Representing String Computations as Graphs for Classifying Malware

Justin Del Vecchio
jmdv@buffalo.edu
University at Buffalo
Buffalo, New York

Steven Y. Ko
stevko@buffalo.edu
University at Buffalo
Buffalo, New York

Lukasz Ziarek
lziarek@buffalo.edu
University at Buffalo
Buffalo, New York

ABSTRACT

Android applications rely heavily on strings for sensitive operations like reflection, access to system resources, URL connections, database access, among others. Thus, insight into application behavior can be gained through not only an analysis of what strings an application creates but also the structure of the computation used to create theses strings, and in what manner are these strings used. In this paper we introduce a static analysis of Android applications to discover strings, how they are created, and their usage. The output of our static analysis contains all of this information in the form of a graph which we call a string computation. We leverage the results to classify individual application behavior with respect to malicious or benign intent. Unlike previous work that has focused only on extraction of string values, our approach leverages the structure of the computation used to generate string values as features to perform classification of Android applications. That is, we use none of the static analysis computed string values, rather using only the graph structures of created strings to do classification of an arbitrary Android application as malware or benign. Our results show that leveraging string computation structures as features can yield precision and recall rates as high as 97% on modern malware. We also provide baseline results against other malware detection tools and techniques to classify the same corpus of applications.

CCS CONCEPTS

• **Theory of computation** → **Program analysis**; • **Human centered computing** → *Mobile computing*; • **Security and privacy** → Malware and its mitigation.

KEYWORDS

Android, String Structure, Static Analysis

ACM Reference Format:
Justin Del Vecchio, Steven Y. Ko, and Lukasz Ziarek. 2020. Representing String Computations as Graphs for Classifying Malware. In *IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387905.3388595

1 INTRODUCTION

Android is one of the most popular platforms for mobile computing and Android's official app store, Google Play, contains more than 2.9 million unique applications (apps) as of December 2018. However, this popularity has also resulted in a dramatic increase in malicious apps targeting Android devices. As of 2019, security experts [1] have identified over 31 million *unique*, malicious mobile apps. To combat this increase in malware the security community has developed many static analysis tools for identifying malicious Android apps. By identifying malware statically, these tools can offer enhanced security for end users who download and use Android apps. Many different types of analyses have proven to be useful such as data and control flow analysis [11, 22, 46], call graph analysis [38], sensitive API analysis [23, 57], extraction of string constants [34, 52], as well as analyzing strings to extract API usage and permissions [53].

String analysis, which focuses on identification of string values within a program, is a common technique used in the detection of malware. For example, being able to identify static string values provides insight into which SQL queries are performed by an application and can be used to identify SQL injection attacks [26, 31, 54].

What all these approaches share is a desire to understand the content of strings; what their values are and what can be understood from these values. Our string analysis is focused on the structure of the strings - how strings are constructed, manipulated, and where in the program they are created as well as used. Our observation is that malicious code on Android uses complex string operations to hide the intent of the computation and evade detection by anti-virus software. In this paper we focus on malicious behavior defined by string operations. We show that analysis of string structures can be useful to distinguish between malicious and benign apps. This occurs because Android apps heavily rely on strings in order to perform potentially sensitive or malicious tasks, such as reading contact information and device IDs, accessing storage, calling hidden APIs, leveraging reflection to hide which code is being called, and using remote servers.

Our string analysis system has three components: (1) a static analysis that extracts strings, computations that use them, and computations that construct them, (2) a feature space generator for extracted string computations, and (3) a k-fold cross validation training and test methodology utilizing multiple machine learning algorithms. Our static analysis examines apps and extracts all string computational structures - storing them as graphs. The feature space generator creates a feature set that records the frequencies of critical structural elements as well as sequences of the structural elements for the computation structures, stores associated graphs in a graph database, and provides analyses over the graphs using graph queries. The classifier component uses gradient boosting with a k-fold cross validation test and evaluation methodology. Our analysis clearly shows that use of gradient boosting with the computational structures treated as features is an effective, generic mechanism to differentiate malware from benign apps.

Contributions. The contributions of this work include:

- An interprocedural static analysis for extracting strings, where they are used, and the structure of the computation that generates them, for Android.
- A generic approach to classify app behavior as malware or benign based on strings and their associated computations. The classification results demonstrate how computational structures can be used as features to label an app as benign or malicious with high precision and recall. Importantly, it is done (1) without developing an expert system dependent on manually generated feature sets, and (2) with no reliance on the string literal values themselves. We devise mechanisms for automatically extracting best-possible features for classification from the results of our static analysis.
- Analysis results from modern malware. Often, malware analysis relies on data sets that collected years in the past, leaving it difficult to say how results compare with what is currently seen in the world today. Results are reported against malware and benign Android apps in use during 2018 and 2019.

2 MOTIVATION

String constants are of interest to developers of malware detection tools. For example, VirusTotal's [48] analyzed malware reports include a section titled ”Interesting Strings” and the online malware detection service HybridAnalysis [6] provides a search capability specifically tailored for strings. Both examples provide only the literal values found in apps statically. Our string computations provide the critical backstory for these literals - how they were created and where they are used. They can serve as a compliment to the string literals provided by both services. Additionally, there are many string values that cannot be analyzed statically but are critically important in order to understand the behavior of malware.

We consider the following two real-world examples discovered in the course of our research to illustrate how uses of strings in computation and the structure of the computation to generate strings can be leveraged to identify malicious behavior. These examples are from a set of malware apps first identified in 2018 by VirusTotal. VirsutTotal URLs for each app are included in the References and provide fine grained detail for each app. In addition, the string literal values can be searched and found on HybridAnalysis.

2.1 Example 1: Useless String Ops

The app `App1SQL` [49] was first identified in June 2018 with 32 of the VirusTotal's 63 detection engines flagging it as malware. These engines list it as belonging to various malware families including Trojan Downloader, Riskware, Spyware and PUA. We reviewed the string computations generated by our analysis of `App1SQL` and identified multiple computations that were suspect. We provide decompiled code for one such computation in Figure 1 via an abbreviated extract of the app's actual code.

```
1  public final class a extends SQLiteOpenHelper {
2   public a(Context context, String str,
        CursorFactory cf, int i) {
3    super(context, str, null, 2);
4   }
5  } //end class a
6
7  public class b{
8   public static a a(Context context) {
9     ...
10    synchronized (b.class) {
11     try {
12       a = new a(context, f.a(context), null, 2);
13     } catch (Throwable th) {
14         Class cls = b.class;
15     }
16    }
17   return a;
18  } //end class b
19
20
21 public class f{
22   public static String a(Context con) {
23    return new
          StringBuffer(b(con)).reverse().toString();
24   }
25
26   private static String b(Context context) {
27    String str = bj.b;
28    if (b.b((Object) str)) {
29      return str;
30     }
31     return "zdlVOdEpGZmV5Sm5ZVzFsWDI1a";
32     }
33 } //end class f
```

Figure 1: Useless String Operations.

Line 1 shows that `App1SQL` extends the `SQLiteOpenHelper` class and one of its constructors that accepts a string as its second argument. Line 12 shows where this class, `a`, is initialized for use in the class `b`. The string passed in as the second argument comes from passing the `Context` object to the method `a` of class `f`. Here, method `a` calls method `b`, shown on line 26, which does nothing with the `Context` object itself (a red herring) and either returns a string returned from another method or the hard coded string value, ”zdlVOdEpGZmV5Sm5ZVzFsWDI1a”. This string value is then turned into a `StringBuffer`, its sequence reversed, and it is next turned back into a `String`. All of these operations are performed to simply obtain the name of the SQLite database this application will use. This is a complex set of string operations purposefully enacted to confuse what the name of the installed database is. Our string analysis was able to capture the structure of these string operations for this example and is explained further in Section 3.

2.2 Example 2: Dynamic Code Loading

The app `App2Dex` [50] was first identified in June 2018 with 36 of the VirusTotal's 63 detection engines flagging it as malware. These engines list it as belonging to various malware families including

```
1  public class c{
2   private boolean a(Context context, File file) {
3      ...
4      DexClassLoader dexClassLoader = this.b;
5      this.c = dexClassLoader.loadClass(
6        b.a("2ab7d44a89503834ffc598", bj.b));
7      Class cls = this.c;
8      this.e = cls.getMethod(
9      b.a("aeaa0994519899d5a82e2d", bj.b), new
           Class[0]);
10     ...
11  }
12 }
13
14 public class b{
15  public static String a(String str, String str2) {
16     try {
17         if (str2.trim().length() < 2) {
18             str2 = b;
19         }
20         return new String(a(a(str), str2));
21     } catch (Exception e) {
22         c.a().a(e);
23         return "208";
24     }
25   }
26
27  private static byte[] a(String str) {
28     byte[] bytes = str.getBytes();
29     int length = bytes.length;
30     byte[] bArr = new byte[(length / 2)];
31     for (int i = 0; i < length; i += 2) {
32         bArr[i / 2] =
33           (byte) Integer.parseInt(new String(bytes,
               i, 2), 16);
34     }
35     return bArr;
36  }
37 }
```

Figure 2: DexClassLoader Example.

Trojan Downloader, Riskware, Spyware and SMS payment thief. We reviewed the string computations our analysis generated for App2Dex and quickly identified multiple computations that were suspect. We provide decompiled code for one such computation in Figure 2 via an abbreviated extract of the app's actual code.

Figure 2 shows how the app uses the class DexClassLoader - a class included in the Android API that can be used to execute code not installed as part of an application [7]. The class b has a method named a that at line 5 uses a DexClassLoader object to dynamically load code. The computational path it uses to derive the name of the class it wishes to instantiate is incredibly convoluted to follow manually but easily connected by our string computations. It supplies the string argument to loadClass of DexClassLoader by calling b.a with the value "2ab7d44a89503834ffc598" (abbreviated from its much longer original value). The method b.a uses multiple nested calls at line 20 that eventually lead to a call to method a at line 27. Here, the initial string passed in as the first argument to b.a is transformed into a byte[], which is then decrypted (this method is not shown) and passed as an argument to the String constructor, which itself is then passed back as the string value to instantiate as a class. This is a perfect example of an extremely complex string construction that goes to extreme lengths to hide the name of the class it is dynamically instantiating. Our algorithm generated a string computation that was able to track the string that is the first argument to the method called within loadClass at line 5 to its translation into a byte[] at line 28.

These two examples demonstrate that a manual examination of a string's computation provides valuable insight into the behavior of an app. We will show that the structure of this computation can be leveraged to distinguish malware from benign applications. We focus the rest of the paper on creating classifiers to automate this manual process. These classifiers will use the computational structures as features and avoid entirely the use of string values, or literals, for classification. Our system implementation sought to answer the following set of research questions:

- Research Question 1: How do ML models generated from our string computations classify malware and benign apps when compared with state of the art malware detection tools or against classification using string literals only?
- Research Question 2: How well do combined approaches work when we add our string computations as another feature for state of the art malware detection tools or combine them with the string literals?
- Research Question 3: How does obfuscation impact our classification results, as well as the results for the state of the art or string literals?
- Research Question 4: What features from the string computation feature space are most important in the classification of malware and benign apps?

3 ANALYZING STRINGS

We implement our string analysis as a flexible system capable of quickly processing and analyzing apps. Figure 3 shows the major components of our system and their integration. The following subsections provide detail on each of these components.

3.1 Structural Analysis of Strings

Input to our system is a set of Android apps to be analyzed. Finding all strings within an app statically can be viewed as a specialized form of constant propagation [52, 55]. This is the starting point of our analysis and provides us with a set of statically computable string constants. Obviously, not all strings can be generated statically as they may depend on dynamically computed values, user input, and other I/O operations. We leverage interprocedural control and dataflow analyses augmented with the ability to reason about common string manipulations (e.g. concatenation, substrings, conversions of literals to strings) and some common libraries (e.g. string builder) to identify all computations used to create a string.

We maintain a complete listing of all the String, StringBuilder, and StringBuffer APIs and record when a string operation is performed and how this operation is linked to preceding and trailing string operations. We do not analyze the internals of the string manipulation libraries themselves. Abstractly, this provides a graph

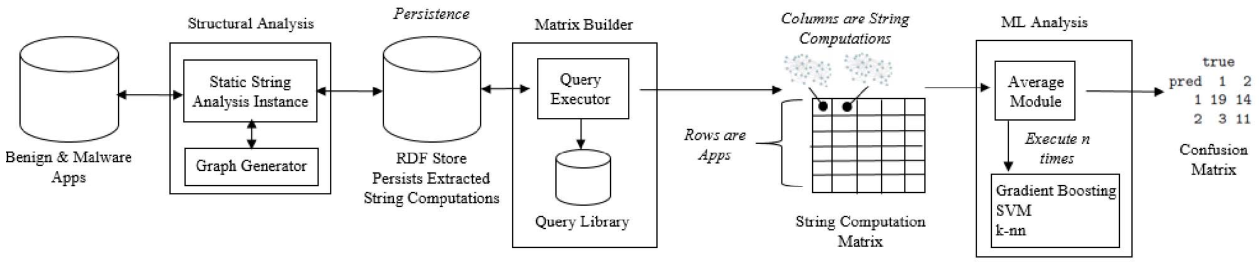

Figure 3: String Analysis System.

that contains control and data dependencies (computation structure) for generating a string, or set of strings in the case of control flow operations like branches. We call the set of strings a string computation graph could generate at runtime a *string family*. Figure 4 provides a pictorial representation of how the intraprocedural analysis works using a code snippet taken from `com.symantec.mobilesecurity`.

The intraprocedural analysis uses as input decompiled code provided by Soot, a Java instrumentation framework [51]. The atomic units of the decompiled code are called Units in Soot's vocabulary. Each method contains a set of Units that embody bytecode operations of the method. We use the backward flow analysis framework provided by Soot to iterate through the method's Units—starting at a unit that uses a string or is an output of our constant propagation and then visiting all units in the method in the reverse order of the method's control flow. While doing so, our string analysis identifies those Units where string objects are used, and performs dependency analysis to identify other Units which the current Unit is control or data dependent upon. Effectively, we compute a backward slice originating from the usage of the string (Line 7 for the string extracted from `builder1` and Line 6 for the string extracted from `builder2`). The backward slice captures all statements necessary to compute the string transitively, so for the use in Line 7 the backward slice would contain Lines 2, 1, and 5. For use in Line 6 the backward slice would contain Lines 4, 3, and 5. We filter the backward slice for each string to only contain string operations (we would filter out Line 5) but maintain their relative positions in the slice, effectively creating a graph. Each graph represents a set of strings that share a common computation structure that the program can compute at runtime.

```
1    StringBuilder builder1 = new StringBuilder();
2    builder1.append("expiry_date");
3    StringBuilder builder2 = new StringBuilder();
4    builder2.append("active_date");
5    ContentValues content = new ContentValues();
6    content.put(builder2.toString(), "val2");
7    content.put(builder1.toString(), "val1");
```

Figure 4: Intraprocedural Constant Propagation.

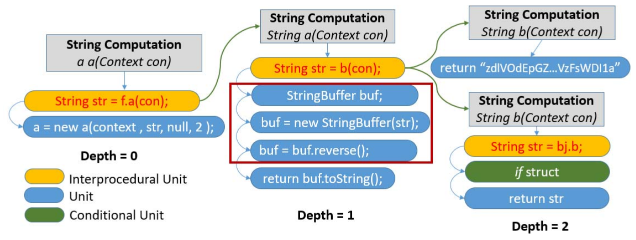

Figure 5: String Computation Graph Structure.

3.2 Representing and Reasoning About Strings as Graphs

Figure 5 pictorially shows the resultant string computations we extract and the graphs we use to represent them. Note, some of the graph content generated by our algorithm is reordered and condensed for read-ability in this figure. This graph is from `App1SQL` described in Section 2 with accompanying code. We read the graph from left to right. At upper left is a gray rectangular box that represents a root node for a string computation, listing the name of the method where the string is created or used. Here, we have detected that a string is created in the method `a a(Context con)` that returns an object of type `a`. The sequence of blue and yellow nodes connected to the StringComputation represents the Soot units used to build this portion of the string within the method. All of the code used by the method to construct the string is captured as a node in our graph structure. Blue nodes are normal Soot units, yellow nodes reflect interprocedural pointers to another StringComputation, and greens are conditionals. We see the first Soot unit added creates a `String` with some content that is interprocedurally obtained. Our graph structure contains an edge that connects the Soot unit with another StringComputation, in this case the string created in `String a(Context con)`. As we trace down this graph we see its first Soot unit is an interprocedurally connected one. We also see that there are additional Soot nodes beneath this StringComputation. We see a series of nodes that take a `String`, turn it into a `StringBuffer`, call `reverse()` on the buffer, and then turn back into a `String`. We last look at the two StringComputation at far right. They both represent content that is returned from `String b(Context con)`. In the upper case it is a hard coded `String` value and in lower case it is a value harvested from a class variable from `bj`. The lower StringComputation represents the fact that an `if` statement occurred on which `str` is control dependent on prior to the `return` – the subgraph rooted at an if node contains operations on the string representing the then clause and the else clause, if present. In this particular example we omit the sub-graph rooted at the if node for readability of the figure.

Each intraprocedural string computation is persisted in the RDF graph database as an individual graph. If an intraprocedural computation is part of an interprocedural graph, it will have a pointer to the other intraprocedural computations whose aggregate forms the interprocedural graph. Our current implementation does lead to multiple string computations with identical structure stored within the graph database. In the future we may merge these identical graphs into a single graph and employ a pointer strategy were we use vertexes that point to the string computation's root node to indicate where in the code base (i.e. method and line number) the merged string computation is used.

Our persistence step, shown in Figure 3 as the storage of string computation graphs, provides us a composite graph database across all apps examined. The graph database is populated with thousands of graphs like the one presented in Figure 5. This allows us to quickly develop graph queries that analyze and extract new feature sets, decoupling string meta-analysis from the static analysis performed within Soot.

We use an advanced graph query engine called SPARQL to pull critical information back from our string analysis outputs. SPARQL is a W3C standard for querying required and optional graph patterns along with their conjunctions and disjunctions [20]. We leverage SPARQL's advanced capabilities to perform graph pattern matching to find patterns of interest. Figure 6 shows a graph query that pulls out portions of the graph presented in Figure 5. This query returns all string computations that have interprocedural points and that are itself a root node (or starting point). SPARQL is somewhat analogous to SQL with a SELECT and WHERE clause. The example provided here condenses some of the variable and relationship names for readability. The SELECT clause pulls out all string computations and then counts the number of interprocedural points that are one hop out from the current string computation graph. Lines 4 and 5 instruct SPARQL to find nodes that are of type StringComputation and that point to a Soot unit node. Line 6 is a critical step. It finds the next Soot unit connected to the first one. Here, using SPARQL's advanced property paths, we can *recursively* search and find all nodes that are connected to the first Soot unit node with the edge name `next`, all enabled by the + syntax. Line 7 finds the root node of the interprocedurally connected sting computation. Line 8 is critical as it excludes any pointers to the root string computation - avoiding reporting on string computations that are interprocedurally connected but are not a root node.

```
1  SELECT ?stringComp (COUNT(?stringComp2))
2  WHERE
3  {
4    ?stringComp <has_part> ?sootUnit .
5    ?stringComp <type_of> StringComp .
6    ?sootUnit <next>+ ?sootUnit2 .
7   ?sootUnit2 <has_interprocedural_part> ?stringComp2
         .
8   FILTER NOT EXISTS(?stringComp ^<has_part>
         ?stringComp3)
9  }
```

Figure 6: SPARQL Example.

3.3 Feature Set Development

Our goal is to run ML algorithms on the extracted string computations and develop a model that will classify an app as malware or benign. We have a single SPARQL query that pulls out each string computation from the graph database. This query is configured to avoid pulling out intraprocedural string computations that are part of larger interprocedural string computations. This reduces the feature space as well as avoids double counting intraprocedural string computations that are a part of larger interprocedural string computation structures. We then apply graph pattern matching to identify the structural string computations that are identical within an app as well as across the entire corpus of apps. Note, we also experimented with feature extraction that collected simpler representations of the string computations by counting the API calls within the computations themselves. This approach was faster to calculate than the string computations but had lower accuracy when compared with string computations using ML trained models.

We then construct a matrix where each header represents a string computation and the matrix rows represent the apps. This is shown in Figure 3. The cells record how many times that row's app exhibits that string computation. We leverage the matrix as the primary datastructure for storing features as well as their static counts per app that will be utilized by our classifier and principle component analysis (PCA).

The size of the string computations themselves in terms of the number of vertexes can range from one vertex, in the case of some string literals, to hundreds of vertexes, representing complex string families. We sampled 2,000 malware apps from our data set, described later in Section 4, and identified the average app contained 1,821 string computation graphs and the average size of graphs in terms of number of vertexes is 7.7. Figure 7 provides the frequency of computation sizes across the 2,000 examined apps whose vertex counts per string computation are between 20 and 100. We see that smaller size string computations dominate but there are still thousands of string computations whose size is greater than 50 vertexes - demonstrating the complexity of the string computations and their resultant string families.

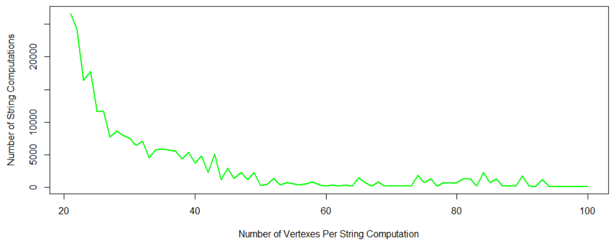

Figure 7: Frequency of String Computation Sizes.

Table 1: Overview of Data Sets Used for Classification

Type	Year	# Apps
benign	2019	16,281
	2018	14,257
malware	2019	7,717
	2018	6,256

3.4 Classification

The last step is to perform classification where apps are labeled as either malware or benign. We use three classification algorithms with multiple configurations to determine which works best. These are gradient boosting, SVM, and k-nearest neighbors (K-NN), discussed in more detail in Section 5.

4 DATA

In this section we introduce the data sets used in the evaluation of our string computation analysis. Results for these data sets are provided in Section 5. Table 1 shows the number of apps analyzed by year for the malware and benign categories. We choose a set of apps that reflected recent malware, between January 2018 and December 2019. We obtained apps from two locations. First is Androzoo [4], a collection of Android Applications collected from several sources, including the official Google Play app market. This site now contains over 10 million Android apps freely available to researchers for download. We also used VirusTotal [48], a virus detection service developed by Google and that contains millions of apps - both malware and benign.

Benign Apps We downloaded apps from Androzoo that met two criteria. First, the dex size of the application needed to be at least 1MB in size. It is important to note that the dex size of an app is only a small portion of the total app size as it does not include metadata, media, and other files packaged inside an app. We wanted to include apps with more complexity and avoid smaller, trivial apps that were devoid of many features and tend to result in over specialized classification models. This was done to create a balanced representation for our baseline analysis where we compare classification using string computations against other techniques. Second, the app had to originate from Google Play as it employs the best security examination of apps posted to its site. For example, Google play removed 700,000 apps in 2017 it self-identified as malware [40]. Selecting apps only from Google plays gives us the best chance to have a mostly benign data set as there is always the possibility of unidentified malware still present within. As an additional mechanism to establish ground truth, we ran our benign set through VirusTotal and removed any apps that were flagged as suspicious.

Malware Apps We downloaded apps from both Androzoo and VirusTotal that met two criteria. First, the dex size of the application needed to be at least 1MB in size. We did this to avoid smaller malware apps that simply lacked many features to classify upon or are trivially identifiable. Second, we limited the apps to only malware that was initially identified on the Google Play platform itself. This allowed us to test our approach on malware purposefully developed to be similar to offerings on Google Play in an attempt to trick users into downloading the apps.

Obfuscated Apps We next take a subset of apps for both malware and benign and run them through three obfuscation tools. We choose as as our obfuscation tools: (1) the Automatic Android Malware Obfuscator (AAMO) [43] which has been used to demonstrate how virus detection tools perform against obfuscation, (2) DroidChameleon[44] also used to demonstrate the performance of virus detection tools in the presence of obfuscation, and (3) Obfuscapk[19] which is a recent, open source obfuscator with multiple advanced obfuscation configurations. We configured the obfuscators to use obfuscation techniques that Hammad et al. [32] identified as challenging for virus detection methods.

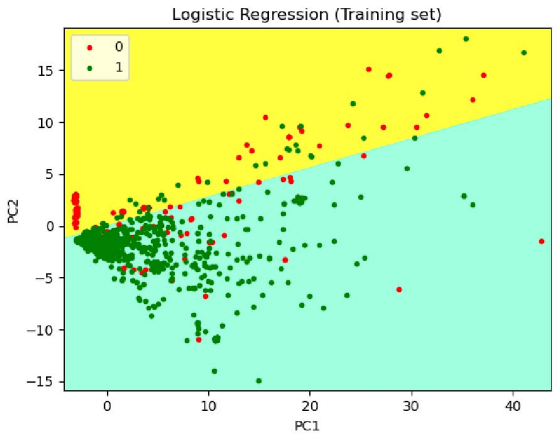

Figure 8: PCA First Two Components.

4.1 Characterization of the Datasets

We next applied PCA to the data sets using the feature set described in Section 3. Our feature matrix composed of string computations is high-dimensional data, where a sample of 4000 apps, 2000 benign and 2000 malware, would identify over 5000 features. We used PCA to reduce the attribute space into a smaller number of factors and then visualize the relationship between the features as well as understand the main variance in the data. We applied the PCA function into the train and test sets for analysis and then fit logistic regression to the train set. We then predicted the training set result, visualized through a scatter plot shown in Figure 8. It identifies that: (1) benign apps and malware apps tend to be located in different areas of the component space, and (2) benign apps and malware apps tend to cluster in these different areas, and (3) there is some overlap between the cluster outliers. PCA shows us that there is high likelihood that the sting computations will lend themselves well as feature for classification. This data is a good visualization of the intuition behind why our approach works at distinguishing malware from benign apps. We present concrete results in the following section.

5 RESULTS

We address each of our research questions in this section. Overall, our results show that string computations are an effective mechanism for classification of malware and benign apps. Importantly, the computations themselves only include structure information about how the strings are created - not the literal string values which proves detrimental when apps are obfuscated using encryption techniques. We begin with a discussion of our statistical analysis approach followed by an examination of each of the research questions in turn.

5.1 Setup

We perform evaluations on our string computations using gradient boosting[25], SVM and K-NN. Note, we report results only on gradient boosting (GB). All described experiments were run using all three classifiers. We found that SVM and K-NN had lower F-measure universally than GB and therefore omit their results as they provide no additional insight over what GB provides. GB itself uses an ensemble of regression trees (or decision trees) and is able to learn higher order interactions between features. This ability to learn the interaction between features is likely the reason why it outperforms both SVM and K-NN. This characteristic is also evident in Kaggle, Google's online community for data scientists and machine learners. GB dominates winners circles for Kaggle competitions that involve structured datasets [5].

We optimize the hyperparameters GB uses as follows. Trees are grown starting with a tree depth of zero and grow greedily [16]. We run GB with tree depths of 10, 15, 20, and 25 using 100 and 200 rounds of training. We report results in terms of the best combination of tree depth and rounds. We utilize 10-fold cross validation, repeated 4 times, and report on the average of these four runs.

We assess accuracy of the classification using the standard F-measure metric described as:

$$F = 2 * \frac{precision * recall}{precision + recall}$$

Here, precision represents TP / (TP+FP) and recall represent TP / (TP+FN). TP is the number of malicious samples correctly classified, FP is the number of benign samples identified as malicious, and FN is the number of malware samples identified as benign. TN is the number of benign samples correctly classified.

5.2 RQ1: Comparison to State of the Art

Our first research question establishes a baseline against which to judge malware classification using string computations as features. We first compare against RevealDroid which generates a small, simple set of features that are selectable by users [27]. Its specified goal is to provide a feature set that is family agnostic as well as obfuscation resilient. RevealDroid is used in numerous studies as a baseline for malware classification, especially in the presence of obfuscation [37, 47]. We downloaded and installed RevealDroid, ensuring that it output the maximum set of features. Its typical F-measure ranges between 93% to 97% - this value dependent on the corpus of apps under evaluation and may range higher of lower in specific use cases [27].

We next compare against the compliment of our string computations; the string literals themselves. Our string computations capture the structure of code execution used to construct strings while literals are just the static values of the string themselves that are computable during a static analysis - such as a message that is written to a log, content of a SQL query, etc. Researchers [34, 52] have utilized string literals to do classification with very good results - reaching F-measures of 97%. Similarly most modern anitvirus software leverages string literals. Our string literals are those string values defined within the app dex file.

We randomly choose as input 8,000 apps from our dataset, 4,000 malware and 4,000 benign. We report on how our string computations, RevealDroid, and string literals perform in terms of precision, recall, and F-measure in Table 2.

Table 2: Baseline Comparison

Tool	Precision	Recall	F-measure
String Computations	97.30	97.33	97.31
RevealDroid	91.42	96.96	94.11
String Literals	94.30	96.12	95.20

RQ1 Answer: The results demonstrate that string computations by themselves perform competitively against classification using RevealDroid or string literals. Our approach is roughly 3% better than RevealDroid and 2% better than string literals. This shows string computations can be an effective feature upon which to classify malware and benign apps. The result also demonstrates that the behavior of strings - how strings are created and used - is different between malware and benign applications.

5.3 RQ2: Comparison of Combined Approaches

We next examine how the approaches work when combined. That is, when string computations are added as another feature for both the RevealDroid ML analysis as well the string literals analysis. Figure 9 shows how to generate a new ML model by combining feature sets. To the left is the feature set for string computations and to the right is the feature set for RevealDroid. As both report results in terms of a matrix where each row is an app and each column a feature, we simply join on the app IDs to create a single matrix. This approach is especially applicable to RevealDroid as it readily accepts the incorporation of new features. For example, an analysis was performed by its creators where results from FlowDroid [10] were included as another feature upon which to perform classification [28]. The combined analysis for string literals now includes the literal value along with the string computation that generated the literal value.

As before, we randomly choose as input 8,000 apps, 4,000 malware and 4,000 benign, from the pool of apps. We report on our combined approach for three different combinations. The first combines all the features RevealDroid outputs with the string computations. The second combines a select set of features from RevealDroid not derived from the source code with the string computations. To clarify, this case uses features RevealDroid extracts from the manifest file, the included native libraries, etc. with the string computations. The third combines the string literals with the string computations.

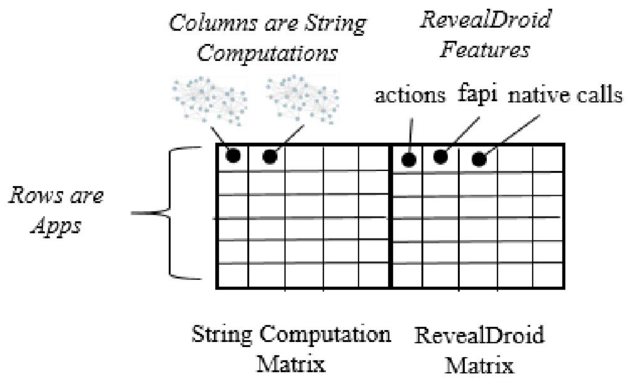

Figure 9: Combined String Computation and Reveal Droid Matrix.

We report results for all three types of combinations in terms of precision, recall, and F-measure in Table 3.

Table 3: Combined Approach

Combination	Precision	Recall	F-meas
All Reveal and Computations	96.93	96.82	96.87
Select Reveal and Computations	97.62	97.63	97.62
Literals and Computations	95.68	97.08	96.37

RQ2 Answer: Combing all of RevealDroid's features with the string computations is better at classification than RevealDroid alone. This is expected as RevealDroid focuses on extraction of a generic set of attributes for high fidelity malware classification across multiple families. String computations are great example of a generic attribute as, regardless of the family of malware, all must use strings to access sensitive operations. However, this combination was not better than using string computations alone. The next combination uses a specific subset of RevealDroid features, everything but the counts of Android API calls, and does show an improvement over string computations alone. In fact, this combination is our best F-measure for malware classification. This is likely due to the fact that the string computations are a more precise feature than counts of Android API calls as the computations embed semantics of how and why strings are used and created. Our last combination shows that string literals combined with string computations classifies better than string literals alone. This is likely due to the fact that apps create many string computations internally where the literal value is only known at runtime. Combining the literal values with the string computations should increase overall F-measure and it does.

5.4 RQ3: Obfuscations Impact on Results

This experiment answers the question of how our approach, as well as RevealDroid and string literals, performs with respect to obfuscated apps. We utilize three tools to perform obfuscation: (1) DroidChameleon [44], an obfuscation tool with various trivial and non-trivial techniques to obfuscate malware applications, (2) AAMO [43] which also allows for an equivalent set of trivial and non-trivial techniques for obfuscation of malware, and (3) Obfuscapk [8], a modular python based obfuscation tool with advanced obfuscators specifically intended to defeat Android malware detection techniques. All tools are freely available and easily scriptable. DroidChameleoon and AAMO were used previously to obfuscate apps and test the resilience of malware detection tools using different combinations of trivial and non-trivial obfuscations [13, 32]. Obfuscapk was released in late 2019 on Github and has a growing community of users.

We base our experiment off of the study conducted by Hammad et al. [32], which provides statistics on obfuscation settings that are most effective at reducing the accuracy of malware detection across a host of industry standard, malware detection tools. Based on the recommendations and results presented in [32], we apply the following obfuscations separately using all three obfuscation tools with the last obfuscator only available on Obfuscapk:

- Control Flow Manipulation (CFM) - Changes the method's control by adding conditions and iterative constructs as well as introducing new methods.
- Member Reordering (MR) - Changes the order of instance variables or methods in a classes.dex file to defeat malware detection tools that look for sequences of members in a class.
- Reflection (REF) - Perform transformations that convert direct method invocations into reflective calls using the Java reflection API.
- String Encryption (ENC)- This encrypts string literal values found in dex files. Obfuscapk will also encrypt supporting app files such as assets or included, native libraries.
- Advanced Reflection (AREF)- This obfuscator uses reflection to invoke dangerous APIs of the Android Framework. It is a novel reflection approach used only by Obfuscapk that focuses on hiding Android API calls which many malware detection tools use to cluster malware. It is a technique used by no other open source obfuscation tools and demonstrates how obfuscators themselves continually evolve.

Note that many of the apps we attempted to obfuscate with the tools simply failed to obfuscate. This is due to the fact that obfuscation tools, especially freely available ones, are notorious for having failures when applying different types of obfuscation [28]. On average, roughly 30% of the apps we used for the evaluations of RQ1 and RQ2 could not successfully be obfuscated by either DroidChameleon or AAMO for each of the obfuscation types applied. Obfuscapk had a much higher success rate at obfuscation versus DroidChameleon and AAMO, failing on less than 5%.

We first obfuscated the apps using the CFM, MR, REF, ENC, and AREF obfuscators separately for each obfuscation tool - AREF only applicable to Obfuscapk. We then ran each obfuscated app through our string computation algorithm, RevealDroid, as well as extracted out the string literals using the linux `strings` command. We then choose as input to the GB classifier 4,000 random apps, 2,000 malware and 2,000 benign, and report on results in Table 4 in terms of F-measure only for brevity.

RQ3 Answer: We see that some obfuscators do have an effect on the outputs of the different analysis tools. First is the ENC obfuscator which has a tremendous, detrimental impact on string

Table 4: GB for Obfuscated Apps

Tool	CFM	MR	REF	ENC	AREF
DroidChameleon					
String Computation	96.58	96.12	95.38	96.33	
RevealDroid	94.11	94.45	93.63	94.08	
Literals	94.88	95.09	95.22	0	
AAMO					
String Computation	96.28	96.85	96.10	96.72	
RevealDroid	94.30	94.55	93.85	94.17	
Literals	95.11	94.02	95.20	0	
Obfuscapk					
String Computation	96.0	96.66	96.98	96.37	96.75
RevealDroid	94.8	94.84	94.62	90.3	90.62
Literals	94.62	94.1	94.80	0	94.22

literal analysis. When we attempt to run the GB classifier on the ENC string literals for any obfuscation tool, it fails to complete as it simply can find no features that are common across two or more apps to even begin the analysis. This is because a unique key is used for each app to encrypt its string literals and therefore there is no commonality in features across the apps. It demonstrates how simple string literal encryption can defeat malware classifiers that focus on the occurrence of tokens (or signatures) to classify apps. The ENC obfuscator has no impact on our string computations as we disregard string literals completely or on RevealDroid for AAMO and DroidChameleon as it does not use string literals extracted from dex files as a feature. However, the literal encryption for Obfuscapk does impact RevealDroid. This is because Obfuscapk also encrypts string literals in files outside the dex file. As RevealDroid relies on some of these strings as features, its performance degrades when they are encrypted uniquely across each app.

Another obfuscator that impacts results is Obfuscapk's AREF. Its ability to obfuscate specific Android API calls - those made to sensitive or dangerous libraries - impacts classification using RevealDroid features. F-measure for RevealDroid drops to 90% when AREF is used. This is because RevealDroid focuses on API calls to specific Android libraries and AREF purposefully obfuscates portions of this feature set. It reduces the total number of identifiable API calls and thus alters the dependency of the feature to app type as demonstrated by the drop in F-measure. Obfuscapks developers designed this obfuscator for this specific case. However, its application is skillful as Obfuscapk's developers only obfuscate a small set of targeted API calls as reflecting on the entire Android API would makes apps unacceptably slow. Also note that AREF has little impact on string computations. As the API calls are themselves turned into strings they become another feature for our string computations to classify upon.

Note, we acknowledge that it would be possible to design an obfuscation technique to fool our analysis. Indeed, Obfuscapk developers recognized that some malware detection tools focus on a select set of Android API calls as a signature and developed a specific obfuscator to impact it. A similar analysis of our string computations might lead an obfuscator designed to develop a specific obfuscator to impact it as well.

```
        PackageManager pm = Context.getPackageManager()
        String packageName = pm.getPackageName()
        PackageInfo pi =
              pm.getPackageInfo(packageName, ...)
```

Figure 10: PackageManager Example.

Our second critical take away is that some obfuscators have little to no impact on analysis results. This is the case with respect to the traditional obfuscation techniques used by all three - CFM, MR, and REF. Table 4 highlights that there is little difference between the GB analysis using the original and obfuscated apps across the board for string computations and RevealDroid. Why is this the case? Consider the definitions of the obfuscation settings applied and how they might interfere with our analysis of string computations.

CFM impacts the app's method control flow as well as introduces new methods. The obfuscators do introduce new methods but only as proxies to existing methods and with minimal method control manipulation based on our manual inspection of the obfuscated code. That is, we opened apps in their original and obfuscated forms and compared where the control flows were introduced to gauge how they would impact generation of our string computations.

MR works with the dex code and moves around chunks to reorder sequences of members in a class. This has little effect on our string computations as we reconstruct how strings are created - not the layout of the dex code itself. It also has minimal impact on RevealDroid as the features it harvests are not sequences of code. For string literals this has absolutely no effect as this obfuscation does not split the literal values.

The REF used by AAMO applies to only statically defined methods. Our computations contain mostly Java string API calls which are not statically defined. Therefore, the original structure of the string computations is again mostly preserved. The latest version of DroidChameleon provided to us by its authors has weak REF capabilities that simply adds a redirect for each method defined within the apps code base. Traditional REF has little effect on RevealDroid as many of the features it extracts are in the Android API namespace and AAMO does not reflect on these. REF had no effect on string literals as it does not split the strings.

5.5 RQ4: Analysis of Variable Importance to Results

This question answers which variables are most important to the GB model for differentiation of malware from benign apps. GB provides a mechanism that explains which features impacted the created models most, ranking them from most to least important. In our case it shows which string computations are most influential to our the model. We use python's gboost variable importance function to return the most important features for GB models for our string computations run on the unobfuscated apps. Figure 10 provides a snippet of the string computation that is most important in classification of malware and benign apps.

RQ 4 Answer: The importance factors indeed shed light on why strings are important to malware. Consider Figure 10 that shows how a string computation is used to interact with `PackageManager`. A very common behavior for malware when first installed is to perform a listing of installed packages and then send the list to the command and control server [2]. An example of malware that uses the `PackageManager` string computation to perform a listing of installed packages is the Anubis banking malware that reappeared in July of 2019 [41]. Also note that there is no associated string literal for the string computation. Other examples of features of importance include the use of strings to communicate between Android Activities using the Android `putExtra()` API call as well as the construction of highly expressive strings using `StringBuilder` that are then thrown as messages by Java errors.

6 RELATED WORK

Application of string analysis for mobile apps has a rich research background as there is a clear value in enhanced string creation understanding for determining app intent or for use in verification techniques. The first holistic examination of string analysis in Java applications was performed by [17] with Java string analysis (JSA). This work focuses on the values that may occur as a result of a string expression, for example statically determining the value of string that represents a SQL insert statement. Violist [36] [35] extended the work of JSA and is an excellent example of an algorithm developed to represent string creation ops for both intraprocedural and interprocedurally created strings. It focuses on the introduction of configuration options that allow users to unroll loop structures how they desire to find values of strings at specific points of program execution.

Our work differs in that we focus on the structure of the graph used to generate the string value as opposed to the string value itself and the consumer of the developed string (i.e. URL connection, reflection call, etc.). Our analysis algorithm develops a graph representation for each string created by an app and persists this graph representation for later analytics. The graph representations can be stored in a singular graph based data store where the outputs of multiple apps are meant to co-located. Analytics routines may easily be developed that pull back the graph representations for all strings supplied to URL connections or calls to Reflection to understand critical differences in the complexities of string creation in benign versus malware apps. This capability can identify similar or like approaches to string creation but also can identify new or novel approaches to the creation of strings that can potentially identify new malware that are different from known malware. This capability was identified as a critical need in malware analysis by [12].

Research exists [33] on extensions to DBDroidScanner that focus heavily on the analysis of string operations for strings used in attacks on databases in benign Android apps. The approach performs a rigorous analysis of how string operations related to URI and SQL calls are constructed. It discusses the structure of string calls Symbolic Finite Transducers to symbolically execute the strings creation. Our work focuses on holistic analysis of the graph patterns themselves used to construct strings and the differences that occur in these patterns between malware and benign apps. In addition [34] provides an extensive examination of a string literals where the literals themselves as treated as grams of one, two, and three - similar to our analysis but solely focused on literals.

Machine learning techniques are also very popular among researchers for detecting malicious Android apps. However, most of these solutions train the classifier only on malware samples and can therefore be very effective to detect other samples of the same family. For example, DREBIN [9] extracts features from a malicious app's manifest and disassembled code to train their classifier, where as MAST [15] leverages permissions and Android constructs as features to train their classifier. Xu [56] provides an embedding approach to represent and compare call graphs across platforms to identify the same, cross compiled malware using a combination of graph matching and machine learning. Shen *et al.* [45] analyze the structure and behavioral features along with Complex-Flows to classify benign and malware apps. We believe these coarse features are a great mechanisms to filter many apps prior to leveraging techniques like our own, which require more analysis of the app internals. There are many other systems, such as Crowdroid [14], and DroidAPIMiner [3], that leverage machine learning techniques to analyze statistical features for detecting malware. Similarly, researchers developed static and dynamic analyses techniques to detect known malware features. Apposcopy [24] creates app signatures by leveraging control-flow and data-flow analysis. RiskRanker [29] performs several risk analyses to rank Android apps as high-, medium-, or low-risk. Sebastian *et al.* [42] analyze dynamic code loading in Android apps to detect malicious behavior. [21], [18] and [30] are all signature-based malware detection techniques and are designed to detect similar malware apps. [30] is of particular interest as it aims to represent know malware signatures as strings instead of hashes. Moonsamy *et al.* [39] provided a thorough investigation and classification of 123 apps using static and dynamic techniques over the apps' Java source code. To the best of our knowledge we are the first to consider leveraging the structure of computation, and specifically the structure of computation of strings, as a feature for classification. We believe our approach is orthogonal to others proposed and our feature set can be combined with other proposed features to further improve classification.

7 CONCLUSION

In this paper we present a string analysis technique that provides a set of computational structures for how a string is constructed and where it is used inside an app. These string graphs can provide us with significant insight into an Android app's behavior. We use traditional ML classification techniques to show that string graphs are excellent features for classification of an app as either malicious or benign. For future work we will focus on applying the techniques discussed in this paper to identify emerging malware families by analyzing string graphs and the evolution of string families as string graphs change as malware families evolve. An examination of string graphs that are present in only benign apps, but suspicious in structure, could identify indicators for new, previously unseen types of malware. In addition we will perform a more exhaustive review of how obfuscation tools alter call graphs.

ACKNOWLEDGMENTS

This work is supported in part by National Science Foundation grants: CRI:1823230 and SHF:1749539.

REFERENCES

[1] [n.d.]. Mobile Threat Report 2019 - McAfee. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf.

[2] 2019. Mobile Malware Analysis : Tricks used in Anubis. https://eybisi.run/Mobile-Malware-Analysis-Tricks-used-in-Anubis/. (2019).

[3] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *Security and Privacy in Communication Networks - 9th International ICST Conference, SecureComm 2013, Sydney, NSW, Australia, September 25-28, 2013, Revised Selected Papers*. 86–103.

[4] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo. (2016), 468–471. https://doi.org/10.1145/2901739.2903508

[5] Yassine Alouini. 2019. Why is XGBoost among most used machine learning method on Kaggle? . https://www.quora.com/Why-is-XGBoost-among-most-used-machine-learning-method-on-Kaggle. (2019).

[6] Hybrid Analysis. 2018. https://www.hybrid-analysis.com.

[7] Android. 2019. https://developer.android.com/reference/dalvik/system/DexClassLoader.

[8] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. 2020. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* 11 (2020), 100403. https://doi.org/10.1016/j.softx.2020.100403

[9] Daniel Arp, Michael Spreitzenbarth, Hugo Gascon, and Konrad Rieck. 2014. Drebin: Effective and explainable detection of android malware in your pocket.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick Mcdaniel. 2014. FlowDroid : Precise Context , Flow , Field , Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *PLDI '14 Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), 259–269. https://doi.org/10.1145/2594291.2594299

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Yves Klein, Jacques a nd Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA.

[12] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining apps for abnormal usage of sensitive data. In *Proceedings - International Conference on Software Engineering*. https://doi.org/10.1109/ICSE.2015.61

[13] Shikha Badhani and Sunil K. Muttoo. 2019. Analyzing Android Code Graphs against Code Obfuscation and App Hiding Techniques. *Journal of Applied Security Research* 14, 4 (2019), 489–510. https://doi.org/10.1080/19361610.2019.1667165 arXiv:https://doi.org/10.1080/19361610.2019.1667165

[14] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices* (Chicago, Illinois, USA) *(SPSM '11)*. ACM, New York, NY, USA.

[15] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. 2013. MAST: Triage for Market-scale Mobile Malware Analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks* (Budapest, Hungary) *(WiSec '13)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2462096.2462100

[16] Tianqi Chen. [n.d.].

[17] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise Analysis of String Expressions. int id (2003), 1–18. https://doi.org/10.1007/3-540-44898-5_1

[18] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. 2005. Semantics-Aware Malware Detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (SP '05)*. IEEE Computer Society, Washington, DC, USA, 32–46. https://doi.org/10.1109/SP.2005.20

[19] Georgiu Claudiu. 2019. Obfuscapk . https://github.com/ClaudiuGeorgiu/Obfuscapk. (2019).

[20] World Wide Web Consortium. 2019. https://www.w3.org/TR/rdf-sparql-query/.

[21] William Enck, Machigar Ongtang, and Patrick McDaniel. 2009. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/1653662.1653691

[22] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Frañziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. 2014. Collaborative Verification of Information Flow for a High-Assurance App Store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. ACM, New York, NY, USA.

[23] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11)*. ACM, New York, NY, USA.

[24] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 576–587. https://doi.org/10.1145/2635868.2635869

[25] Jerome Friedman. 2002. Stochastic Gradient Boosting. *Computational Statistics Data Analysis* 38 (02 2002), 367–378. https://doi.org/10.1016/S0167-9473(01)00065-2

[26] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. 2007. A static analysis framework for detecting SQL injection vulnerabilities. *Proceedings - International Computer Software and Applications Conference* 1, Compsac (2007), 87–94. https://doi.org/10.1109/COMPSAC.2007.43

[27] Joshua Garcia, Mahmoud Hammad, and Sam Malek. 2018. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. *ACM Transactions on Software Engineering and Methodology* 26, 3 (2018), 1–29. https://doi.org/10.1145/3162625

[28] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. [n.d.]. *Obfuscation-Resilient, Efficient, and Accurate Detection and Family Identification of Android Malware*. Technical Report. http://cs.gmu.edu/703-993-1530

[29] 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* (Low Wood Bay, Lake District, UK) *(MobiSys '12)*. ACM, New York, NY, USA, 281–294. https://doi.org/10.1145/2307636.2307663

[30] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-Cker Chiueh. 2009. Automatic Generation of String Signatures for Malware Detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection* (Saint-Malo, France) *(RAID '09)*. Springer-Verlag, Berlin, Heidelberg, 101–120. https://doi.org/10.1007/978-3-642-04342-0_6

[31] William G J Halfond and Alessandro Orso. 2005. Amnesia. *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05* 5 (2005), 174. https://doi.org/10.1145/1101908.1101935 arXiv:1203.3324

[32] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products. (2018), 421–431. https://doi.org/10.1145/3180155.3180228

[33] Behnaz Hassanshahi and Roland H.C. Yap. 2017. Android Database Attacks Revisited. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS '17*. https://doi.org/10.1145/3052973.3052994

[34] Richard Killam, Paul Cook, and Natalia Stakhanova. 2014. Android Malware Classification through Analysis of String Literals. (2014).

[35] Ding Li, Yingjun Lyu, Jiaping Gui, and William G. J. Halfond. 2016. Automated energy optimization of HTTP requests for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. https://doi.org/10.1145/2884781.2884867 arXiv:arXiv:1508.06655v1

[36] Ding Li, Yingjun Lyu, Mian Wan, and William G J Halfond. [n.d.]. String Analysis for Java and Android Applications. ([n. d.]).

[37] Zhiqiang Li, Jun Sun, Qiben Yan, Witawas Srisa-an, and Yutaka Tsutano. 2019. Obfusifier: Obfuscation-Resistant Android Malware Detection System. In *Security and Privacy in Communication Networks*, Songqing Chen, Kim-Kwang Raymond Choo, Xinwen Fu, Wenjing Lou, and Aziz Mohaisen (Eds.). Springer International Publishing, Cham, 214–234.

[38] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*.

[39] Veelasha Moonsamy, Moutaz Alazab, and Lynn Batten. 2012. Towards an understanding of the impact of advertising on data leaks. *Int. J. Secur. Netw.* 7, 3 (March 2012).

[40] Steven Nichols. 2017. How Google Fights Android Malware. https://www.zdnet.com/article/how-google-fights-android-malware/. (2017).

[41] Charlie Osborne. 2019. Anubis Android banking malware returns . https://www.zdnet.com/article/anubis-android-banking-malware-returns-with-a-bang/. (2019).

[42] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[43] Mila Dalla Preda and Federico Maggi. 2017. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques* 13, 3 (2017), 209–232. https://doi.org/10.1007/s11416-016-0282-2

[44] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. 2013. DroidChameleon: Evaluating Android Anti-Malware against Transformation Attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (Hangzhou, China) *(ASIA CCS '13)*. Association for Computing Machinery, New York, NY, USA, 329–334. https://doi.org/10.1145/2484313.2484355

[45] Feng Shen, Justin Del Vecchio, Aziz Mohaisen, Steven Y. Ko, and Lukasz Ziarek. [n.d.]. Android Malware Detection using Complex-Flows. In *Proceedings of The 37th IEEE International Conference on Distributed Computing Systems (ICDCS '17)*.

[46] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y. Ko, and Lukasz Ziarek. 2014. Information Flows As a Permission Mechanism. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. ACM, New York, NY, USA.

[47] Guillermo Suarez-Tangil, Santanu Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. 2017. DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware. https://doi.org/10.1145/3029806.3029825

[48] Virus Total. 2018. https://www.virustotal.com.

[49] Virus Total. 2019. https://www.virustotal.com/#/file/9361abaff9fabc7fbf875802d0bc4f8a.

[50] Virus Total. 2019. https://www.virustotal.com/#/file/09f59201d2d0e41169b9bd0c332b6275.

[51] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press.

[52] Justin Del Vecchio, Feng Shen, Kenny M. Yee, Boyu Wang, Steven Y. Ko, and Lukasz Ziarek. 2015. String Analysis of Android Applications (N). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 680–685. https://doi.org/10.1109/ASE.2015.20

[53] Wei Wang, Zhenzhen Gao, Meichen Zhao, Yidong Li, Jiqiang Liu, and Xiangliang Zhang. 2018. DroidEnsemble: Detecting Android Malicious Applications with Ensemble of String and Structural Static Features. *IEEE Access* 6 (2018), 31798–31807. https://doi.org/10.1109/ACCESS.2018.2835654

[54] Gary Wassermann and Zhendong Su. 2007. Sound and precise analysis of web applications for injection vulnerabilities. *ACM SIGPLAN Notices* 42, 6 (2007), 32. https://doi.org/10.1145/1273442.1250739

[55] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (April 1991), 181–210. https://doi.org/10.1145/103135.103136

[56] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17* (2017). https://doi.org/10.1145/3133956.3134018

[57] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. 2013. Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security* (Berlin, Germany) *(CCS '13)*. ACM, New York, NY, USA.