based on this analysis, we were able to characterize three major string encryption classes, which will be described in more detail later on in this section. Note that our taxonomy focuses not on the actual string encryption algorithm but on how they are implemented. That is, our taxonomy does not divide classes based on whether the string has been obfuscated using XOR, AES, or DES, but rather where and how the deobfuscation routine is implemented.

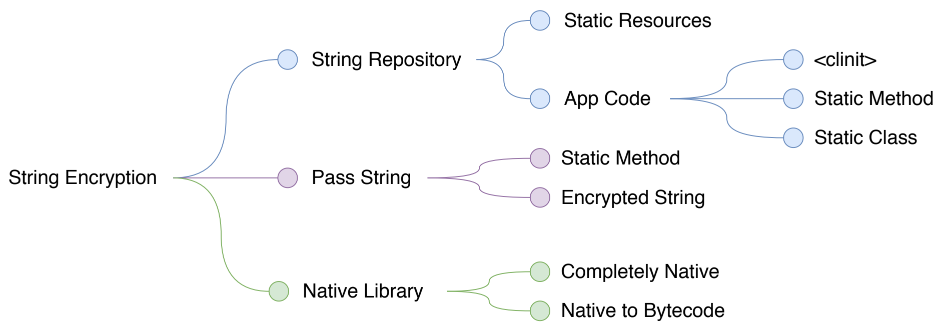


Figure 2: Taxonomy of Android string encryption techniques.

At the time of writing, Malpedia contains 96 different Android malware families. We favored Malpedia over other datasets such as the Android Malware Dataset (AMD) [6] or Android PRAGuard [7], because it is standardized, maintained, and the most up to date dataset available. Moreover, as Malpedia contains multiple versions of malware samples seen over time, it allows for the observation of evolving behaviors, such as obfuscation techniques, which possibly change with a new generation of a malware family.

In our analysis of the malware samples of the Malpedia corpus, we found that 50 out of the 96 families made use of string encryption. Most of the samples used just a single string encryption technique. Still, we also encountered samples using two or more different techniques. Moreover, some samples used different routines to implement the same string encryption technique (cf. Table 4 for details).

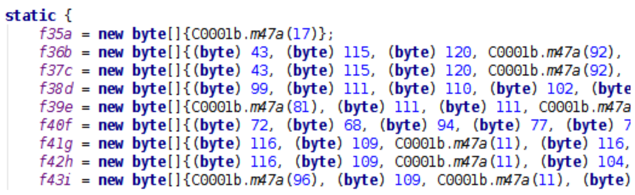
Fig. 2 visualizes our taxonomy. We distinguish between three main classes: *String Repository* (SR), *Pass String* (PS), and *Native Library* (NL). Due to space limitation, we can not provide a detailed descriptions of all of the single sub-classes. Please refer to the online version of our taxonomy [5] for a more elaborate description including code examples.

In the following subsections, we provide a description of the top-level string encryption classes we found during our analysis of the malware samples contained in the Malpedia corpus. The *String Repository* class contains all string encryption techniques where the encrypted strings are centrally provided and the deobfuscation part queries this central location for the values it needs. On the other hand, *Pass String* refers to implementations where the encrypted strings are directly passed to the decryption routines. The *Native Library* method indicates that the encrypted strings as well as their corresponding decryption routines are part of the native code. Access to the decrypted data structures is provided only through the Java Native Interface (JNI, cf. [8]). Fig. 3 illustrates these different string encryption types. In this figure, ① indicates an encrypted value and ② the appropriate decryption routine. ③ and ④ denote decrypted values. For the *Native Library* method, ③ depicts the decrypted value passed from the native code to the bytecode and ④ the provision of the decrypted strings in the context of the bytecode.

Note that our taxonomy is agnostic with regard to the representation of the encrypted strings. The strings can either be encrypted ASCII or UTF-8 strings or a byte, int, or char array, which is later—usually after the decryption—transformed into a regular string.

2.1 String Repository

This class of string encryption has all of the encrypted strings stored in a table-like data structure. There can be one central String Repository for the entire app or there can be a String Repository per class. For performance reasons, the String Repositories are typically provided in static fields and, therefore, get initialized in the static class initialization method `<clinit>` (cf. [9, 10]). Fig. 4 shows encrypted strings as byte arrays inside the static class initializer of a decompiled *Backdoor.Android.OS.Triada 2016* sample.



```
static {
    f35a = new byte[]{C0001b.m47a(17)};
    f36b = new byte[]{(byte) 43, (byte) 115, (byte) 120, C0001b.m47a(92),
    f37c = new byte[]{(byte) 43, (byte) 115, (byte) 120, C0001b.m47a(92),
    f38d = new byte[]{(byte) 99, (byte) 111, (byte) 110, (byte) 102, (byte
    f39e = new byte[]{C0001b.m47a(81), (byte) 111, (byte) 111, C0001b.m47a
    f40f = new byte[]{(byte) 72, (byte) 68, (byte) 94, (byte) 77, (byte) 7
    f41g = new byte[]{(byte) 116, (byte) 109, C0001b.m47a(11), (byte) 116,
    f42h = new byte[]{(byte) 116, (byte) 109, C0001b.m47a(11), (byte) 104,
    f43i = new byte[]{C0001b.m47a(96), (byte) 109, C0001b.m47a(11), (byte)
```

Figure 4: String Repository implemented as `<clinit>`-method from a decompiled sample of the malware *Backdoor.Android.OS.Triada 2016*.

In some cases the decryption is already carried out inside the method which initializes the String Repository. This basically means that the String Repository data structure never gets populated with encrypted values, but straight with the decrypted values.

Apart from the static class initializer, other resources are used as string repositories, too. For instance, a file inside the `assets` folder containing the encrypted strings or resource files inside the `res` folder. A detailed list of resource locations used by the malware families we analyzed can be found online [5].

Common to all String Repository implementations is the fact that either the already decrypted values or the encrypted values, which are then passed to their decryption routines, are pulled from that table. The left side of Fig. 3 illustrates this basic idea of this string encryption type.

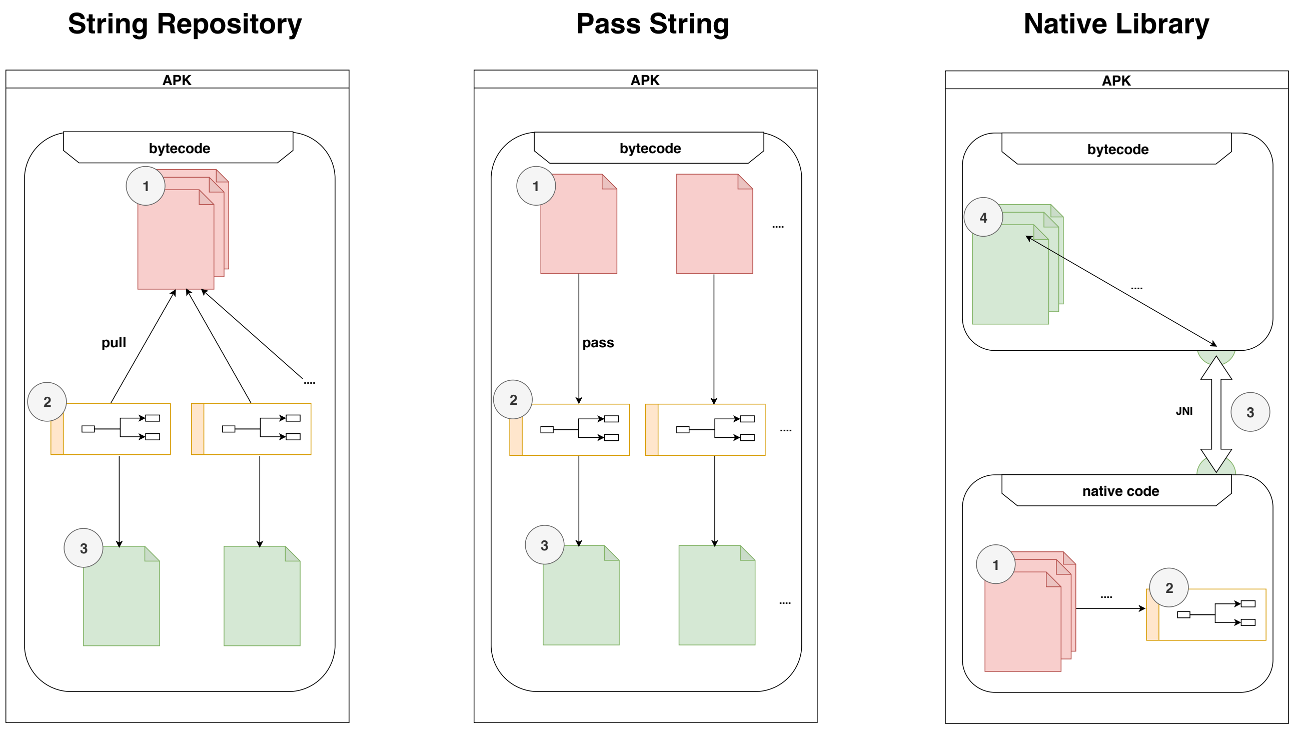


Figure 3: Basic method of operation of the top-level string encryption classes.

2.2 Pass String

In this approach, the string decryption part is invoked whenever a string has to be decrypted, with the encrypted strings being passed directly to the deobfuscation routine. A simplified example of this method is shown in Fig. 5. Here, the two encrypted strings `v0` and `v1` are directly passed to the decryption routine and the decrypted string is then stored in the register `v0`.

```
const-string v0, "TWF5IHRoZSBiaXRzIGFuZAo="
const-string v1, "IGJ5dGVzIHdpdGggeW91Lgo="
invoke-static {v0, v1}, La/b/c;->a(
    Ljava/lang/String;Ljava/lang/String;)
    Ljava/lang/String;
move-result v0
```

Figure 5: Pass String example in smali code.

We differentiate between two subclasses: *encrypted string* and *static method*. The only difference between these two classes of Pass String are the levels of indirection to resolve until the encrypted strings are passed to the deobfuscation routine. These indirections have several reasons. One is the conversion of the encrypted string into the data structure its corresponding decryption routine expects. Another form of indirection is the invocation of, for instance, a static method which only returns the encrypted string as parameter for the decryption routine.

2.3 Native Library

The third class we identified to implement string encryption is *Native Library*. Here, encrypted strings as well as their corresponding decryption routines are part of the native code ($\textcircled{1}$ and $\textcircled{2}$ on the right of Fig. 3). After the deobfuscation part has been executed, the JNI provides access to the decrypted data structures ($\textcircled{3}$) and fills a *String Repository* like data structure in the bytecode context ($\textcircled{4}$). Unique to the native library approach is the complete implementation of the string encryption into the native part of an Android application.

Theoretically, a native library could also be used as a form of a *String Repository* containing the encrypted strings. The same holds for *Pass String*. Here, only the decryption routine could be implemented in the native library and executed whenever an encrypted string is passed to it trough the JNI. However, we never encountered these implementations of string encryption during our analysis.

In our dataset the Native Library method was used only in the *flexispy* malware family [11]. Inside the native library multiple functions build and decrypt the encrypted strings. These decrypted strings are provided to the app to fill a byte array with the decrypted content.

2.4 String Encryption in Android Malware

Table 1 provides an overview of the classification of the malware families we analyzed with regard to our taxonomy. As already mentioned, there are 46 families which do not use string encryption at all, while 50 families do. Among these families *String Repository* and *Pass String* are the most frequently used methods. To be more precise, the `<clinit>`-method is the preferred

String Repository implementation and passing the decrypted strings directly to its decryption routine is the preferred *Pass String* implementation. *NL*, on the other hand, has been rarely used. The table also reflects the fact that there are some families which use more than one technique. This is why the numbers do not add up to 50.

A more detailed listing of the classification can be found in the Appendix in Table 4.

3 Related Work

While the field of Android string encryption is still mostly uncharted, there are a few noteworthy works related to string encryption in Android apps. First, there are approaches focusing on the identification of encrypted strings or encryption and decryption routines; second, there are approaches trying to actually resolve string (and other) deobfuscations.

3.1 Identification of Encrypted Strings

Dong et al. [4] conducted a large-scale survey of obfuscation techniques implemented in Android apps. They used apps from various app stores and malware databases to derive machine learning models or signatures to detect the four most popular obfuscation techniques. To detect string encryption, they built a machine learning model based on the commercial obfuscators *DashO* and *DexProtector*. Moreover, they modified existing approaches to detect cryptographic function to be better suited for the Android platform. They found that string encryption is mostly present in malware and not in benign apps. Moreover, they identified that malware samples typically use more than one cryptographic function. Yet, it is clear whether the cryptographic functions identified are, in fact used, for string encryption or for different purposes. Furthermore, it is not obvious whether the model used to detect encrypted strings works equally well for strings that have not been encrypted with one of the two commercial obfuscators.

Another approach was used in AndrODet[12] which focused on certain features of strings inside an APK which is then used for their online learning system through Data Stream Mining (henceforth DSM). However, this approach focuses only on encrypted strings which are already in the constant pool of an APK. Hence, encrypted strings which are stored as other data types such as character or byte arrays are not considered.

Instead of only considering the encrypted strings, Kühnel et al. [13] focuses on identifying the cryptographic routines among others inside an APK. Hereby they identify whether or not encryption is used there. Although they are able to identify string decryption routines with this, they also identify encryption routines with other purposes (e.g. encrypted network communication).

3.2 Deobfuscation Approaches

Dex-Oracle [14], JMD [15], Deobfuscator [16] as well as the approach of Moses and Mordekhay [17] are using pattern recognition techniques to identify string obfuscation in static analyses. The encrypted strings, together with a list of function calls and argument values, are then passed to a dynamic module. This dynamic module executes the provided function calls and returns the results to a module, which then patches the app code using the decrypted strings. All of these approaches work well for obfuscation techniques for which a corresponding pattern is known and implemented. They are, however, not able to detect and resolve techniques that cannot be described using a static pattern.

Simplify [18, 19] uses virtual execution to resolve different obfuscation types an app uses. A context sensitive graph is generated representing every possible execution path including all possible registers as well as class states for each execution of each instruction. Afterwards, the graph is analyzed and several optimizations are applied such as constant propagation. Depending on the sample, especially on huge samples, this can result in what is known as the path explosion problem. Moreover, keeping the virtual execution environment up to date with current changes to Android is quite cumbersome and error prone.

Another approach is the monitoring of executed DEX bytecode as it is done in DexMonitor [20]. To reduce the scope of traced instructions it uses selective monitoring to monitor only `invoke` and `return` instructions. This way, all method invocations and their return values are traced which eventually yields the decrypted strings. However, only executed code paths of the app can be monitored. In addition, this approach is susceptible to time bombs deliberately included to impede dynamic analyses.

TIRO [21] is able to detect and deobfuscate language-based and runtime-based obfuscations via dynamic instrumentation. The dynamic instrumentation is based on a prior static analysis to indicate what has to be instrumented. Harvester [22] uses static code slicing to execute paths leading to specific code locations, such as reflection invocations, and is able to deobfuscate all obfuscated values encountered in those executed paths. However, both approaches focus only on certain parts of an application and could, therefore, miss obfuscated strings. Furthermore, both approaches are not fully available.

4 DeStroid

In this chapter, we introduce DeStroid, our approach to automatic string decryption. Fig. 6 illustrates its main steps: given an APK or DEX file, an initial preprocessing step is performed. Here, the bytecode of the app is transformed into an object-oriented representation. At its core, DeStroid consists of two stages: a static analysis stage called *DeStroid Heuristic* and a

Class	Sub-Class	Occurrances	Aggregated Values
String Repository	Static Resources	6	50
	<clinit>	21	
	Static Method	1	
	Static Class	1	
Pass String	Encrypted String	49	
	Static Method	7	
Native Library	Completely Native	1	
	Native to Bytecode	1	
No String Encryption		46	46

Table 1: String Encryption distribution in malware families and its derivation for each sample taken from Malpedia (as of October 2019).

dynamic analysis stage called *Dynamic Deobfuscator*. Provided with the object-oriented representation, the *DeStroid Heuristic* is responsible for detecting the encrypted strings and the deobfuscation part contained in the app. Moreover, it also infers the type of the string encryption technique used. This information is then passed to the *Dynamic Deobfuscator*. This stage implements the actual deobfuscation of the strings by dynamically executing the deobfuscation parts identified in the previous step. Finally, the APK or DEX file is patched with the deobfuscated strings. At the same time, the decrypted strings as well as the identified deobfuscation routines are written to separate result files.

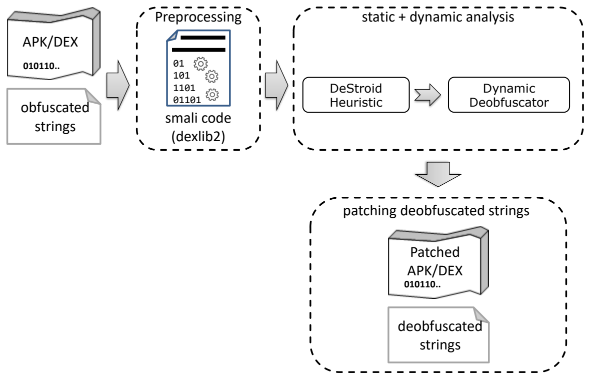


Figure 6: DeStroid architecture.

The following sections describe the steps DeStroid performs in more detail.

4.1 Preprocessing

As already mentioned, the preprocessing step is responsible for transforming the bytecode of an app into an object-oriented representation ((1) in Fig. 7). In our implementation, we use *dexlib2* [23] for this task, but in principle, other frameworks like *Soot* [24] could also be used here. The only requirement we have on the framework being used is that the output must be smali code. Of course, a higher quality output of the preprocessing tool is beneficial for our further analysis steps. The contrary is also true: malware with means to obstruct analyses with the preprocessing framework will also evade the further analysis of our approach.

4.2 DeStroid Heuristic

The DeStroid Heuristic serves two main purposes: first, it identifies the encrypted strings. Second, it is responsible for finding the subset of the app that is required to perform the decryption of these strings.

Fig. 7 illustrates the DeStroid Heuristic stage in more detail. The left part ((2)) depicts the identification process using two heuristics. Here, also the string encryption class is inferred, which, in turn, determines what we call the deobfuscation type. This type defines which runtime deobfuscation techniques will be applied in the following Dynamic Deobfuscator stage and corresponds to one of the classes of the taxonomy defined in Section 2.

Based on the results of the heuristics, we generate an APK to be used in the Dynamic Deobfuscator stage ((3)). DeStroid comes with a *deobfuscation template engine*. This is used to generate a minimal app which gets extended with the encrypted strings and the deobfuscation part of the original app. Besides these parts of the original app, the generated APK contains just the code required to execute the deobfuscation part on the obfuscated strings.

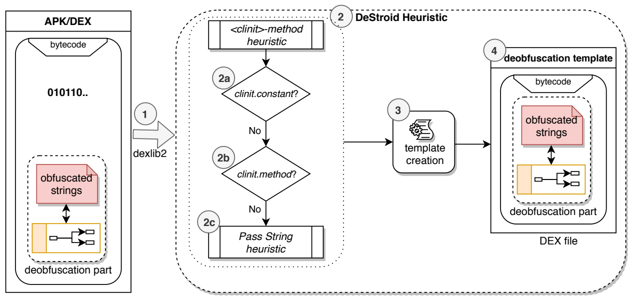


Figure 7: Steps of the *DeStroid Heuristic* stage.

Currently, our prototypical implementation of De-

Stroid supports the three most common obfuscation classes found during our manual analysis: `<clinit>` (SR), Encrypted String (PS), and Static Method (PS). The other classes are currently not considered yet, but can be easily implemented later on.

In the following paragraphs, we describe how we handle the currently supported obfuscations types.

The most commonly used technique is the `<clinit>` class. In addition to the static class initialization, also static fields are initialized within the `<clinit>`-method.

The basic idea of our approach to detect and deobfuscate this class is to exploit that the `<clinit>`-method will either contain the encrypted strings and build a string table with the encrypted strings or the decryption of the encrypted strings is directly applied inside the `<clinit>`-method. We call the former manifestation *clinit.method* and the latter *clinit.constant*.

If *clinit.method* has been detected, the decryption routine has to be executed on values of fields of type `java.lang.String` or on array fields of type int, char, or byte in order to obtain the decrypted values. The expected decryption routine is of type static because the fields which have been written have also to be static.

If *clinit.constant* has been detected, the deobfuscation takes place implicitly through the static initialization block. Therefore, the deobfuscated value is expected to be written to certain fields. Those fields are called *deobfuscation fields*.

When the string table with the encrypted strings is build, there will be write access to all of the static fields which are used for the string table at some point. Consequently, there will be a method which is invoked for each field of the string table. This method is the decryption routine. When the decryption is already applied inside the `<clinit>`-method, then all of the static fields which get written inside the method will contain the decrypted values.

More precisely, our approach works as follows. First, the APK is read and filtered for classes containing a `<clinit>`-method. Next, the `<clinit>`-methods are analyzed to check whether they fulfill at least one of the following criteria:

- *Array instructions*: Either an array has to be written or a new array of type *byte*, *int*, or *char* has to be created inside the `<clinit>`-method. Furthermore, this new array should have at least a size greater or equal to twenty. We choose twenty as manual tests indicated that this value works best.
- *String instructions*: strings or a buffer of strings is created or initialized inside the `<clinit>`-method.

Classes fulfilling the above mentioned criteria are the starting point for the further analysis steps where we compute a static backward slice of the remaining `<clinit>`-method.

A program slice is an executable program that is obtained from the original program by removing statements that are not required in order to replicate the behavior of the original program with respect to the a slicing criterion—the value of interest. This slicing criterion is defined by a program point and a set of program variables.

When a backward slice is computed it consists of all statements and predicates that may affect the value of those variables. On the other hand, a forward slice consists of all statements and predicates in a program that may be affected by the value of those variables [25]. Static indicates, that the slicing is done only on the statically available information of the sliced program.

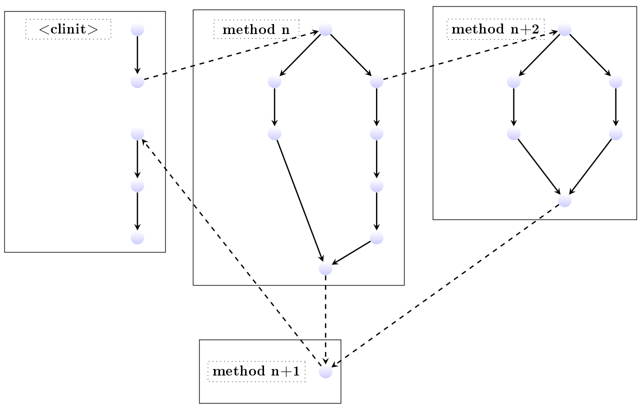


Figure 8: Static backward slicing outlined for the `<clinit>`-method.

The slicing criterion in our example is the `<clinit>`-method and therefore the original program is reduced to all statements which a necessary to execute this `<clinit>`-method.

Thus, all method invocations and class instantiations inside the `<clinit>`-method have to be resolved. Fig. 8 illustrates this static backward slicing.

The subset of the original APK contains either the decryption routine or the instructions to decrypt the encrypted values inside the `<clinit>`-method. We first focus on all fields of the `<clinit>`-method where write access is performed by methods other than the `<clinit>`-method. If the field is of type string (i.e. contains the bytecode sequence *Ljava;lang/String;*) it is added to a list of possible deobfuscation fields. A deobfuscation field is a field which contains the decrypted string value when its appropriate `<clinit>`-method gets executed. In case the field is not of type string but an array of type char, integer, or byte, a counter for each of the fields is incremented. If this counter is greater or equal to the number of elements in the list of possible deobfuscation fields, the decryption is set to the deobfuscation type *clinit.method*. Otherwise, the deobfuscation type is set to *clinit.constant*. In this case, it will stop here and a subset of the original APK is built for the runtime decryption. Otherwise, it will proceed in detecting the decryption routine.

In order to identify the decryption routine for the deobfuscation type *clinit.method* a static forward slice is computed on each field (cf. 10) until it is passed to a method. This implies following the register which referenced the appropriate field. Usually, this results in the same method for each field, but there is the possibility that not all fields have been used as a container for the encrypted strings. Therefore, the method where the static forward slice results the most is specified as the decryption routine. When the `<clinit>` heuristic successfully detects the decryption routine it will stop here (and starts building the *deobfuscation template*) otherwise the second heuristic will be executed.

The second approach—*pass string heuristic*—is using an extended version of the heuristics of Type E1 and E2 by Kühnel et al. [13] to identify the used decryption routines. The E1 heuristic is used to identify encryptions which are based on common encryption API calls in Android. On the contrary, the E2 heuristic is used to identify custom encryption routines by looking for certain patterns of bit and byte operations.

Unlike stated in the paper by Kühnel et al. (cf. 3.1), at first we filter all implemented methods for their parameters and return value (cf. ① in Fig. 9). The parameters and the return value have to contain at least the bytecode type *Ljava/lang/String;* or an array of type char, int or byte. When these requirements are fulfilled the heuristics E1 and E2 are applied in a little different way (illustrated as ②). Otherwise, the heuristic will abort and report that no string encryption could be found.

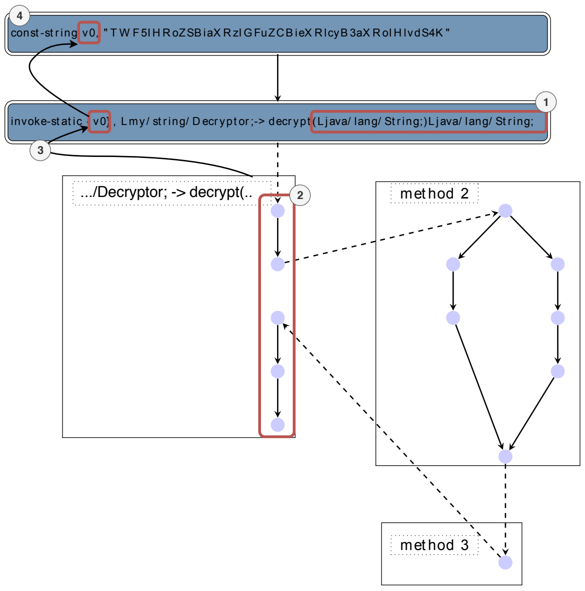

4
const-string v0, "TWF5IHRoZSBiaXRzIGFuZCBieXRlcyB3aXRoIHlvdS4K"
1
invoke-static {v0}, Lmy/string/Decryptor;-> decrypt(Ljava/lang/String;)Ljava/lang/String;
3
.../Decryptor; -> decrypt(..
2
method 2
method 3

Figure 9: Static backward slicing of a decryption routine and their input

4.2.1 E1

Our implementation of E1 tries to detect the standard Java encryption API or common third party cryptographic libraries as well. Those libraries are *java/security*, *javax/crypto*, *Lorg/jasypt*, *Lgnu/crypto*, *org/bouncycastle* and *com/chilkatsoft*. Next—which is an extension to the stated E1 heuristic of Kühnel et al.—we consider only those methods as a string decryption routine if we find an instruction which indicates a decryption invocation. This can be easily identified for each library regarding their given parameters. When we encounter the appropriate instructions which indicates a decryption with the used cryptographic library, the function is added to the list of possible decryption routines.

4.2.2 E2

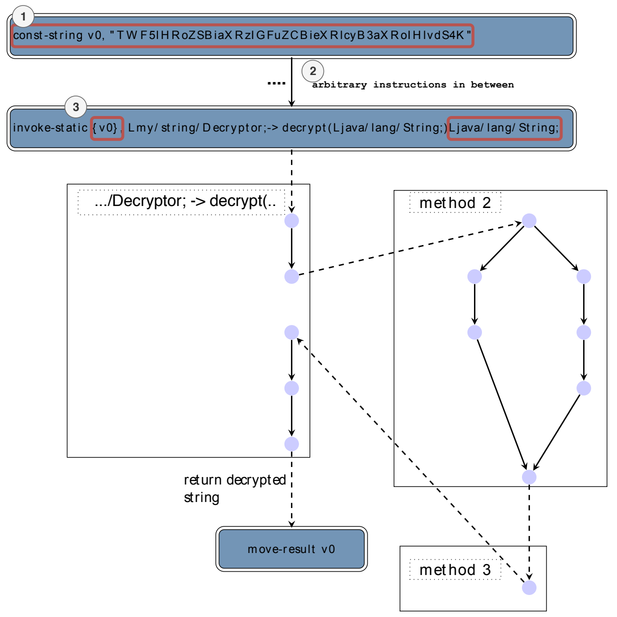
The adapted E2 version tries to identify custom decryption methods which are used for decrypting strings. Those decryption functions follow a certain sequence of operations [13, p. 4]. At first, an array of data is passed to the function. This array data can be passed in three different ways. Data is either converted into a byte array directly, a new empty byte array is created and afterwards filled with data, or an array is returned as a parameter from a method. Eventually, this array data is read and multiple bit and byte operation are applied, ending with an element assignment. Those bit or byte operations consist of bitwise operations, shifts and bit or byte math operations. The bit and byte operations including their assignment should repeat in blocks (cf. [13]). For each block a counter is incremented. When this counter is five or higher (cf. [13]) this functions gets added to the list of possible decryption routines.

After E1 and E2 are finished, the deobfuscation type is set to *Pass String* and the list of possible decryption routines is sorted by the number of invocations. The top five functions with at least four invocations are now considered as decryption functions and they are added to a list of decryption routines. Due to our manual analysis of the different Android malware samples we found that each decryption routine was invoked at least four times. Subsequently, a backward slice based on each decryption routine is computed on register level to receive the encrypted input values (③-④ in Fig. 9). If the input value cannot be computed because, for instance, the input value has to be computed during runtime or received from network, this decryption routine is removed from the list of decryption routines. Otherwise, the encrypted values which has to be either a string or an array of type char, int or byte are put into a map regarding their decryption routine. Hence, the second approach stops here and starts building the *deobfuscation template*.

In some cases it might happen that both approaches are not able to detect neither the encrypted strings nor their decryption routine. For those cases we offer the possibility to set the decryption routine manually or to set an encrypted string for the *DeStroid Heuristic*. If an encrypted string (either with its associated instruction or not) is specified, the APK searches for the bytecode containing this value (see ① in Fig. 10). When the appropriate bytecode is identified, a forward slice on the register containing the encrypted value is performed until its value is passed to string like data structure (see (3)).

The method which has been used to return a string through that value is then classified as the decryption routine. Fig. 10 visualizes the static forward slicing.

After the decryption routine has been identified, a backward slice is computed on register level to receive the encrypted input values (cf. section 4.2.2 and the *deobfuscation template* is generated.



```
const-string v0, "TWF5IHRoZSBiaXRzIGFuZCBieXRlcyB3aXRolHlvdS4K"
invoke-static {v0}, Lmy/string/Decryptor;-> decrypt(Ljava/lang/String;)Ljava/lang/String;
move-result v0
```

1
2
....
arbitrary instructions in between
3
.../Decryptor; -> decrypt(..
method 2
return decrypted string
method 3

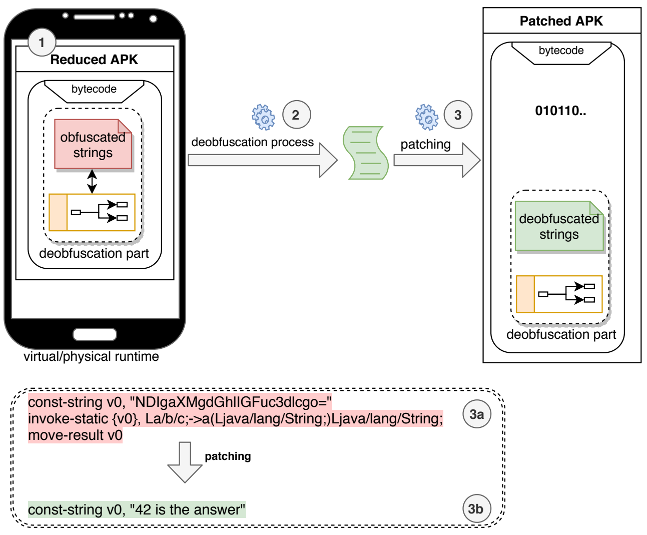
Figure 10: static forward slicing on an encrypted string

Independent from the heuristic a subset of the original app is created for the runtime decryption—the *deobfuscation template*. This subset contains only those instructions which are needed to execute the deobfuscation process. Those instructions are identified through the computed static backward slice in the beginning. Furthermore, an additional class to this subset is included which state the identified deobfuscation type (cf. 4.2). In addition to this, a map for the deobfuscation process is created depending on the identified deobfuscation type. The map for deobfuscation types *clinit.method* and *Pass String* contains the decryption routine and their appropriate encrypted passed values, whereas the deobfuscation type *clinit.constant* is containing a map of the `<clinit>`-method and its deobfuscation fields (see 4.2).

4.3 Dynamic Deobfuscator

The result of the *DeStroid Heuristic* is a subset of the original DEX bytecode with some additional information (reduced APK) to trigger the decryption routines of the analyzed APK. Depending on the string encryption type, this additional information is either a map of encrypted values and their decryption routine or a map with classes and their deobfuscation fields which should hold the decrypted values (cf. section 4.2). This reduced APK is then pushed on an Android device or an emulator (see (1) at Fig. 11).

Next the deobfuscation process is triggered (cf. (2)). According to its deobfuscation type an instance of the respective decryption routine or of the `<clinit>`-method is created. The decryption routine is directly executed with its mapped input and the `<clinit>`-method is already executed when its class gets instantiated. Therefore, the deobfuscation fields of the class are now invoked.



```
const-string v0, "NDIgaXMgdGhlIGFuc3dlcgo="
invoke-static {v0}, La/b/c;->a(Ljava/lang/String;)Ljava/lang/String;
move-result v0
```

```
const-string v0, "42 is the answer"
```

Figure 11: Runtime deobfuscating with Dynamic Deobfuscator and its patching process.

The values of the decrypted strings are reported to *logcat*, Android's logging facility. A separate process monitors the log and keeps a record of the dynamic information reported. When the deobfuscation is finished, this record is used by the patching routine. Beside the decrypted values, the record also contains the exact bytecode location in order to apply the patching.

Finally the patching of the encrypted strings is applied (cf. (3)). In order to do this, the patching routine replace the encrypted strings and their decryption invocation ((3a)) given by its bytecode location with their decrypted counterpart ((3b)).

5 Evaluation

As already mentioned, our dataset consists of 50 different Android malware families in varying versions. This yields a total of 79 different APKs with string encryption.

Due to our manual analysis of each sample we are aware of each decryption implementation—either it is a decryption routine or there is a decryption part directly applied to the encrypted strings. In order to specify the total number of decrypted strings for each sample, we count the number of invocations of the decryption routine. Normally, this reveals the number of decrypted strings. Unfortunately, in some cases the

Class (# deobfuscations)	Sub-Class	deobfuscator	JMD	Dex-Oracle	simplify	DeStroid
String Repository (2434)	Static Ressources (29)	0	0	0	0	0
	<clinit>(1678)	0	0	967	33	1453
	Static Method (421)	0	0	0	0	0
	Static Class (306)	0	0	0	0	302
Pass String (6658)	Encrypted String (6534)	0	0	2117	37	4529
	Static Method (124)	0	0	0	0	106
NL (117)	Completely Native (51)	0	0	0	0	0
	Native to Bytecode (66)	0	0	0	0	0
Total **9209**		**0**	**0**	**3084**	**70**	**6390**

Table 2: Overview of the evaluation results.

decryption is directly applied to the encrypted strings and, therefore, there was no decryption routine at all whose invocations could be count. In those cases, all of the decrypted results have been passed to an array data structure. Then, we use the number of elements in those arrays as the total number of decrypted strings. Table 3 list the total number of decrypted strings for each evaluated sample.

To assess the effectiveness and efficiency of our approach, we compared it against all free available string deobfuscation approaches (cf. 3.2). Those approaches are Dex-Oracle [14] (version 1.0.6), simplify [18, 19] (version 1.2.1), JMD [15] (version 1.6) and Deobfuscator [16] (version 1.0.0).

Simplify is a comprehensive project which offers a lot of possibilities for deobfuscation due to its different optimizers. It works best, when we point simplify directly to the part of the APK where we want to remove obfuscations (cf. [26]). Thus, we run simplify in two configurations: the default option and only deobfuscating the class initializer of each class. For JMD as well as for Deobfuscator we use their generic deobfuscation approach which is generic string decryption for JMD and constant folding for Deobfuscator utilizing its *PeepholeOptimizer*. The runtime deobfuscation of DeStroid and Dex-Oracle are executed on a Google Pixel 2 XL phone with Android 9.0 for applying their decryption logic. Apart from the aforementioned options, we used the default configurations of the deobfuscation approaches.

The assessment of the deobfuscated samples was done by comparing the strings (DEX bytecode *const-string* or *const-string/jumbo*) of the original sample and its corresponding deobfuscated sample. The number of new strings inside the deobfuscated sample is counted and compared to the expected total number of decrypted strings (cf. beginning of section 5). The reason for this, is that all approaches have in common that they reveal a patched respectively deobfuscated APK. The patching — this means inserting new strings to the APK—are the results of their deobfuscation approach.

Table 2 lists the results of the different deobfuscation approaches. The first column indicates the different string encryption types and their total number of deobfuscations. This column highlight the effectiveness of the respective deobfuscation approach in terms of the existing string obfuscation. All further columns list the different approaches regarding their total deobfuscations for each string encryption type. The last row summarize the overall numbers of deobfuscations of the respective deobfuscation approach. A detailed listing of the evaluation results can be found in Appendix on Table 3.

A false positive in the sense of this evaluation would applying a false decryption routine to the encrypted strings which didn't occur during our evaluation and thus not present in the table. This is due to the fact, that all deobfuscation approaches rely on the code of the given sample and only implicitly on the identified encrypted strings. Most approaches search for simple string encryption patterns and then watch for their transformation during execution. Simplify, on the other hand, uses constant propagation on all methods and our approach either applies a decryption routine to its given input or treats the content of possible deobfuscation fields as decrypted strings. Thus, none of the approaches are able to use a false decryption routine to encrypted strings or use a decryption routine with false strings.

Our prototype of DeStroid was able to successfully deobfuscate at least some encrypted strings in 78 % of all obfuscated samples. To be more precise, 44 of the 79 samples could be deobfuscated completely and 10 other samples could be deobfuscated to a degree of $\geq 50\%$ and $< 100\%$. Eight more samples could at least deobfuscated to $< 50\%$. Only 17 samples could not be deobfuscated.

We investigated the overall 17 samples which have not been deobfuscated and found that these samples could not be deobfuscated because the detection of the decryption routine was unsuccessful. In case of the *retefe* malware family [27] the deobfuscation failed because the encrypted resources are stored in the resource file which is currently not considered as a form of string table implementation. An extension of *clinit.constant* to consider resource files as string tables could solve this. In the case of the *marcher* malware family [28] the deobfuscation failed because the string encryption used by this malware family employs a string replacement removing the noise in the initialized strings. This kind of string replacement is currently not considered as a decryption routine. Further analysis of the partial deobfuscated samples reveals that multiple string encryptions in one sample occurred. Therefore only one string encryption type has been deobfuscated because our current prototype stops further analyzing when it successfully identified an obfuscation type.

The other evaluated deobfuscation approaches performed quite different. JMD and Deobfuscator performed as expected due to their focus on commercials obfuscators and limited generic deobfuscation approaches which could not identify any obfuscated strings in the dataset.

Simplify, on the other hand, ran into timing issues or errors while trying to run the samples on its virtual execution environment *smalivm*. Those erros mostly caused by handling abstract methods or when instantiating an abstract class or an interface.

Dex-Oracle worked well for some string encryption implementations. This is because of its string encryption heuristic and the fact that it is only able to run static methods. As a consequence, non-static decryption routine could not be executed. Furthermore, its string encryption heuristic which is based on regular expression can only identify very specific implementations of string encryption.

The results of our evaluation clearly show that our approach works best for *String Repository* and *Pass String* based obfuscations.

6 Limitations and Future Work

Although our approach proved to be able to improve the current state in automated decryption of encrypted strings in Android malware, there are still some limitations which will be discussed in this section.

In cases where multiple string encryptions from different types occurred in one sample, our approach was only able to deobfuscate the strings of one of the implemented encryption types. This happens because the current prototype stops its analysis when an obfuscation type is detected. This is, of course, just an implementation issue of our prototype and not a flaw of the general approach.

Furthermore, DeStroid currently focus only on the bytecode of an analyzed sample. Native code is currently not analyzed. Here, it would be necessary to look for native decryption routines or deposited strings inside the native code. This is a complex research topic on its own and beyond the topic of this paper.

For the detection of decryption routines, currently simple string obfuscations, such as noise on a string, are not considered. These are planned for future versions of our prototype.

Further, nested apps which are packed inside a sample are not considered. The topic of automatic unpacking of Android application is a research field on its own and beyond the scope of this paper.

Finally, DeStroid is highly dependent on the quality of the framework used in the preprocessing step (currently *dexlib2*), which makes it vulnerable to all attacks against this framework. Thus, samples which are protected against analyses with *dexlib2* can currently not be analyzed by DeStroid. Although these cases were not encountered during our evaluation, they should be considered in future developments.

7 Conclusion

In this paper, we presented a survey of string encryption techniques used in Android malware and its different implementations. We proposed a taxonomy of these approaches and identified three major methods used by malware authors: *String Repository*, *Native Library*, and *Pass String*. Moreover, we proposed DeStroid, a hybrid approach for the extraction of the decrypted strings by applying the decryption routine on the encrypted strings. We have shown that this approach works best for *String Repository* based obfuscation as well as for *Pass String* based obfuscation (see table 2). Besides that, it has the big advantage, that based on a given decryption routine, it is able to decrypt all corresponding strings.

Using the decrypted strings produced by DeStroid, it is possible for existing security analysis tools to achieve a more complete analysis and detection of Android malware.

Acknowledgments: We thank our colleagues Jan-Niclas Hilgert and Daniel Plohmann for many useful comments.

Author details

Daniel Baier

Fraunhofer FKIE
Zanderstr. 5, 53177 Bonn
daniel.baier@fkie.fraunhofer.de

Martin Lambertz

Fraunhofer FKIE
Zanderstr. 5, 53177 Bonn
martin.lambertz@fkie.fraunhofer.de

References

[1] D. Plohmann, M. Clauß, S. Enders, and E. Padilla, “Malpedia: a collaborative effort to inventorize the malware landscape,” *Proceedings of the Botconf*, 2017.

[2] E. Protalinski, "Android passes 2.5 billion monthly active devices | VentureBeat," 2019. [Online; https://venturebeat.com/2019/05/07/android-passes-2-5-billion-monthly-active-devices/; accessed 01-July-2019].

[3] I. Statista, "Number of available applications in the Google Play Store from December 2009 to March 2019." [Online; https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/; accessed 26-June-2019].

[4] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *International Conference on Security and Privacy in Communication Systems*, pp. 172–192, Springer, 2018.

[5] D. Baier, "DeStroid - Fighting String Encryption in Android Malware." [Online; https://github.com/fkie-cad/DeStroid].

[6] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 252–276, Springer, 2017.

[7] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers & Security*, vol. 51, pp. 16–31, 2015.

[8] Google Inc., "JNI tips." [Online; https://developer.android.com/training/articles/perf-jni; accessed 28-May-2019].

[9] Oracle America, Inc., "Chapter 5. Loading, Linking, and Initializing." [Online; https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html; accessed 27-May-2019].

[10] Oracle America, Inc., "Static initializers." [Online; https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.7; accessed 27-May-2019].

[11] D. Plohmann, "apk.flexispy." [Online; https://malpedia.caad.fkie.fraunhofer.de/details/apk.flexispy; accessed 27-June-2019].

[12] O. Mirzaei, J. de Fuentes, J. Tapiador, and L. Gonzalez-Manzano, "Androdet: An adaptive android obfuscation detector," *Future Generation Computer Systems*, vol. 90, pp. 240–261, 2019.

[13] M. Kühnel, M. Smieschek, and U. Meyer, "Fast identification of obfuscation and mobile advertising in mobile malware," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, pp. 214–221, IEEE, 2015.

[14] C. Fenton, "Oracle." [Online; https://github.com/CalebFenton/dex-oracle; accessed 26-June-2019].

[15] E. Schoffstall, "Java bytecode analysis/deobfuscation tool." [Online; https://github.com/contra/JMD; accessed 26-June-2019].

[16] Java Deobfuscator, "Java deobfuscator." [Online; https://javadeobfuscator.com/; accessed 26-June-2019].

[17] Y. M. Yoni Moses, "Android app deobfuscation using static-dynamic cooperation." [Online; https://www.virusbulletin.com/uploads/pdf/magazine/2018/VB2018-Moses-Mordekhay.pdf; accessed 26-May-2019].

[18] C. Fenton, "Simplify - Generic Android Deobfuscator." [Online; https://github.com/CalebFenton/simplify; accessed 26-June-2019].

[19] C. Fenton, "TetCon 2016 - Android Deobfuscation: Tools and Techniques." [Online; https://calebfenton.github.io/2016/04/23/tetcon-2016-android-deobfuscation/; accessed 21-June-2019].

[20] J. H. Y. Haehyun Cho and G.-J. Ahn, "DexMonitor: Dynamically Analyzing and Monitoring Obfuscated Android Applications," *IEEE Access*, vol. 6, pp. 71229–71240, 2018.

[21] M. Y. Wong and D. Lie, "Tackling runtime-based obfuscation in android with {TIRO}," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 1247–1262, 2018.

[22] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime data in android applications for identifying malware and enhancing code analysis," tech. rep., Technical Report TUD-CS-2015-0031, EC SPRIDE, 2015.

[23] B. Gruver, "About smali." [Online; https://github.com/JesusFreke/smali/tree/master/dexlib2; accessed 24-June-2019].

[24] Sable Research Group, "Soot - A framework for analyzing and transforming Java and Android applications." [Online; https://sable.github.io/soot/; accessed 24-June-2019].

[25] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, pp. 439–449, IEEE Press, 1981.

[26] C. Fenton, "Simplify - Generic Android Deobfuscator." [Online; https://github.com/CalebFenton/simplify/blob/master/README.md; accessed 26-June-2019].

[27] D. Plohmann, "apk.retefe." [Online; https://malpedia.caad.fkie.fraunhofer.de/details/apk.retefe; accessed 26-June-2019].

[28] D. Plohmann, "apk.marcher." [Online; https://malpedia.caad.fkie.fraunhofer.de/details/apk.marcher; accessed 26-May-2019].

APPENDIX

In the following we list the details of our findings regarding the classification of the ground truth as well as the evaluation details.

7.1 Evaluation details

Table 3 details the results of the evaluation (cf. section 5).

7.2 Ground Truth Details

Table 4 details our finding for building and classifying the ground truth (cf. Section 2).

APK (# deobfuscations)	deobfuscator	JMD	Dex-Oracle	simplify	DeStroid
adultswine.apk (26)	– 0	– 0	– 0	– 0	● 26
anubisspy_sample1.apk (18)	– 0	– 0	– 0	– 0	– 0
anubisspy_sample2.apk (18)	– 0	– 0	– 0	– 0	● 18
asacub.apk (11)	– 0	– 0	● 11	– 0	● 11
bahamut_sample1.apk (6)	– 0	– 0	– 0	– 0	● 6
bahamut_sample2.apk (6)	– 0	– 0	– 0	– 0	● 6
bahamut_sample3.apk (6)	– 0	– 0	– 0	– 0	● 6
bankbot_sample1.apk (426)	– 0	– 0	● 426	– 0	● 426
bankbot_sample2.apk (454)	– 0	– 0	– 0	– 0	● 454
bankbot_sample3.apk (98)	– 0	– 0	– 0	– 0	● 98
bianlian.apk (421)	– 0	– 0	– 0	– 0	– 0
brata_sample1.apk (12)	– 0	– 0	– 0	– 0	◐ 10
brata_sample2.apk (12)	– 0	– 0	– 0	– 0	◐ 10
catelites_2017-12-15.apk (71)	– 0	– 0	– 0	– 0	◐ 65
catelites_2017-12-17.apk (71)	– 0	– 0	– 0	– 0	● 71
catelites_2017-12-21.apk (70)	– 0	– 0	– 0	– 0	● 70
catelites_2018_01_19.apk (66)	– 0	– 0	– 0	– 0	◐ 60
cerberus.apk (73)	– 0	– 0	– 0	– 0	● 73
comet_bot.apk (498)	– 0	– 0	◐ 341	◔ 37	● 498
charger.apk (44)	– 0	– 0	– 0	– 0	● 44
chrysaor_sample1.apk (5)	– 0	– 0	– 0	– 0	– 0
chrysaor_sample2.apk (5)	– 0	– 0	◐ 3	– 0	● 5
doublelocker.apk (586)	– 0	– 0	– 0	– 0	◐ 287
dualtoy.apk (5)	– 0	– 0	– 0	– 0	● 5
dvmap.apk (16)	– 0	– 0	– 0	– 0	● 16
exobot_sample1.apk (381)	– 0	– 0	◐ 329	– 0	◔ 4
exobot_sample2.apk (374)	– 0	– 0	◐ 323	– 0	◔ 4
exobot_sample3.apk (374)	– 0	– 0	◐ 323	– 0	◔ 4
exodus_sample1.apk (5)	– 0	– 0	◔ 1	– 0	● 5
exodus_sample2.apk (5)	– 0	– 0	◔ 1	– 0	● 5
exodus_sample3.apk (6)	– 0	– 0	◔ 1	– 0	● 6
flexispy.apk (87)	– 0	– 0	– 0	– 0	◔ 21
flexnet.apk (58)	– 0	– 0	– 0	– 0	● 58
glancelove.apk (118)	– 0	– 0	– 0	– 0	● 118
goldenrat.apk (20)	– 0	– 0	– 0	– 0	– 0
gustuff_sample1.apk (72)	– 0	– 0	– 0	– 0	● 72
gustuff_sample2.apk (78)	– 0	– 0	– 0	– 0	● 78
hiddenad.apk (135)	– 0	– 0	– 0	– 0	● 135
hydra.apk (51)	– 0	– 0	– 0	– 0	– 0
joker.apk (12)	– 0	– 0	– 0	– 0	◔ 4
kevdroid_sample1.apk (26)	– 0	– 0	– 0	– 0	● 26
kevdroid_sample2.apk (26)	– 0	– 0	– 0	– 0	● 26
lokibot.apk (525)	– 0	– 0	◐ 221	– 0	● 525
marcher_2016-10-19.apk (145)	– 0	– 0	● 145	– 0	– 0
marcher_2016-12-01.apk (154)	– 0	– 0	● 154	– 0	– 0
marcher_2017-01-29.apk (221)	– 0	– 0	● 221	– 0	– 0
marcher_2017-07-26.apk (367)	– 0	– 0	● 367	– 0	◔ 4
mazarbot_2017-01-01.apk (44)	– 0	– 0	– 0	◐ 33	● 44
mazarbot_2017-08-20.apk (58)	– 0	– 0	– 0	– 0	● 58
mazarbot_2017-10-11.apk (58)	– 0	– 0	– 0	– 0	● 58

APK (# deobfuscations)	deobfuscator	JMD	Dex-Oracle	simplify	DeStroid
mazarbot_2017-11-07.apk (65)	— 0	— 0	— 0	— 0	● 65
monokle.apk (48)	— 0	— 0	— 0	— 0	◐ 30
mysterybot.apk (304)	— 0	— 0	◐ 151	— 0	● 304
podec.apk (418)	— 0	— 0	— 0	— 0	◯ 26
pornhub.apk (626)	— 0	— 0	— 0	— 0	◐ 391
premier_rat.apk (1)	— 0	— 0	— 0	— 0	— 0
raxir.apk (1001)	— 0	— 0	— 0	— 0	● 1001
retefe_2014-06-23.apk (2)	— 0	— 0	— 0	— 0	— 0
retefe_2014-09-12.apk (2)	— 0	— 0	— 0	— 0	— 0
retefe_2014-11-10.apk (2)	— 0	— 0	— 0	— 0	— 0
retefe_2015-01-29.apk (2)	— 0	— 0	— 0	— 0	— 0
retefe_2015-05-13.apk (2)	— 0	— 0	— 0	— 0	— 0
skygofree_2016-11-24.apk (302)	— 0	— 0	— 0	— 0	● 302
slempo.apk (46)	— 0	— 0	— 0	— 0	● 46
slocker.apk (11)	— 0	— 0	— 0	— 0	● 11
smsspy.apk (5)	— 0	— 0	— 0	— 0	◯ 2
spybanker.apk (39)	— 0	— 0	— 0	— 0	● 39
svpeng.apk (8)	— 0	— 0	— 0	— 0	● 8
telerat.apk (6)	— 0	— 0	— 0	— 0	— 0
tempting_cedar.apk (428)	— 0	— 0	— 0	— 0	— 0
tinyz_sample1.apk (66)	— 0	— 0	● 66	— 0	● 66
triada.apk (93)	— 0	— 0	— 0	— 0	● 93
triout.apk (34)	— 0	— 0	— 0	— 0	◐ 18
viper_rat_dropper.apk (3)	— 0	— 0	— 0	— 0	● 3
yellyouth.apk (19)	— 0	— 0	— 0	— 0	— 0
zoopark_v4.apk (15)	— 0	— 0	— 0	— 0	◐ 10
ztorg.apk (10)	— 0	— 0	— 0	— 0	◐ 6
ztorg_downloader.apk (368)	— 0	— 0	— 0	— 0	● 368
ztorg_payload.apk (81)	— 0	— 0	— 0	— 0	● 81
$\sum \#deobfuscations$: **9209**	**0**	**0**	**3084**	**70**	**6390**

● = 100% deobfuscation ; ◐ = deobfuscation $\geq 50\%$ and $< 100\%$; ◯ = deobfuscation $< 50\%$; — = no deobfuscation; 🟦 = developed prototype; the hashes of each sample can obtained from the provided taxonomy (cf. [5])

Table 3: Evaluation results in detail.

APK	Classification							
	String Repository				NL		Pass String	
	<clinit>	ressources	static class	static method	completely native	native to bytecode	encrypted string	static method
adultswine.apk								✓
anubisspy_sample1.apk								✓
anubisspy_sample2.apk								✓
asacub.apk	✓							
bahamut_sample1.apk							✓	
bahamut_sample2.apk							✓	
bahamut_sample3.apk							✓	
bankbot_sample1.apk							✓(2)	
bankbot_sample2.apk							✓(2)	
bankbot_sample3.apk							✓(4)	
bianlian.apk				✓				
brata_sample1.apk							✓	
brata_sample2.apk							✓	
catelites_2017-12-15.apk	✓							
catelites_2017-12-17.apk	✓							
catelites_2017-12-21.apk	✓							
catelites_2018_01_19.apk	✓							
cerberus.apk							✓	
charger.apk							✓	
comet_bot.apk							✓(3)	
chrysaor_sample1.apk	✓							
chrysaor_sample2.apk	✓							
doublelocker.apk	✓							
dualtoy.apk							✓	
dvmap.apk	✓							
exobot_sample1.apk							✓	
exobot_sample2.apk							✓	
exobot_sample3.apk							✓	
exodus_sample1.apk							✓	
exodus_sample2.apk							✓	
exodus_sample3.apk							✓	
flexispy.apk	✓					✓	✓(2)	
flexnet.apk	✓							
glancelove.apk	✓							
goldenrat.apk							✓(2)	
gustuff_sample1.apk							✓	
gustuff_sample2.apk							✓	
hiddenad.apk	✓							
hydra.apk					✓			
joker.apk							✓(3)	
kevdroid_sample1.apk							✓	
kevdroid_sample2.apk							✓	
lokibot.apk							✓(10)	
marcher_2016-10-19.apk	✓							
marcher_2016-12-01.apk	✓							
marcher_2017-01-29.apk	✓							
marcher_2017-07-26.apk	✓							
mazarbot_2017-01-01.apk							✓	
mazarbot_2017-08-20.apk							✓	
mazarbot_2017-10-11.apk							✓	

APK	Classification							
	String Repository				NL		Pass String	
	<clinit>	ressources	static class	static method	completely native	native to bytecode	encrypted string	static method
mazarbot_2017-11-07.apk							✓	
monokle.apk	✓						✓(3)	
mysterybot.apk							✓(5)	
podec.apk							✓(2)	
pornhub.apk	✓						✓(6)	✓
premier_rat.apk							✓	
raxir.apk							✓(2)	
retefe_2014-06-23.apk		✓						
retefe_2014-09-12.apk		✓						
retefe_2014-11-10.apk		✓						
retefe_2015-01-29.apk		✓						
retefe_2015-05-13.apk		✓						
skygofree_2016-11-24.apk			✓					
slempo.apk							✓	
slocker.apk							✓	
smsspy.apk							✓	
spybanker.apk							✓	✓
svpeng.apk							✓	
telerat.apk							✓	
tempting_cedar.apk							✓	
tinyz_sample1.apk	✓							
triada.apk	✓							
triout.apk							✓	
viper_rat_dropper.apk							✓	
yellyouth.apk		✓						
zoopark_v4.apk							✓(2)	
ztorg.apk							✓	
ztorg_downloader.apk	✓							
ztorg_payload.apk							✓	

✓= used string encryption type; (#) = number of different decryption routines; further information (e.g. the exact location of the deobfuscation routine) about each sample can obtained from the provided taxonomy (cf. [5])

Table 4: Ground truth classification details