UNDERSTANDING THE EFFECTS OF OBFUSCATION ON STRING ANALYSIS FOR MALWARE CLASSIFICATION

by

Justin Michael Del Vecchio

August 31, 2020

A dissertation submitted to the
Faculty of the Graduate School of
the University at Buffalo, The State University of New York
in partial fulfilment of the requirements for the
degree of

Doctor of Philosophy

Department of Computer Science and Engineering

Acknowledgments

I have received a tremendous amount of support and guidance during the writing of this dissertation.

I would first like to thank my supervisor, Professor Lukasz Ziarek, whose expertise is exceptional and who always took the time to explain how to perform proper, rigorous research. His insights were tremendous and it was his guidance that enabled this dissertation to grow from a set of research ideas, into a research plan, and then into this document. I could not be more appreciative of his help.

I would also like to thank my committee members Professors Steve Ko and Bharat Jayaraman who took the time to understand the research plan and provide critical feedback. They provided perspective on how others would view the research and helped make for a much more well-rounded dissertation.

I would like to thank my mother and father who could not be more proud of this accomplishment.

I would like to thank my beautiful wife Keri, who was supportive on the long nights as I worked, seemingly endless, to get research performed and papers written. She was a tremendous help and her encouragement helped make this dissertation a reality. Lastly, I would like to thank (and hug!) my three wonderful boys, Jack, Mark, and Calvin who are perfect in every way and the best children anyone could have.

Table of Contents

List of Tables

List of Figures

Abstract

A high quality malware detection tool must utilize a feature set that provides high classification accuracy but is also difficult for malware developers to mask with obfuscation. We propose that the use of string structures provides this type of feature as it can classify malicious code with high accuracy even in the presence of obfuscation. Obfuscation itself provides malware developers tools and techniques to hide malicious intent. This significantly degrades the ability of many malware detection tools to accurately classify malware as occurrences of malicious intent used as classification features are no longer present or purposefully misrepresented. String structures capture how strings are created and employed by mobile applications and are themselves resilient to different forms of obfuscation. This is because string structures are not focused on the literal content of the string and are focused instead on the set of operations needed to create the string which is more difficult to meaningfully obfuscate. We provide an analysis of multiple obfuscation tools, both academic and open source, and examine how different types of applied obfuscation change an application's code structure. Our research shows that these changes do not significantly affect string structures nor the most important features from this set used for malware classification. Our research furthers the field of malware analysis in two ways. First, it provides string computations as a new, obfuscation resilient feature for classification of malware and benign applications. This feature type can be used either independently or as a complement to other feature types for more accurate classification of malware. Second, it provides a set of metrics that may be used to measure obfuscation resiliency in

terms of the feature set used for classification. Currently, obfuscation resiliency is tied to accuracy measures for obfuscated applications but ignores how the underlying feature set changes and its implications. Our metrics allow this change to be measured and compared across different obfuscation tools and their obfuscators.

Chapter 1

Introduction

1.1 Introduction

Android is one of the most popular platforms for mobile computing and Android's official app store, Google Play. However, this popularity has also resulted in a dramatic increase in malicious apps targeting Android devices. As of 2019, security experts [78] 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 [44, 90, 15], call graph analysis [73], sensitive API analysis [111, 45], extraction of strings constants [102, 65], as well as analyzing strings to extract API usage and permissions [103].

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 (app) and can be used to identify SQL injection attacks [58, 106, 50]. This analysis of literal values includes classification of apps as malware or benign based upon assessment of the

complete set of literal values present in apps [65].

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. Our research focuses 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. Importantly, these string structures offer obfuscation resilience. We show that when obfuscated by reflection, indirection, etc., string computations remain a resilient and effective feature set with which to classify malware and benign apps. This is an important characteristic as malware detection algorithms must generate a feature set that is both effective at classification but is not significantly disrupted when different obfuscators (reflection, indirection, etc.) are applied to the set of apps being classified. Here, a disruption introduced by obfuscation would be complete removal of the feature or changes in the frequency of the feature that make it less reliable as a discriminator to assist in malware classification.

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 that supplies the feature spaces to a machine learning (ML) classifier. 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 databases, 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, and obfuscation resilient 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.

- Analysis is performed in the presence of different types of obfuscations. We show that when reflection, goto, and indirection obfuscation are applied to a set of apps, the extracted feature set composed of obfuscated string computations continues to perform exceptionally well at classification. The obfuscators used have been identified by prior researchers as the most complex forms of obfuscation and thus most likely to degrade string computations as a feature set [59, 84].

- Lastly, the obfuscation analysis includes a rigorous analysis of changes to the features most important for classification. This reveals how obfuscators change feature sets and *why* accuracy remains the same or dips. Included is a comparison against another state-of-the-art malware detection tool to understand how its most important classification features are impacted by the same types of obfuscation.

Chapter 2

Related Work

2.1 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. It should be noted that most of what is referred to as string analysis is focused upon the different values that a string expression may take during the execution of a program [24] as opposed to the set of operations used to create the string itself.

2.1.1 String Analysis Focused on Values

The first holistic examination of string analysis in Java applications was performed by [33] 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. JSA creates a directed flow graph which is turned into a context free grammar (CFG). The CFG can be examined to identify the values of strings at different program execution points. Violist [68, 67] extended the work of JSA and was developed to represent string creation ops for both intraprocedurally and interprocedurally

created strings. It focuses on the introduction of configuration options that allow users to unroll loop structures and find values of strings at specific points of program execution. An implementation of JSA was also developed to check for errors in dynamically generated SQL query string for Java Web applications.

BEK [61] is an example of an automata-based string analysis algorithm. It enables precise analysis of Web sanitizer behavior that examine untrusted user supplied data to stop cross site scripting attacks. It uses a finite state automata to model the functionality of the sanitizer and can take a target string that is the desired malicious *output* text for the sanitizer, run the sanitizer in reverse, and then determine if it is a valid output as well as the form of the original string. This is a relevant example as it demonstrates how a vast majority of string analysis focuses on the values of literals and how the values themselves change at different points during program execution.

Our work differs from JSA and BEK 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 [17].

Research exists [60] 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, called Symbolic Finite Transducers, that symbolically execute the string's creation. Here, the research is focused on the string literal value and much less on the operation used in that literal value's creation.

[65] provides an extensive examination of a string literals where the literals themselves are treated as n-grams. N-grams are a concept used in computational linguistics for predicting the next item in a sequence. The research created n-grams of size one, two, and three and then performed classification on the different sized feature sets. This analysis is analogous to ours in the sense that it performs classification solely on the string literal values of all the string computations we would compute.

Researchers have also developed many static tools to examine sensitive data usage in apps where this data utilizes strings: FlowDroid [15], StubDroid [13], CHEX [73], AndroidLeaks [105], SCanDroid [51]. Barros *et al.* [22] present static analysis of implicit control flow for Android apps to resolve Java reflection and Android intents. Slavin *et al.* [93] propose a semi-automated framework for detecting privacy policy violation in Android app code based on API mapping and information flows. Yang *et al.* [108] develop a control-flow representation based on user-driven callback behavior for data-flow analyses on Android apps. Apposcopy [47] presents a semantics based approach to identify malware that is sensitive to the use of obfuscation. AppContext [109] extracts context information based on app contents and information flows and differentiates benign and malicious behavior. However, it requires manually labeling of a security-sensitive method call based on existing malware signatures. Type-based analyses have been used to detect privacy disclosure in Android apps as well [63, 64]. IccTA [69] and Epicc [81] are frameworks that perform ICC analysis to detect privacy leaks that may be careless or malicious in nature. BlueSeal [90] proposes a new permission mechanism that leverages static data flow analysis to discover sensitive data usage and improves Android app security. We believe our structure analysis of strings is complimentary to many of these approaches and is unique in that it analyzes the *structure* of the computation as opposed to only checking for the

presence of a particular feature.

2.1.2 Analysis of Obfuscation in Android Apps

A recent study of scholarly research into obfuscation [62] reviews 357 papers through 2018. It identifies the main reasons obfuscation is applied are to protect intellectual property in benign apps, or to hinder the process of extracting actionable information in the case of malware apps. The research also lists what specific characteristics of obfuscation have been proposed and studied by these papers. These characteristics may be binned into the following categories:

- Target of transformation,
- Level and stage, and
- Cost and effectiveness of the approach.

Our research focuses on the first item - target of transformation. While a great deal of research has been performed on the introduction of obfuscation and its ability to confound classification, little research has been focused on understanding the complexity obfuscation adds in terms of specific changes to the target structures of the code. Further, current research does not examine code transformations between different obfuscators and the impact these changes have on the extracted feature sets used for classification. We provide a summary of major obfuscation research areas in the following subsections.

2.1.3 Detecting Obfuscation in Android Apps

AndrODet [76] is obfuscation detection tool that can identify renaming, string encryption, and control flow obfuscation in Android apps. AndrODet achieves an accuracy of 92.02% for identifier renaming detection, 81.41% for string encryption detection, and 68.32% for control flow obfuscation detection. AndrODet uses an online machine learning approach -

continually learning from a stream of obfuscated and unobfuscated app examples. This provides a more robust implementation than a batch machine learning process which generates a static model at the end of the training step. Such a model is limited by the characteristics of the training set. AndrODet itself uses the byte code directly to train and test the model, versus an intermediary form or the original Java code.

A recent technique to understand obfuscation is the use of machine learning to classify obfuscation tool outputs. This can be done to identify specific obfuscator techniques $reflection, indirection, etc.$ or the obfuscation tool used[104]. Machine learning was employed with excellent result to predict which tool was used to obfuscate a particular Android app. The algorithm itself uses SVM for classification which provides good results but lacks explainability of these results. That is, the classifier can be trained to learn the differences between obfuscated and unobfuscated code but cannot explain the complexity of the differences or any specific details about the structural changes made to code. As with AndrODet, this research directly utilizes the bytecode for classification of obfuscated code.

Both of these research examples demonstrate how analysis of Android malware is spurred in part by its representation as bytecode [76]. This bytecode presents much more semantic information than machine level code and therefore makes it easier to analyze. Multiple tools exist to extract bytecode from apps and in some cases re-represent in intermediary from - including DexDump [41], Dex2Jar [40], Androguard [6], Apktool [9], and Soot [101]. Easy access to these tools feeds directly into the explosion of analysis tools focused on identification of behavioral characteristics of Android apps, both benign and malicious.

2.1.4 Measuring Obfuscation

An early examination of obfuscation [48] examined 240 apps under seven different types of obfuscation. This work is important as it is one of the first to tackle how to measure the complexity introduced by obfuscation using a variety of techniques. It provides the

following set of measures:

- Cyclomatic Complexity and Dependency Degree (CCD): Method corresponds to the number of the linearly independent circuits in the control flow graph (CFG) [75]
- Dependency Degree metric (DD): Defined with help of the dependency graph,which is constructed for the given CFG as follows. [23]
- Weighted Methods pro Class (WMC): the number of the methods defined in a class. [32]
- Depth of Inheritance Tree (DIT): the number of classes along the path from the given class to the root in the inheritance tree. [32]
- Number of Children (NOC): the number of direct subclasses.
- Coupling Between the Object classes (CBO): the number of coupled classes. [32] Here, two classes are considered coupled, if one of them uses methods or instance variables of the other one.
- Response Set for a Class (RFC): the number of declared methods plus the number of methods the declared ones call. [32]
- Lack of Cohesion in Methods (LCOM): Method analysis indicates long, incohesive methods with high degree of goto usage.[32]

All of these measure types give sheer numbers while none dive into the specifics of *how* the obfuscation tools work beneath. That is, anyone of these measures could be extremely high but if the underlying pattern used by the obfuscator is easily understood then the obfuscations value is lessened as it is easier to understand and subsequently defeated. An early study into Android [29] performed a holistic view of obfuscation where obfuscation was introduced and then measured in terms of these types. The work demonstrates that

these metrics can be used to successfully measure the net impact of obfuscation in Android apps.

A thorough examination of control flow obfuscation that is closer to the assessment of obfuscation our research targets is provided in [21]. This work examines multiple obfuscation tools and then measures the: (1) size before and after obfuscation, (2) net increase per app, and (3) a dynamic analysis for the running time of some of the obfuscated methods. Figure 2.1 shows how complexity is measured with multiple runs of obfuscation and a complexity increment measured at the completion of each run. The analysis does an outstanding job of showing the impact of obfuscation tools and different obfuscators in terms of net size and running time. However, this research does not examine the complexity introduced into the code. That is, if an app increases by 20% in size, what is that 20% attributed to? What is the nature of the structural patterns and how are they employed?

	Method proposed in [18] ($n_c = 3$)		Round 1 ($n_c = 15$)	Round 2 ($n_c = 12$)	Round 3 ($n_c = 9$)	Round 4 ($n_c = 6$)	Round 5 ($n_c = 3$)
App name	# obfuscated methods	Complexity increment	Complexity increment	Complexity increment	Complexity increment	Complexity increment	Complexity increment
Winamp	2,936	14,680 (+86.82%)	12,155 (+71.89%)	15,221 (+90.02%)	18,928 (+111.94%)	23,528 (+139.15%)	28,978 (+171.39%)
Wechat	13,350	66,750 (+117.73%)	65,909 (+116.24%)	81,057 (+142.97%)	98,723 (+174.12%)	116,475 (+205.43%)	139,305 (+245.70%)
Line	13,317	66,585 (+95.48%)	55,454 (+79.52%)	70,658 (+101.32%)	86,883 (+124.58%)	108,483 (+155.55%)	132,453 (+189.92%)
Sound cloud	6,022	30,110 (+102.48%)	18,700 (+63.65%)	24,706 (+84.09%)	32,450 (+110.45%)	44,354 (+150.96%)	55,869 (+190.12%)
Photo editor	2,154	10,770 (+98.45%)	10,693 (+97.74%)	12,849 (+117.45%)	15,621 (+142.79%)	18,925 (+172.99%)	22,455 (+205.26%)

Figure 2.1: Net Measure of Code Complexity Increase in Obfuscated Code

Research has been performed into code comprehension and creating metrics to represent the decrease in comprehension of code is [18]. These metrics mainly focus on net measure of complexity and avoid a detailed analysis of the new structure of the code.

Research also exists into the impact of obfuscation on code packing. Obfuscation analysis techniques exist that identify how an app is unpacked and loaded by the respective VM as malware developers take advantage of this to hide malicious code[66]. This area of research is not applicable to our obfuscation analysis as it focuses mainly on dynamic analysis and understanding how apps load into the VM and advertised entry points. These entry points are examined to identify when obfuscation is employed to hide malicious behavior

and intent.

2.1.5 Machine Learning for Malware Detection

Machine learning techniques are also very popular among researchers for detecting malicious Android apps. Classification of malware utilizing machine learning algorithms has matured to the degree that research is being performed to classify the types of machine learning algorithms being used. This research [4, 72] typically provides a hierarchy of the types of machine learning algorithms employed by current research, an example of such a hierarchy is provided in 2.2 as a Venn diagram, taken from Liu. It presents different types of machine learning where brackets are research that employs it for malware classification. Central to this diagram is the use of decision trees, both individually or as part of an ensemble machine learning approach. Our research employs decision trees as well and we explain later why decision trees work so well utilizing string structure as features for classification.

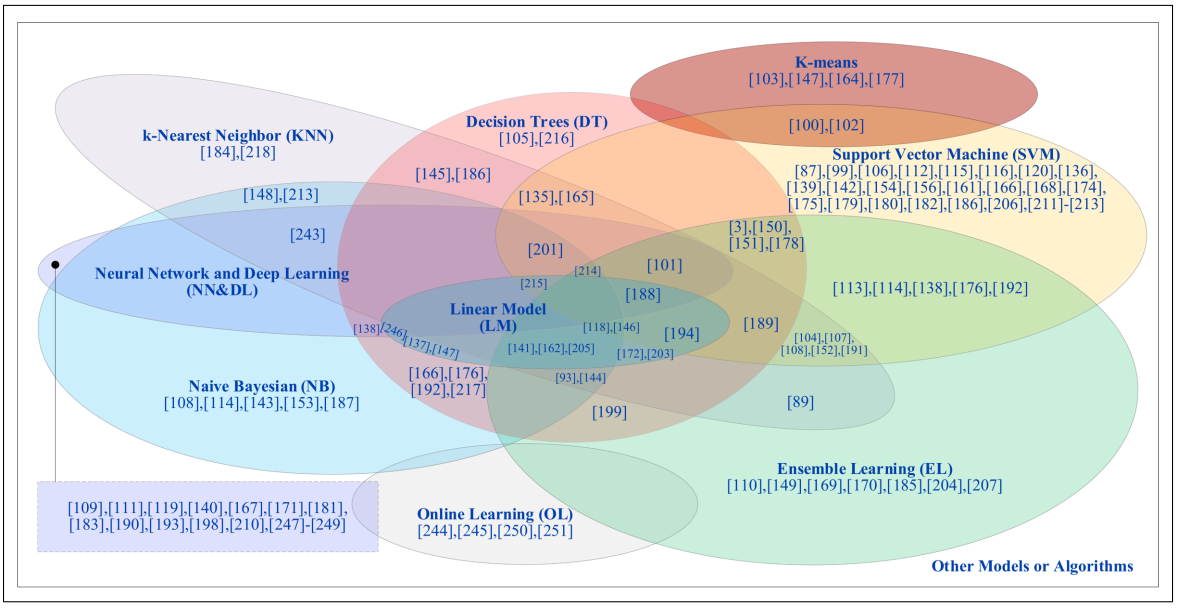

Figure 2.2: Venn Diagram of Machine Learning as Applied to Android Malware Classification

Most of these solutions identified in [72] train the classifier only on malware samples and can therefore be very effective to detect other samples of the same family. For example,

DREBIN [11] extracts features from a malicious app's manifest and disassembled code to train their classifier, where as MAST [30] leverages permissions and Android constructs as features to train their classifier. Shen *et al.* [89] 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 [27], and DroidAPIMiner [1], 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 [46] creates app signatures by leveraging control-flow and data-flow analysis. RiskRanker [55] performs several risk analyses to rank Android apps as high-, medium-, or low-risk. Sebastian *et al.* [83] analyze dynamic code loading in Android apps to detect malicious behavior. [43], [34] and [56] are all signature-based malware detection techniques and are designed to detect similar malware apps. [56] is of particular interest as it aims to represent know malware signatures as strings instead of hashes. Moonsamy *et al.* [79] 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.

2.1.6 Analysis of Impacts of Obfuscation on Classification

Research exists that accesses the impact of obfuscation on features used for classification. DroidSieve [96] identifies that extraction of lightweight syntactic features, otherwise known as signatures, are used heavily for successful malware classification. However, they are brittle in the sense that these signatures are susceptible to obfuscation. For example, individual API calls are excellent at classification of malware but are susceptible to obfus-

cation. The research creates two classes of features that should be resilient to obfuscation. First is resource-centric features like certificates, native code, etc. The second, and more germane to string analysis, are syntactic features. These include [96]:

- DEX-based Features. Tag each method based on the libraries it invokes from the Android Framework.
- Intents and Permissions. Scan the code to identify any explicit intents, which are used to start services within the same app.
- Evasion Techniques. Search for techniques that are frequently used to confuse analysis systems, i.e., native code, cryptographic libraries, or reflection.

DroidSeive then extracts these features and demonstrates that they are useful for malware classification - even when obfuscation is used upon syntactic features. The above listed features do work for classification in the presence of obfuscation for reflection, encryption and dynamically-loaded native code. However, the authors do not list what specific tools were used for obfuscation - rather indicating that the apps examined were obfuscated a priori. Our research seeks to understand the specific impacts of tools - namely how would specific obfuscators impact features used for classification, how detrimental is this impact, and can they degradation be overcome.

There are many examples of the application of machine learning classification on Android app string literals that are highly successful with accuracy rates typically above 98% [12, 71, 39]. Though successful, this feature set is especially vulnerable to encryption obfuscation as document by [74, 28]. This research shows it relatively easy to employ a basic string encryption technique to quickly drop the accuracy of classification using to string literals to no better than a coin flip.

A thorough taxonomy of obfuscators is provided in [36, 85]. Obfuscators are the type of obfuscation (reflection, indirection, etc.) while the obfuscation tools apply the obfuscator. A thorough analysis of the effects of obfuscation tools and obfuscators when applied

to Android apps subsequently analyzed by commercial and academic malware detection tools is provided in [84, 59]. The analyses demonstrate that obfuscation can defeat these detection tools but do not provide a detailed analysis of the differences in specific features that would lead to this degradation. An analysis of reflection [10] as applied by the commercial tool Proguard is the best example of a detailed analysis of how underlying code is changed by an obfuscation tool. The analysis does not include an assessment of how these changes effect feature sets used by tools that perform malware classification.

Researchers have investigated how to deobfuscate known obfuscated apps. Dong et al [42] provides mechanisms for the deobfuscation of encryption, reflection, identifier renaming, and encryption. Their results focus on singular changes in the source code versus control flow changes that show how sequences of code are transformed. Of note, the authors show how encryption works. They indicate that encryption keys may be included as part of the application - easy to use - or more commonly dynamically generated at runtime. This leaves static analysis unable to determine what the value of encrypted strings are. For reflection, they identified that malware and benign apps use reflection at the same rate. The main difference between the two classes is the complexity of how reflection is employed - where malware uses more complex patterns specifically to make static analysis more difficult for intent discovery. This characteristic is also seen in the complexity of renaming.

Chapter 3

Motivation

3.1 Motivation

String constants are of interest to developers of malware detection tools. For example, VirusTotal's [97] analyzed malware reports include a section titled "Interesting Strings" and the online malware detection service HybridAnalysis [5] 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 three 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.

```
1     public final String toString() {
2        StringBuilder sb = new StringBuilder("TxLocation{");
3        sb.append("level=").append(this.d).append(",");
4        sb.append("name=").append(getName()).append(",");
5        sb.append("address=").append(getAddress()).append(",");
6        sb.append("provider=").append(getProvider()).append(",");
7        sb.append("latitude=").append(getLatitude()).append(",");
8        sb.append("longitude=").append(getLongitude()).append(",");
9        sb.append("altitude=").append(getAltitude()).append(",");
10       sb.append("accuracy=").append(getAccuracy()).append(",");
11       sb.append("cityCode=").append(getCityCode()).append(",");
12       sb.append("areaStat=").append(getAreaStat()).append(",");
13       sb.append("nation=").append(getNation()).append(",");
14       sb.append("province=").append(getProvince()).append(",");
15       sb.append("city=").append(getCity()).append(",");
16       sb.append("district=").append(getDistrict()).append(",");
17       sb.append("street=").append(getStreet()).append(",");
18       sb.append("streetNo=").append(getStreetNo()).append(",");
19       sb.append("town=").append(getTown()).append(",");
20       sb.append("village=").append(getVillage()).append(",");
21       sb.append("bearing=").append(getBearing()).append(",");
22       sb.append("time=").append(getTime()).append(",");
23       sb.append("}");
24       return sb.toString();
25    }
```

Figure 3.1: Assembly of Location Information

3.1.1 Example 1: Obtain Geolocation Information

The app `DualSpace` [100] is used to coordinate multiple social media accounts for a user and is both a complement and competitor to the app WhatsApp. It was first identified in September 2018 with 16 of the VirusTotal's 63 detection engines flagging it as malware. These engines list it as belonging to malware families focused on Adware. Its specific risks include that it checks GPS information, contains a Linux executable, uses reflection, and accesses telephone details like imei and Mac address. We reviewed the string computations generated by our analysis of `DualSpace` and identified multiple computations that were suspect related to collected GPS information. We provide decompiled code for one such GPS-centric string computation in Figure 3.2.

The code listing shows how DualSpace collects as much information about a user's location as possible, using a `StringBuilder` object to hold the information as it is accumulated. This behavior may seem benign on the surface but DualSpace's download page sheds light on the application's true intention. The following is an excerpt from the download page:

> *DualSpace applied for as much system permissions as we can to ensure that applications cloned in Dual Space will run normally. For example, if Dual Space is not permitted to acquire camera permission, you will be unable to use camera function in some apps that run in Dual Space. Dual Space does not collect your personal information to protect privacy [94]*

This disclaimer indicates that the app simply needs to obtain the permissions to run the clone application - not for itself. However, this harvesting of user location information is *inside* the DualSpace code itself. It has no need to collect this information as each cloned social media app is able to collect it independently.

3.1.2 Example 2: Useless String Ops

The app `App1SQL` [98] 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 3.2 via an abbreviated extract of the app's actual code.

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

```
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 3.2: Useless String Operations.

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 4.1.

3.1.3 Example 3: Dynamic Code Loading

The app `App2Dex` [99] 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 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 3.3 via an abbreviated extract of the app's actual code.

Figure 3.3 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 three examples demonstrate that a manual examination of a string's computation

```
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 3.3: DexClassLoader Example

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. Our research focused 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 baseline 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 solely?
- 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?
- Research Question 5: How would results differ for classification if string computation subgraphs were used for classification as opposed to the entire string computation?
- Research Question 6: What would classification results look like if we simply counted the use of string API calls (i.e. calls to String, StringBuilder, StringBuffer, etc.)? Would this simple approach do better or worse than classifying upon the string computations themselves?

Chapter 4

Algorithm Implementation

4.1 Analyzing Strings

We implement our string analysis as a flexible system capable of quickly processing and analyzing apps. Figure 4.1 shows the major components of our system and their integration. The following subsections provide detail on each of these components.

4.1.1 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 [107, 102]. 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

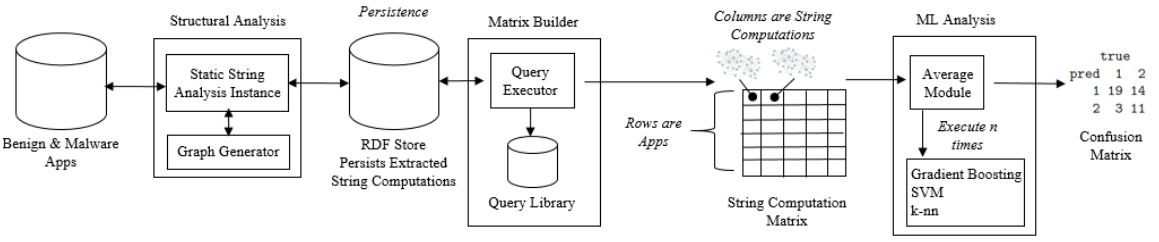

Figure 4.1: String Analysis System

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 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 4.2 provides a pictorial representation of how the interprocedrual analysis works using a code snippet taken from `com.symantec.mobilesecurity`.

The interprocedural analysis uses as input decompiled code provided by Soot, a Java instrumentation framework [101]. 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 1 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

```
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 4.2: Interprocedural Constant Propagation

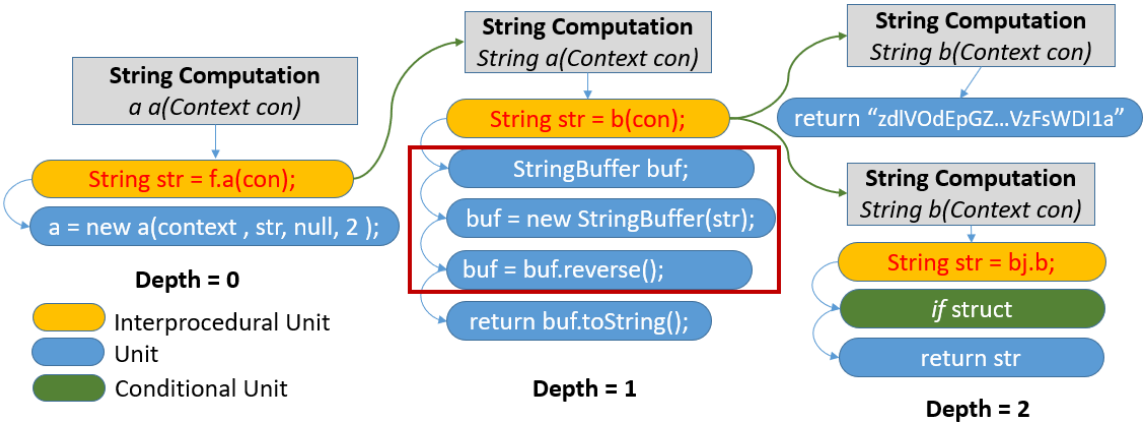

Figure 4.3: String Computation Graph Structure

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.

4.1.2 Representing and Reasoning About Stings as Graphs

Figure 4.3 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 3.1 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.

Our persistence step, shown in Figure 4.1 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 4.3. 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 [37]. We leverage SPARQL's advance capabilities to perform graph pattern matching to find patterns of

```
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 4.4: SPARQL Example

interest. Figure 4.4 shows a graph query that pulls out portions of the graph presented in Figure 4.3. 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.

4.1.3 Feature Set Development

API Usage Count Matrix: Our first set of features simply counts the occurrences of Java string API calls per application under evaluation. We reviewed Java's API documentation and multiple open source articles on how Strings are created in Java to understand all the different possible ways a string comes into existence. This led us to capture as features

Table 4.1: Example Core String Feature Set

	String(byte[])	String(byte[], int)
App1	0	0
App2	4	1
App3	1	2

all the information related to the String, StringBuilder, and StringBuffer APIs as well as instances of creation of byte[] and char[] arrays. These array structures are often starting points to strings not created from String literals. These string operations fall into the following categories:

- **Initializations** - These include the initialization of String, StringBuilder, StringBuffer, CharSequence, char[], and byte[]. The later two are included as they may be used as inputs to string initialization operations and our graph analysis collects and connects them to constructors when that is the case.
- **Appends** - These include the append operations of StringBuilder, StringBuffer, and the + operators used to concatenate Strings and StringBuilders.
- **Modifications** - These include the modification operations of StringBuilder and StringBuffer such as insert, replace, and remove.
- **Ancillary** - These include all other operations in the StringBuilder, StringBuffer, and String classes.

We develop a matrix of 126 features where each column represents a Soot decompilation unit from one of these categories of string operations. When we say feature, we mean a call to a method in one of these APIs. Table 4.1 provides an example for two features, the `String` constructor that accept `byte[]` and `byte[],int` respectively. We simply count how many times the app uses either one of these `String` constructors. Here, App1 never uses the constructor while the remaining Apps do.

We use this approach for two reasons. First, it provides us with a relatively small feature set that is rich with the structural elements that compose our string computations. Recent research performed by Garcia et al. [52] demonstrates that highly effective classifiers can be developed using such smaller sized feature sets specifically to avoid the scalability issues of larger feature set. Second, it provides a simple baseline for later classification as we can compare classification using string computations versus just simply counting the occurrence of string API calls and classifying on those calls.

Complete Graph Matrix: 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.

We then construct a matrix where each header represents a string computation and the matrix rows represent the apps. This is shown in Figure 4.1. 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 7.1, 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 4.5 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.

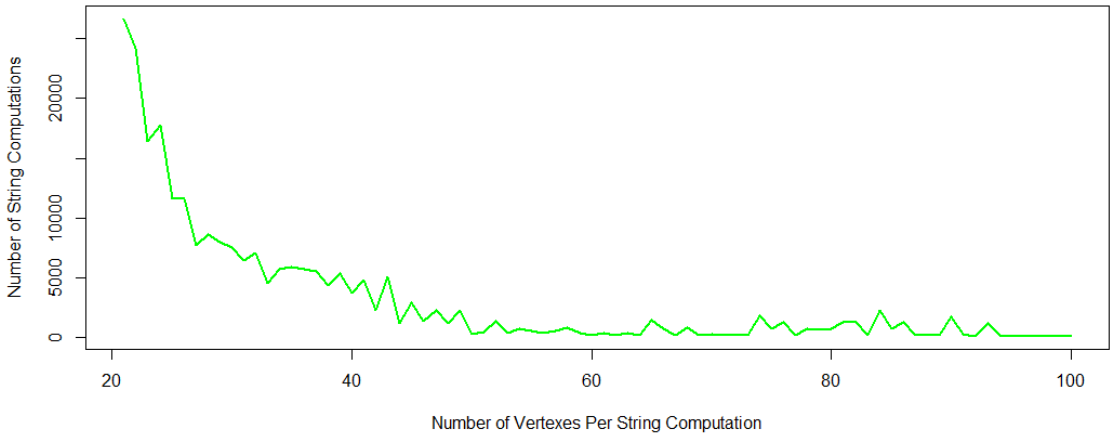

Figure 4.5: Frequency of String Computation Sizes

Subgraph Matrix: We next create a matrix that using subgraphs as opposed to the complete graph as the feature. That is, we create a matrix where each row represents an app and the columns represent subgraphs of three or six, sequential, connected Soot units. We record the number of times an app exhibits each subgraph pattern in the matrix. The subgraphs themselves are created as follows. We take all the string computations for an app and chunk the graphs into subgraphs with three vertexes and two connecting edges and then six vertexes and five connected edges. For example, for the StringComputation presented in the center of Figure 4.3 we would create one subgraph with three vertexes as follows in Figure 4.6:

```
1 Vertex 1:{StringBuffer: StringBuffer reverse()}
2 Vertex 2:{StringBuffer: String toString()}
3 Vertex 3:{return String}
```

Figure 4.6: Three Vertex Subgraph Example

Breaking the computations into smaller subgraphs allows us to normalize the string computations while still capturing the critical sequence of structural information. Note, all information related to Soot's record keeping is stripped from these subgraphs as well as any reference to actual string values or literals. Thus, the subgraphs contain pure structural information and nothing else.

This approach leads to matrices with large numbers of columns. For example, an analysis of 200 malware and 200 benign apps generated over 273,000 distinct subgraphs with three vertexes, meaning our matrix also has this many columns. We apply a sparsity metric to reduce the number of columns with sparsity set to 80%. That is, we only include a subgraph as a feature if 80% of the total apps for the analysis exhibit the feature. This greatly reduces the number of columns for our matrix. The number of features is typically between 250 and 400 for an analysis of 2,000 total apps.

Complete Graph Matrix: Our final feature set include complete graphs as features. That is, we treat each string computation graph as feature. In terms of the feature matrxi, each column header represents a string computation, each row an app, and each cell is the count of how many times that application exhibits the string computation. Counts of zero indicate the application does not exhibit the string computation. As with the subgraphs approach, this lead to a huge number of features in the feature space. We apply the same sparsity metric as applied to subgraphs to winnow the feature set to between 250 and 400 features dependent on the set of malware and benign applications under examination.

One note on the complete graph matrix. We only include interprocedural graphs or intraprocedrual graphs that are not a part of any interprocedural graphs. If a interprocedural graph is composed of four intraprocedural graphs we will *not* consider the four intraprocedural graphs in our feature matrix. We seek to only include in this feature set complete graph structures, not the individual intraprocedural parts in the case of interprocedurally defined string computations.

4.1.4 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 and again in 7.1 when machine learning processes are run on obfuscated applications.

Chapter 5

Results

We address each of our research questions defined in 3.1 in this section. 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[49], 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

Table 5.1: Overview of Data Sets Used for Classification

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

and machine learners. GB dominates winners circles for Kaggle competitions that involve structured datasets [3].

We optimize the hyperparameters GB uses as follows. Trees are grown starting with a tree depth of zero and grow greedily [31]. We run GB with tree depths of 10, 15, 20, 25, and 32 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 access 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 Characterization of the Datasets

Table 5.1 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 [2], 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 [97], 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 5MB 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 [80]. Selecting apps only from Google plays gives us the best chance to have a truly benign data set with no unidentified malware 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.

One note with respect to the benign and malware apps used in the analysis. The set collected from both Androzoo and VirusTotal contain apps that are freely available. That is, the apps did not have to be purchased. Our results are therefore limited by the fact that only freely available apps were used and we cannot report on differences for string analysis

for purchased apps.

Obfuscated Apps We next take a subset of apps for both malware and benign and run them through two obfuscation tools. We choose as as our obfuscation tools: (1.) The Automatic Android Malware Obfuscator (AAMO) [84] which has been used to demonstrate how virus detection tools perform against obfuscation, (2.) DroidChameleon[86] also used to demonstrate the performance of virus detection tools in the presence of obfuscation, and (3.) Obfuscapk[35], a recent open source obfuscation tool with a large community following. We configured the obfuscators to use obfuscation techniques that Hammad et al. [59] identified as challenging for virus detection methods.

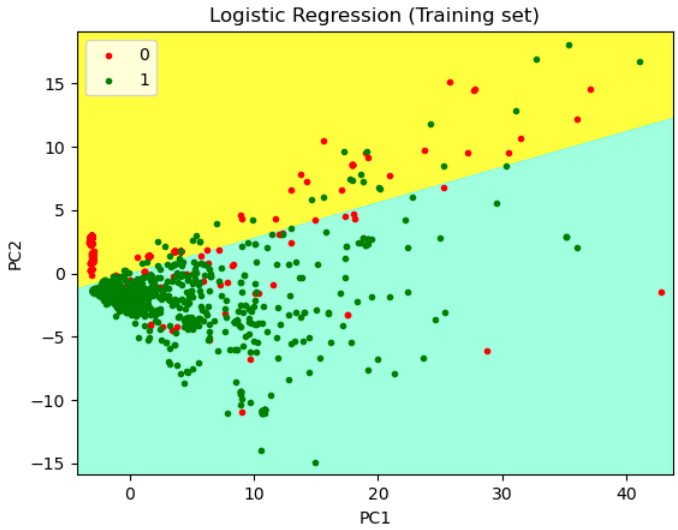

Figure 5.1: PCA First Two Components

We next applied PCA to the data sets using the feature set described in Section 4.1. Our feature matrix composed of string computations is high-dimensional data, where a sample of 1000 apps from either the pool of benign or malware apps would identify over 5000 features. We used PCA to visualize the relationship between the features as well as understand the main variance in the data. 5.1 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. 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.

5.3 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 [52]. It's 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 [70, 95]. 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 [52].

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 [65, 102] have utilized string literals to do classification with very good results - reaching F-measures of 97%. Similarly most modern anitvirus software leverages string literals.

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 5.2.

RQ1 Answer: The results demonstrate that string computations by themselves perform competitively against classification using RevealDroid or string literals. Our approach is

Table 5.2: Baseline Comparison

Tool	Precision	Recall	F-measure
String Computations	96.62	97.27	96.94
RevealDroid	91.42	96.96	94.11
String Literals	94.30	96.12	95.20

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.

5.4 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 5.2 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 [14] were included as another feature upon which to perform classification [53]. 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 how our combined approach of string computations and RevealDroid and string literals in terms of precision, recall, and F-measure in Table 5.3.

RQ2 Answer: The results demonstrate that in both cases the combination of string computations with RevealDroid or with string literals does better than either alone. This

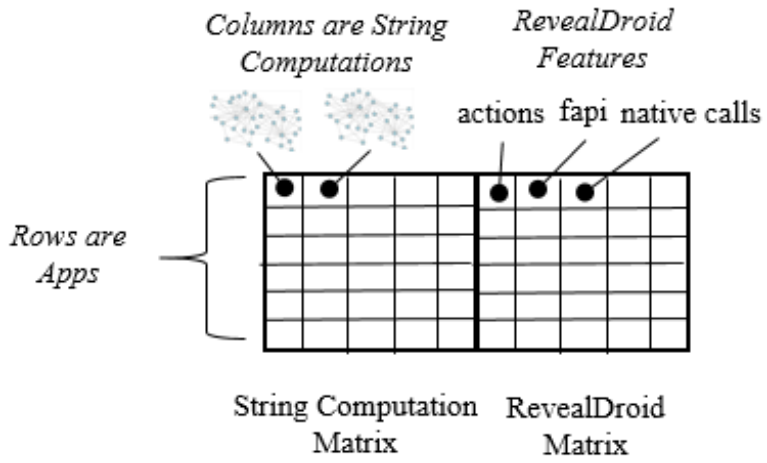

Figure 5.2: Combined String Computation and Reveal Droid Matrix

Table 5.3: Combined Approach

Combination	Precision	Recall	F-measure
RevealDroid and Computations	95.30	97.26	96.27
Literals and Computations	95.68	97.08	96.37

is expected for RevealDroid as it 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. This outcome also expected for string literals. Apps create many string computations internally where the literal value is only known at runtime. Combining the literal values with the string computations increases the overall F-measure. In both cases the F-measure's of the combined approach are very near the best F-measure for classification with string computations only.

5.4.1 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 [86], an obfuscation tool with various trivial and non-trivial techniques to obfuscate malware applications, (2) AAMO [84] 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 [59, 20]. 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. [59], 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 [59], 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

Table 5.4: 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	95.28	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	94.0	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

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 [53]. 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 5.4 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 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. Obfuscapk's 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 ob-

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

Figure 5.3: PackageManager Example

fuscator 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.

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 the R, caret package's built in `varImp()` function to return the most important features for GB models for our string computations run on the unobfuscated apps. Figure 5.3 provides a snippet of the string computation that is most important in classification of malware and benign apps.

5.6 RQ4 Answer:

The importance factors indeed shed light on why strings are important to malware. Consider Figure 5.3 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 [77]. 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 [82]. Also note that there is no associated string literal for the string computation.

5.7 RQ5: Comparison to ML Classification on Subgraphs

The goal of this research question is to understand if it more effective to classify on complete string computations or to break the string computations into subgraphs with a set number of vertexes and classify on the subgraph patterns. The potential value of these subgraph patterns is that they better normalize the feature set across the apps. For example, if malware developers *independently* utilize smaller patterns in building and utilizing strings for malicious intent, then smaller sized subgraphs would capture these patterns and accuracy of the results would increase. An example might be to append information to a `StringBuilder` object and then immediately call `reverse()` before calling `toString()`.

To test this idea we randomly choose as input 8,000 apps from our dataset, 4,000 malware and 4,000 benign. The string computations for the 8,000 apps are broken into subgraphs of size three vertexes and then six vertexes. These are the smaller sized subgraphs we hope will capture fine grained examples of malicious behavior across malware developers. A GB model was built for both and we report on how our string computations, and string computations broken into subgraphs of size three and six vertex perform in terms of precision, recall, and F-measure in Table 5.5.

Table 5.5: Subgraph Comparison

Tool	Precision	Recall	F-measure
String Computations	96.62	97.27	96.94
Three Vertex	92.17	96.41	94.08
Six Vertex	90.72	92.25	91.60

5.8 RQ5 Answer:

We see that subgraph string computations do no reach the same f-measure as does using the complete string computations. Breaking the computations into smaller sizes creates no advantage with respect to ML classification. This is likely because the smaller graph sizes

in effect remove some of the important features for classification - in effect normalizing features across the two classes too well and removes larger string computations useful for classification.

5.9 RQ6: Comparison to Ml Classification on String API Calls

This question seeks to answer if a simpler approach toward string analysis might achieve the same or better results than string computations. If we simply count the number of times string related APIs are called by an app - can that set of features outperform ML using string computations? The graph computation vertexes represent operations performed by the Java string APIs (i.e. `java.lang.String`, `java.lang.StringBuffer`, and `java.lang.StringBuilder`) as strings are built and used. We create a matrix that sums the number of times an app makes a call to one of these APIs methods (i.e. `java.lang.String reverse()`, `java.lang.StringBuilder append(float)`, etc.) This matrix simply reports on how often apps call the string API methods and has 117 features in total. Table 5.6 provides an example of this matrix. There are four apps. For App1 it makes 44 total calls to `java.lang.String reverse()` and 12 calls to `java.lang.StringBuilder append(float)` and so on for all the methods available in the Java string APIs.

Table 5.6: Matrix of Aggregate String Computations Per App

AppID	String reverse()	StringBuilder append(float)	...
App1	44	12	
App2	0	17	
App3	10	0	
App4	0	1	

The advantage here is that it is significantly simpler to count the occurrences of string API calls than to create string computations. It is very similar to the approach used by

RevealDroid to simply count API calls they have identified to be sensitive in the Android system libraries. We randomly choose as input 8,000 apps from our dataset, 4,000 malware and 4,000 benign in terms of precision, recall, and F-measure in Table 5.7.

Table 5.7: Counts Comparison

Tool	Precision	Recall	F-measure
String Computations	96.62	97.27	96.94
API Counts	91.31	92.49	91.78

5.10 RQ6 Answer:

Our results show that classification using API counts alone exceeds 90% f-measure but is worse that classification that utilizes string computations. The results are of interest because, despite not being as good as string computations, it is clear that there is a difference in how benign and malware apps use the string APIs. It helps validate the results we see for string computations.

5.11 Summary

Our initial set of research questions and answers established that string computations are an effective feature set to utilize in the classification of Android benign and malware apps. They perform well against other state-of-art techniques and are resilient to different forms of obfuscation. The latter point identified a new set of research questions that focused on a more thorough analysis of the impacts of obfuscation - both on our string computations and state-of-the-art techniques. In 6.1 we analyze how string computations are impacted by different obfuscation tool's implementation of obfuscators like reflection or call indirection. Obfuscators do have different implementations and these differences impact how string computations are generated. In 7.1 we examine how the different obfuscation tools and their obfuscators impact the most important features used for classification. We perform a

novel analysis that answers *why* string computations are obfuscation resilient. It reviews obfuscation's impact on the most important features used for classification. The analysis shows how different obfuscation tool's implementations of specific obfuscators are not problematic for harvesting the string computations most effective at malware classification.

Chapter 6

Obfuscation Impacts on Code

6.1 Obfuscation

Malware detection algorithms focus on development of a feature set useful in the detection of malware. Obfuscation is the malware developer's complement to this approach. It attempts to hide, or obfuscate, features of the application that would differentiate it from benign applications - those with no intent to harm users who install the app. Creators of malware detection algorithms are aware of this and attempt to account for fact. Typically, they perform machine learning classification on their tool's generated feature set utilizing a set of apps that are clean (or unobfuscated) and then obfuscated. This analysis approach is shown in [52, 16, 91] and many others. The outcomes of the approach are also very similar; results for clean and obfuscated apps show only a small decrease in accuracy for classification of the obfuscated set. The malware detection algorithm is termed 'obfuscation resilient' based on the perceived insignificance of the accuracy decrease.

However, this approach is not rigorous enough as it lacks a detailed examination of the *features* themselves. How did the obfuscation impact the features output by the detection algorithm? Further, how was the feature set most important to classification impacted by the obfuscator? This type of analysis is needed as it is precisely what a malware devel-

Table 6.1: 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	95.28	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	94.0	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

oper will do. They will understand the detection tool and analyze the feature set extracted to determine how their malware is being detected. Current, state-of-the-art, obfuscation tools now include obfuscators not meant to simply obfuscate the semantics of the code but specifically to defeat malware detection tools based on their understanding of detection algorithm feature sets. Obfsucapk [35] includes an obfuscator targeted to alter the feature sets most commonly relied upon by malware detection tools by hiding calls to Android sensitive operations. This is a significant milestone; an obfuscation tool that directly appeals to malware developers needs. Malware detection researchers must now consider their feature set not only in terms of the current state-of-the-art obfuscators but how resilient the feature set is when it is examined and understood by malware and obfuscator developers.

Consider the results provided earlier in our analysis of malware detection tools when obfuscators are applied. We present a summary of these results in the table Table 6.1 for three different obfuscators with different types of obfuscation where gradient boosting is used as the ML algorithm for classification.

There are three examples in this table where an analysis of the feature set explains the result shown for an obfuscator. A trivial example is the impact of string literal encryption on the ML model's performance. It has little impact on string computations but has a

significant impact on RevealDroid and string literals. The reason is obvious for string literals - a unique key is generated for each application to encrypt the string literals within the Java code as well as in the Android asset files. This guarantees that the encrypted string literals equate to different values across the encrypted apps. The table shows that a gradient boosting model cannot be developed for encrypted string literals as there is no commonality between the encrypted string literals across apps.

RevealDroid also suffers from basic string encryption. However, it is not entirely because of encryption applied to the Java code itself. RevealDroid harvests features from asset files within the Android app and the encryption used by Obfuscapk encrypts these asset files as well as the original source code. This provides a near six percent drop in precision for RevealDroid.

Advanced reflection, a sub-class of reflection implemented by Obfuscapk, is an example of a targeted obfuscator specifically engineered to degrade the features most important for malware classification by detection algorithms. Its developers [19] established a static runtime model of the Android's framework to study its internals and identify the framework's protected resources. It is these protected resources that malware detection developers focus upon - and that Obfuscapk purposefully attempts to obfuscate. RevealDroid shows a six percent drop.

We next perform a deep dive analysis of different obfuscation tools and obfuscators to understand, specifically, how they impact our string computation feature set. This information will help us to interpret the results provided in section 7.1 where we examine classification results when obfuscation is applied across thousands of apps.

6.1.1 Obfuscators Utilized

Two obfuscators were used for the analysis - Obfuscapk and AAMO. These obfuscators provide a set of characteristics necessary for a rigorous analysis. Both Obfuscapk and AAMO offer advanced obfuscator routines that other, freely available obfuscators do not -

such as DroidChameleon. These include reflection and indirection. Garcia [53] performed a survey of obfuscators and their impact on classification. They identified that few tools exist that offer these capabilities outside of commercial tools. Georgiu[35], one of Obfuscapk's creators, also identified a lack of open source obfuscators with advanced obfuscation capabilities and pointed to AAMO as one of the only other freely available candidates. Tools do exist in the commercial space with these capabilities, such as DEXGuard, but are not available for this research. Reasons include cost of the tools and permissibility of use. DEXGuard was contacted to see if an academic version of the tool is available and they replied it was not and, further, that their license prohibits academic analysis of the tools to be published in journals or conferences.

Both AAMO and Obfuscapk have a development community that is open to questions and willing to provide detailed replies in a timely manner. Obfuscapk is provided on github and all questions submitted to the project were answered in under six hours. The authors of AAMO replied to emails about functionality of the code base within one day of submission. This responsiveness makes a tremendous difference in analysis as questions arise about obfuscator functionality that need input from the authors of the obfuscation tools. DroidChameleon authors were contacted multiple times about code functionality and were unresponsive. Therefore this tool was not considered to be part of the evaluation as questions that needed clarification to accurately assess the tool and reports of its use in other academic research could not be verified.

A Soot backward flow analysis was implemented to capture the output of both obfuscation tools. The exiting string analysis implementation was modified to capture control flow graphs for each method identified as reachable by Soot. These graphs were serialized and then written to disk. This technique allowed for an app to be examined and persisted prior to obfuscation and then, again, after the application of obfuscation. A set of comparator utilities were created to:

- Identify methods in the original and obfuscated apps that had identical control flows,

- Identify methods in the original and obfuscated apps that had different control flows as well as indicate where the differences occurred,
- Identify methods in the original app not in the obfuscated app, and
- Identify methods in the obfuscated app not in the original app.

This core set utilities proved extremely useful in analysis and examination of original and obfuscated apps.

6.1.2 Analysis

A data set consisting of applications that were non-obfuscated were needed as ground truth for the obfuscation examination. These needed to be applications that were not obfuscated, whose third party libraries were not obfuscated, and for which the original code was known. The last point was necessary as compilation into DEX code using current Android Studio applies light obfuscation via the R8 compiler [54]. For example, it shortens the name of classes and members for smaller overall DEX size. Selection of an arbitrary application from Google Play, for example, might exhibit this type of obfuscation and it would be difficult to determine if the obfuscation is from the R8 compiler or from a commercial product. Compiling from the original source code found on git alleviates this problem and allows the original form of the code on the date of the compilation.

The analysis utilized four applications made available on github by Facebook. These apps are found at the following URL: https://github.com/facebook/facebook-android-sdk with tutorials provided for application developers on how to integrate Facebook functionality into Android applications. The apps themselves contain reasonable amounts of business logic, are easy to follow, and clearly are of benign origin. Facebook provides the set of unobfuscated code we require, both in terms of the main code of the application and the supporting third party libraries most of which are from Facebook. Table 6.2 provides statistics relevant to the deep dive apps. The total lines of active code measures how many

Table 6.2: Statistic for Deep Dive Apps

Name	Lines of Code	Reachable Methods	# SCs
HelloFacebookSample	127,988	8,827	2,870
MessengerSendSample	83,584	5,783	1,883
RPSSample	185,487	12,788	3,072
SwitchUserSample	121,243	8,112	2,579

lines of code exist within the entire app. The number of reachable methods in terms of graph reachability for the apps' call graph. Reachability [87] is well researched. Though an app may call a method, it may not be reachable as part of Soot's static analysis as the method was called via Java reflection or possibly dynamically loaded. The total number of string computations generated by the apps is also included. These statistics will serve as a baseline for later analysis.

Randomness and Limitations of Introduced Obfuscation

The first question answered by the research is a generic question related to obfuscators; do obfuscators introduce obfuscation randomly, or is repeatable, deterministic process? Intuitively it would seem that obfuscators would introduce obfuscation randomly in an effort to make it difficult to reverse engineer outputs. The analysis did the following:

1. Each of the four apps used for the deep dive analysis were obfuscated using each obfuscator. The results were saved.

2. Step one was repeated and the results saved to a different location.

3. The obfuscated apps were compared against each other. That is, the two HelloFacebookSample apps obfuscated with reflection were compared against each other.

4. The analysis looked at the number of lines of code, reachable methods, and generated string computations.

The results were identical for both obfusators and for each type of obfuscation such as reflection or goto. The only identifiable difference were the random names applied to newly

introduced methods. This proves that for these two obfuscation tools, application of obfuscation is deterministic and identical. Each obfuscation tool's source code was examined to understand why this is the case. Both tools utilize smali[57], an assembler/disassembler for the DEX format used by Dalvik. Smali is utilized by many tools, including Soot itself, to disassemble code. The disassembled code is provided by smali as files where the obfuscator tools iterate over each line of smali code and perform operations upon it. In this respect, smali itself is deterministic and will always present the disassembled code for an app in the exact same order.

There is still an opportunity for the obfuscation tools themselves to add randomness in spite of smali's deterministic disassembly. Both obfuscation tool's source code was examined and neither did this. Figure 6.1 demonstrates how AAMO iterates through the smali code, identifies opportunities for injection, and then injects. No randomness is involved in when, in this case, reflection is applied. Of course, it is possible for a developer to introduce their own randomness for when obfuscation is applied but, out-of-the-box, the obfuscation tools always obfuscate in a deterministic fashion.

There are also limitations to the application of obfuscation both in terms of its introduction into code and performance. All obfuscation tools are constrained by the fact that the total number of references that can be invoked by code with the DEX bytecode file is 65,536 [54]. That is, an Android app may only reference 65,536 methods if it uses a single DEX file. Obfuscapk is aware of this limitation and, as a rule, monitors how many new method references have been introduced and simply does not allow that number to exceed the upper limit. AAMO ignores this limitation and will create apps that violate this limit. This is problematic as it leads to creation of apps that cannot be successfully processed by Soot. To fix this issue the AAMO code was modified to ensure that it obeyed the upper limit for introduced, new methods.

A separate and more pragmatic limitation is the effect of obfuscation on application performance. The inclusion of reflection is known to slow applications but other impacts

```
1
2     def change_all_method(smali_file, new_method,
          all_method_list):
3         """Redirect all the method calls"""
4         for smali_line in u.open_file_input(smali_file): # For
              each line
5         class_match = re.search(...) # Match the class
              declaration
6         if class_match is not None:
7             class_name = class_match.group('className') # Find
                  the class name
8         invoke_match = re.search(...)
9         if invoke_match is not None:
10            if not is_init(invoke_match.group('invokeMethod')):
11                change_match_line(smali_line, ...)
12            else:
13                print smali_line, # Print the line unchanged
14        else:
15            print smali_line, # Print the line unchanged
```

Figure 6.1: Access of Disassembled Code in Obfuscation Tools

exist as well - including exhaustion of system resources if introduced obfuscation is too complex [26]. The developers of Obfuscapk are aware of these limitations and add in obfuscation to a level that does not significantly degrade application performance [35]. The developers of AAMO do not provide such controls outright but it is relatively easy to modify the source and limit the amount of introduced obfuscation to a target level.

Analysis of Reflection Obfuscator

Both Obfuscapk and AAMO provide a reflection obufscator. Documentation for Obfuscapk and a review of its source code base indicate that it will apply obfuscation as follows [35]:

- The method has public visibility,
- The method is not a constructor,
- The method is not part of the Android framework, the reason for this discussed in the AdvancedReflection subsection, and

- There are enough method invocations remaining.

In contrast AAMO indicates that it performs reflection only on static method calls, replacing the static calls with reflection calls. This difference demonstrates that obfuscation tools may use a kind of obfuscation, for example reflection, but its implementation may be drastically different.

The analysis begins with Obfuscapk's reflection focused first on the lines of active code and reachable methods. Table 6.3 provides reachability results and net changes for Obfuscapk reflection. The table shows an increase for all four applications in the amount of active code and demonstrates that use of reflection requires introduction of code that, in this case, exceed the amount of code removed due to reachability.

Results also demonstrate that thousands of methods are removed from Soot analysis as their reachability is broken by the introduction of Java reflection. Obfuscapk only introduces two new methods into each application. These new methods are proxy methods that centralize the introduction of reflection into the applications. Figure 6.2 shows the mechanics of this proxy process. A new class named `ApiReflection` has a large static intitializer where the obfuscator places method calls for reflection in a `List` object. A single accessor method accepts the location of the method to be called within the list and this accessor method is placed in the app's classes where the newly introduced reflection is called. The accessor method accepts an integer argument that is the precise location in the `List` of the method to be reflected upon. Interestingly, the import section of the Java code for `ApiReflection` may become enormous as it includes imports for classes across the entire app.

```
public class ApiReflection {
    private static final List<Method> obfuscatedMethods = new ArrayList();

    static {
        try {
            obfuscatedMethods.add(MediaBrowserCompat.ConnectionCallback.class.getDeclaredMethod("onConnected", (Class[])
            obfuscatedMethods.add(MediaBrowserCompat.ConnectionCallback.class.getDeclaredMethod("onConnectionFailed", (
            obfuscatedMethods.add(MediaBrowserCompat.ConnectionCallback.class.getDeclaredMethod("onConnectionSuspended",
            obfuscatedMethods.add(MediaSessionCompat.class.getDeclaredMethod("ensureClassLoader", new Class[]{Bundle.cla
            obfuscatedMethods.add(MediaSessionCompat.class.getDeclaredMethod("ensureClassLoader", new Class[]{Bundle.cla
```

Figure 6.2: Obfuscapk Proxy Method to Call Reflection

Table 6.3: Obfuscapk Reflection Impact on Lines of Active Code and Methods

	HFS	**MSS**	**RPS**	**SUS**
Lines Active Code in Original	127,988	83,584	185,487	121,243
Lines Active Code in Obf	129,360	82,160	185,207	121,792
# Methods in Original	8,827	5,783	12,788	8,112
# Methods in Obf	7,425	4,281	11,196	6,639
# Original Method Not Reachable	1,404	1,504	1,594	1,475
# Methods in Common	7,423	4,279	11,194	6,637
# Methods in Common Changed	1,518	1,135	2,331	1,476
# New Methods in Obf	2	2	2	2

Obfuscapk's reflection introduction impacts the string computations extracted from the apps. Table 6.4 shows how string computations are impacted across the four apps. There is a notable jump in the number of string computations in the obfuscated apps versus the original apps. This is directly attributable to how Obfuscapk calls reflection using the `ApiReflection` class. In the case of these specific apps, thousands of new string computations are added as each method is now a string literal that will be invoked via reflection. These new computations are not overly complex - simply two vertex graphs with one node the string literal value, the other its addition to the `Method` object it is added to. Also note the last two rows of the table which examines the effect of reflection on string computations where the Java reflection is introduced. The total number of string computations extracted from the methods that changed are nearly identical. This is due to how Obfuscapk introduces the reflection. The existing string computations in the obfuscated app are the same as those in the unobfuscated app with the exception that certain vertexes are swapped from their former calls to new calls in the `ApiReflection` class. Figure 6.3 demonstrates this effect. The top code sample is from the obfuscated HFS app, the bottom its original form. The red boxes show the same string literal created in both. String computations will, in both cases, be extracted the same way. There is no splitting of computations or removal. However, there are a few edge cases that add string computations. For example, if reflection accepts a `StringBuilder` object and a string literal to append to it, the string literal will be viewed as a new string computation though it is unified with the `StringBuilder`

object via the reflective call.

Table 6.4: Obfuscapk Reflection Impact on String Computations

Reflection	**HFS**	**MSS**	**RPS**	**SUS**
String Computations in Original	2,870	1,889	3,083	2,588
String Computations in Obf	5,731	4,568	5,962	5,409
# String Computations Original Removed	175	374	188	215
# String Computations New Methods	3,035	3,049	3,067	3,035
# String Computations Orig Changed Methods	800	583	882	761
# String Computations Obf Changed Methods	801	587	882	762

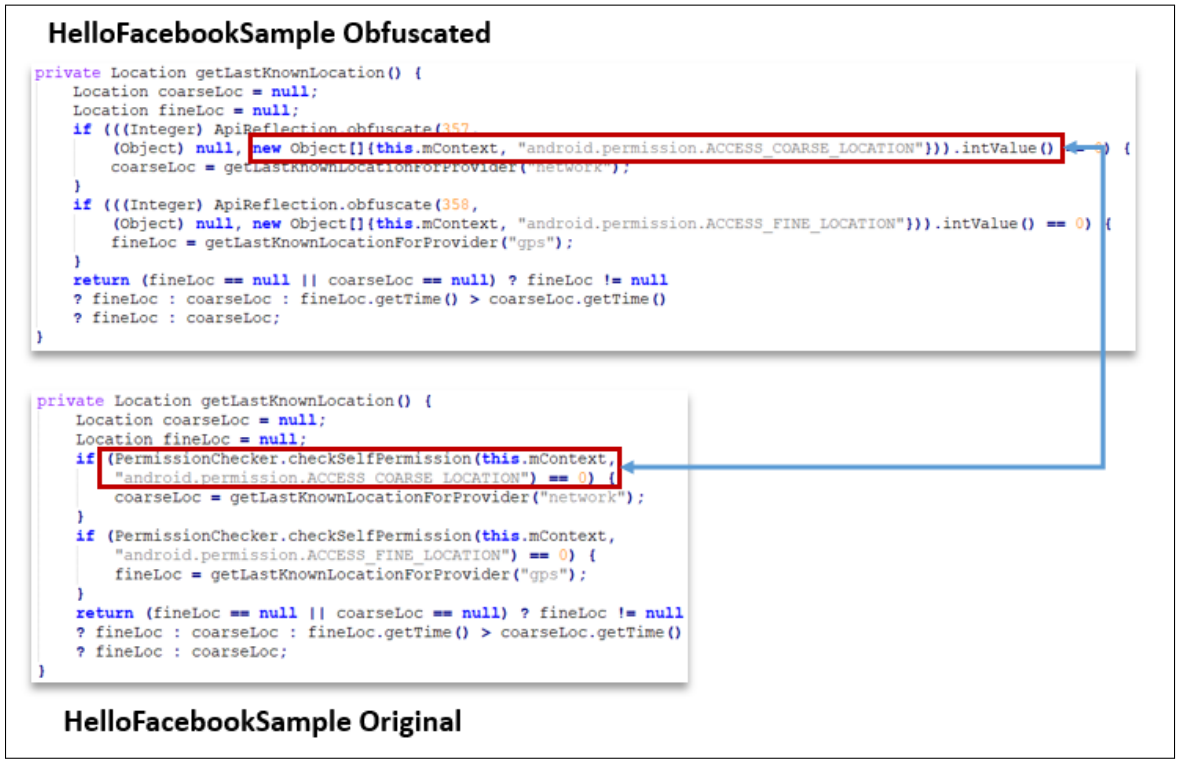

Figure 6.3: Obfuscapk Reflection Impact on Maintained String Computations

AAMO's reflection results on lines of active code and reachability are shown in 6.5. Significantly less methods are not reachable when compared with Obfuscapk. This is due to AAMO's focus on only statically called methods as reflection candidates; there are less targets in total where reflection may be introduced. AAMO differs drastically in the number of new methods introduced. Whereas Obfuscapk introduced only two new methods for each app, AAMO introduces hundreds of new methods. This is due to its implementation

approach. Each new reflection call is its own method as demonstrated in Figure 6.4. Each reflection call wraps a *single* call from the original app and adds it to a new, uniquely generated method name. All the newly introduced methods are in the same Java class as the location of the single, original call. The implementation approaches of both AAMO and Obfuscapk are important because of their differences. Despite both using reflection obfuscation, the changes to the structure or the underlying code are quite different though resulting in the same effect.

An interesting side note is that AAMO also introduces a small amount of indirection as part of the reflection obfuscator. Results were run multiple times to verify this was the case. This indirection will introduce new methods that simply redirect calls in the code but these new methods do not include reflection. It is not clear why AAMO does this and it may be due to reuse of the indirection code base and then focusing on those newly introduced methods to find static methods where reflection may be introduced.

Table 6.5: AAMO Reflection Impact on Lines of Active Code and Methods

	HFS	**MSS**	**RPS**	**SUS**
Lines Active Code in Origina l	127,988	83,584	185,487	121,243
Lines Active Code in Obf	130,077	81,748	186,861	121,951
# Methods in Original	8,827	5,783	12,788	8,112
# Methods in Obf	9,352	6,009	13,401	8,398
# Original Method Not Reachable	29	149	228	71
# Methods in Common	8,798	5,634	12,560	8,041
# Methods in Common Changed	143	111	188	86
# New Methods in Obf	554	375	841	357

```
public static float OkjviapHwECfDrZ(ResultPoint resultPoint) {
    return ((Float) ResultPoint.class.getMethod("getY",
    new Class[0]).invoke(resultPoint, new Object[0])).floatValue();
}
```

Figure 6.4: AAMO Example New Reflection Method

AAMO's reflection introduction impacts the string computations extracted from the apps, shown in Table 6.6. The reflection introduced by AAMO is less than Obfuscapk as

Table 6.6: AAMO Reflection Impact on String Computations

Reflection	**HFS**	**MSS**	**RPS**	**SUS**
String Computations in Original	2,870	1,889	3,083	2,588
String Computations in Obf	3,025	1,925	3,394	2,592
# String Computations Original Removed	1	78	35	51
# String Computations New Methods	144	114	311	49
# String Computations Orig Changed Methods	46	43	110	31
# String Computations Obf Changed Methods	58	43	145	37

it only reflects on statically defined methods. New string computations are introduced in the new method calls. The last two rows list the number of string computations extracted from methods in common. The numbers are close as AAMO works similarly to Obfuscapk and simply reflects on a single operation - which leads to no breakage in the existing string computation graphs. However, some of the string computations will become interprocedural in nature as they now point to the newly introduced methods and, as with Obfuscapk, in a few cases string literals will be viewed as new computations though the reflective call will unify them with the object passed.

Analysis of Advanced Reflection Obfuscator

The analysis includes a review of Obfuscapk's advanced reflection obfuscator. As noted earlier, this obfuscator focuses on a subset of API calls to perform reflection upon in an effort to fool malware detection tools. AAMO has no equivalent obfuscator for this. The analysis of advanced reflection's impact on the lines of active code and reachable methods is provided in Table 6.7.

Of interest are the number of methods in the original app and the obfuscated as well as the number of original methods not reachable. There are only two new methods added and no methods are removed from reachability. In the prior examples, introduction of reflection specifically removed methods from reachability - why is that not the case here? First, remember that advanced reflection only operates on methods in the Android package. These libraries are typically found on the Android device itself and not packaged with the app.

Table 6.7: Obfuscapk Advanced Reflection Impact on Lines of Active Code and Methods

	HFS	**MSS**	**RPS**	**SUS**
Lines Active Code in Original	127,988	83,584	185,487	121,243
Lines Active Code in Obf	128,414	83,960	186,081	121,673
# Methods in Original	8,827	5,783	12,788	8,112
# Methods in Obf	8,829	5,785	12,790	8,131
# Original Methods Not Reachable	0	0	0	0
# Methods in Common	8,827	5,783	12,788	8,112
# Methods in Common Changed	1,490	1,080	2,282	1,458
# New Methods in Obf	2	2	2	2

Table 6.8: Obfuscapk Advanced Reflection Impact on String Analysis, String Literals, and RevealDroid

Reflection	**HFS**	**MSS**	**RPS**	**SUS**
String Computations in Original	2,870	1,889	3,083	2,588
String Computations in Obf	2,885	1,901	3,098	2,603
# String Computations Original Removed	0	0	0	0
# String Computations New Methods	15	12	15	15
# String Computations Orig Changed Methods	820	641	879	775
# String Computations Obf Changed Methods	820	641	879	775

Second, Soot will only analyze class files found within the app itself. It therefore has no knowledge of the Android class files - whether it deals with the original or obfuscated app. The two, newly introduced methods are the same proxy methods described in Obfuscapk's reflection obfuscator. This result also proves that advanced reflection works as advertised; apply reflection only to classes in the Android package structure.

The analysis next reviews the string computations introduced by advanced reflection, shown in Table 6.8. As there are no methods removed due to reachability, none of the string computations in the original app are removed from its obfuscated form. Few new string computations are added, under 15 for each application. This is due to the nature of advanced reflection focusing a small subset of Android API calls that perform sensitive operations. As these applications are example apps developed by Facebook, they contain some of these sensitive calls but not nearly as many in an arbitrary app downloaded from an app provider like Google Play.

Analysis of Goto Obfuscator

Both Obfuscapk and AAMO provide a goto obufscator. Documentation for Obfuscapk and a review of its source code base indicate that it will apply obfuscation as follows [35]:

- Inserts a goto instruction pointing to the end of the method,
- Inserts another goto pointing to the instruction after the first goto, and
- Modifies the control-flow graph by adding two new nodes.

Interestingly, this is *exactly* the same implementation for goto obfuscation implemented by AAMO [84]. Here, in contrast to reflection, the two obfuscation tools have chosen the same approach to implement an obfuscator. The result is that both approaches have identical results with respect to method reachability, demonstrated in Tables 6.9 and 6.10.

Table 6.9: Obfuscapk GoTo Impact on Lines of Active Code and Methods

AAMO	**HFS**	**MSS**	**RPS**	**SUS**
Lines Active Code in Original	127,988	83,584	185,487	121,243
Lines Active Code in Obf	136,631	89,258	198,128	129,177
# Methods in Original	8,827	5,783	12,788	8,112
# Methods in Obf	8,827	5,783	12,788	8,112
# Original Methods Not Reachable	0	0	0	0
# Methods in Common	8827	5,783	12,788	8,112
# Methods in Common Changed	8,460	5,489	12,301	7,747
# New Methods in Obf	0	0	0	0

Table 6.10: AAMO GoTo Impact on Lines of Active Code and Methods

AAMO	**HFS**	**MSS**	**RPS**	**SUS**
Lines Active Code in Original	127,988	83,584	185,487	121,243
Lines Active Code in Obf	136,631	89,258	198,128	129,177
# Methods in Original	8,827	5,783	12,788	8,112
# Methods in Obf	8,827	5,783	12,788	8,112
# Original Methods Not Reachable	0	0	0	0
# Methods in Common	8827	5,783	12,788	8,112
# Methods in Common Changed	8,460	5,489	12,301	7,747
# New Methods in Obf	0	0	0	0

Table 6.11: Goto Impact on String Computations

	HFS	**MSS**	**RPS**	**SUS**
String Computations in Original	2,870	1,889	3,083	2,588
String Computations in Obf	2,870	1,889	3,083	2,588
# String Computations Original Removed	0	0	0	0
# String Computations New Methods	0	0	0	0
# String Computations Orig Changed Methods	2,627	1,727	2,793	2,314
# String Computations Obf Changed Methods	2,627	1,727	2,793	2,314

Goto impact on string computations is minimal. For brevity, its impact for both Obfuscapk and AAMO is provided in a single Table, 6.11, as the results for both tools are identical. As goto obfuscation is injected at the very beginning and at the very end of the method, there is no opportunity to break or remove string computations. They remain perfectly in tact. There appears to be little value in the introduction of goto obfuscation.

This result also brings up a compelling question - why is goto introduced to Android apps so ineffective? In C, the use of an unconditional goto can introduce problems in following the control flow of an application. Application of an unconditional goto should create the same issues in Java. There are two reasons why this is not the case, one related to the Java programming specification itself and the other related to Dalvik bytecode and its implementation of goto.

The Java specification reserves the `goto` keyword but does not implement it[38]. It only offers a conditional goto in the form of `break` statements. Developers typically use the default case for the `break` and do not specify a label for a redirect despite break statements being developed to allow conditional gotos that redirect. If obfuscation is applied at the level of Java source code level, then unconditional goto is simply not feasible.

However, most obfuscation tools work at the Dalvik bytecode level. The use of goto in Dalvik bytecode is allowed and often used. Nothing would prohibit an obfuscation developer from introducing a goto statement in the bytecode itself. In fact, this is exactly what Obfuscapk and AAMO do to implement goto. We provide in Figure 6.5 an example of the Dalvik bytecode for an Obfuscapk goto obfuscated method. We see a goto statement

```
1        public boolean onHoverEvent(android.view.MotionEvent
            r8) {
2              r7 = this;
3              goto L_0x0067
4           L_0x0003:
5           ...
6           L_0x0067:
7              goto L_0x0003
```

Figure 6.5: Goto Implementation in Bytecode

at top in line 3, the statement points to another goto at bottom, line 7, with that goto pointing to a code point at top. After this jumping, the code execution begins as would be normal for the method. Note, an infinite loop is avoided as the return for the method avoids calling the goto at end once again. The example shows that obfuscation developers can implement a goto. When Soot performs its backward flow analysis of the app, it finds these goto statements and outputs them as Soot units as they are, in actuality, legitimate Java conditional gotos.

Goto obfuscation can be implemented at the bytecode level that is purely unconditional, though significant technical challenges exist for obfuscation tool developers. [21] identifies that, though unconditional goto obfuscation is possible in Dalvik bytecode, there is a problem described as the register-type conflict problem. Specifically, "A complex control-flow obfuscation on Dalvik bytecode may split method instructions into different code segments, relocate and link them. Such actions will generally cause the verifier to report a register-type conflict[21]." This presents a problem for obfuscation developer as they need to understand how the Dalvik verifier works - for which there is no documentation. [21] implemented a solution to introduce unconditional goto. Their results were preliminary, appear successful, but the source code for their obfuscation tool was not released.

Obfuscapk does implement the use of unconditional goto in its reorder obfuscator. We ran the obfuscator on the four example apps and in all cases the APK it output had two issues. First, the output APK could not be disassembled by Soot. Each time, Soot threw the same exception, shown in Figure 6.6. Note the stack trace and the call on line 4. This is

```
1  java.lang.NullPointerException
2        at soot.DEXpler.DEXType.toSoot(DEXType.java:130)
3        at soot.DEXpler.DEXType.toSoot(DEXType.java:105)
4        at
            soot.DEXpler.DEXBody$RegDbgEntry.<init>(DEXBody.java:179)

5        at soot.DEXpler.DEXBody.<init>(DEXBody.java:308)
6        at soot.DEXpler.DEXMethod$1.getBody(DEXMethod.java:119)
7        at
            soot.SootMethod.retrieveActiveBody(SootMethod.java:402)
```

Figure 6.6: Soot Exception Upon Introduction of Unconditional Goto

```
1  private void calculatePageOffsets(...) {
2            r11 = this;
3            goto L_0x03da
4        L_0x0004:
5            float r6 = r6 + r4
6            goto L_0x014d
7        L_0x0009:
8            int r4 = r4 + 1
9            goto L_0x02c8
10       L_0x000f:
11           int r7 = r7 + 1
12           goto L_0x02f1
```

Figure 6.7: Unrecoverable Java Source Code

in-line with [21]'s assessment that brute force introduction of unconditional goto will lead to registry conflict error. It is impossible to say if what Obfuscapk generated as an APK would run on the Android VM. Also, it is impossible to retrieve the original Java source code when unconditional gotos are introduced into the code. We ran the reorder obfuscated apps through JADX to decompile into original Java source and where unconditional gotos were added into the bytecode, original source code could not be rendered as there is no equivalency with the Java specification. Figure 6.7 shows an example of Java source code that cannot be successfully rendered. The introduction of multiple unconditional gotos has made it impossible to generate Java source.

Analysis of Indirection Obfuscator

Both Obfuscapk and AAMO provide an indirection obufscator. Documentation for Obfuscapk and a review of its source code base indicate that it will apply obfuscation to modify the control-flow graph as follows with the express goal of not impacting code semantics [35]:

This technique modifies the control-flow graph without impacting the code semantics: it adds new methods that invoke the original ones. For example, an invocation to the method m1 will be substituted by a new wrapper method m2, that, when invoked, it calls the original method m1.

AAMO[84] takes a similar approach for indirection and indicates it functions as follows:

This obfuscator aims to evade signatures based on the application's call graph. In practice, we redirect each method call to proxy methods that call the original method. These proxies share the same prototype of the original method, including parameters order and type, return type, invocation type, and registers. Return values, if any, are returned by the proxy methods. Each proxy method is a public static method.

The analysis begins with an examination of Obfuscapk's indirection obfuscator on method reachability with results provided in Table 6.12. The number of methods in the obfuscated application sometimes differ dramatically from the original, with one including twenty five thousand methods in total - nearly five times the number of methods in the original. Other applications have only a few hundred more methods than the original. It demonstrates that Obfuscapk's implementation of indirection is opportunistic and will maximize its introduction into an app when possible. The results also show that no methods from the original application are lost due to reachability. Again, this is an expected result as indirection essentially adds proxy methods for the original methods. Additionally, nearly all the original methods change. Figure 6.8 provides an example of an indirection introduction by Obfuscapk that proxies a method call to a logging function. The indirec-

tion added by Obfuscapk, as well as AAMO, are nearly all single method propagations as shown in the figure.

Table 6.12: Obfuscapk Indirection Impact on Lines of Active Code and Methods

	HFS	**MSS**	**RPS**	**SUS**
Lines Active Code in Original	127,988	83,584	185,487	121,243
Lines Active Code in Obf	138,772	179,500	199,267	131,167
# Methods in Original	8,827	5,783	12,788	8,112
# Methods in Obf	9,421	25,035	13,102	8,628
# Original Methods Not Reachable	0	0	0	0
# Methods in Common	8,827	5,783	12,788	8,112
# Methods in Common Changed	8,460	5,489	12,301	7,747
# New Methods in Obf	594	19,252	314	516

```
public static int FhvsbxCMVZtOdMbT(String str, String str2) {
    return Log.i(str, str2);
}
```

Figure 6.8: AAMO Example New Reflection Method

Examination of the string computations is provided in Table 6.13. The results show that new string computations are introduced as a result of indirection but the number of string computations in the original methods modified by indirection are identical. Obfuscapk's definition indicates that proxy methods are introduced and this impacts string computations in two ways. First, it sometimes changes the type of vertexes in a string computation with calls that were formerly intraprocedural switching to interprocedural as it must call the indirection method. Second, there is an edge case that also splits the string computations. Both these issues will be problematic for machine learning classification - discussed later in Section 6.1.3 along with an approach to correct the problem.

We next examine AAMO's indirection obfuscator with method reachability results provided in Table 6.14. It shows that AAMO introduces much more indirection into the apps, adding tens of thousands of new methods and losing no original method reachability.

Lastly, we examine AAMO's indirection impact on string computations, with results provided in Table 6.15. We see that it generates string computations in much the same way

Table 6.13: Obfuscapk Indirection Impact on String Computations

	HFS	**MSS**	**RPS**	**SUS**
String Computations in Original	2,870	1,889	3,083	2,588
String Computations in Obf	3,170	5,479	3,097	2,782
# String Computations Original Removed	0	0	0	0
# String Computations New Methods	167	3,339	25	97
# String Computations Orig Changed Methods	2,627	1,727	2,793	2,314
# String Computations Obf Changed Methods	3,003	2,140	3,072	2,685

Table 6.14: AAMO Indirection Impact on Lines of Active Code and Methods

AAMO	**HFS**	**MSS**	**RPS**	**SUS**
Lines Active Code in Original	127,988	83,584	185,487	121,243
Lines Active Code in Obf	201,114	199,543	244,962	195,167
# Methods in Original	8,827	5,783	12,788	8,112
# Methods in Obf	25,862	30,370	24,936	25,582
# Original Methods Not Reachable	0	0	0	0
# Methods in Common	8,827	5,783	12,788	8,112
# Methods in Common Changed	8,460	5,489	12,301	7,747
# New Methods in Obf	17,035	24,587	12,148	17,470

that Obfuscapk does. Also, some of its generated string computations have the same two characteristics as described for Obfuscapk indirection string computations.

Table 6.15: AAMO's Indirection Impact on String Computations

Indirection	**HFS**	**MSS**	**RPS**	**SUS**
String Computations in Original	2,870	1,889	3,083	2,588
String Computations in Obf	5,343	5,426	4,265	5,078
# String Computations Original Removed	0	0	0	0
# String Computations New Methods	2,685	3,285	1,465	2,757
# String Computations Orig Changed Methods	2,627	1,727	2,793	2,314
# String Computations Obf Changed Methods	2,658	2,141	2,800	2,321

6.1.3 Semantic Graph Equivalence

Obfuscation impacts string computations in three ways. These include:

- String computations are simply removed due to reachability of their parent method being removed via reflection obfuscation,

- Intraprocedural string computations may maintain the same isomorphic form between the original and obfuscated form by the types of vertices may be changed via indirection obfuscation, and

- Intraprocedural string computations may be broken apart via indirection obfuscation.

The first impact is difficult to remediate. It would require an analysis of string computations, identification of those that invoke Java reflection, identification if the string literals for the methods to be called are present, and then propagate these references to a modified version of Soot that treats the methods as reachable.

The latter two impact items can be addressed relatively easily. We review both cases in Figure 6.9. The upper portion shows the original code from the *HelloFacebookSample*. We focus on the string created for the `IlleagleArgumentException`. It shows four string parts combined with the Java + operator. The string analysis algorithm will create a four vertex string computation - one for each part. Underneath, Java translates the + operator into a `StringBuilder` object and performs four appends.

The lower portion of the figure shows the obfuscated form of the same exception string after application of Obfuscapk indirection. We see multiple implications for the string computations. First, a string computation is created that is four vertices in size - similar to the string computation in the original code. However, each vertex represents an interprocedural call, the Obfuscapk indirection proxy method, as opposed to a `StringBuilder append`. So, the isomorphic form is the same, but the type of each vertex changes.

Second, four new string computations are generated. Each argument passed to the proxy method is itself a string computation and not associated with the `StringBuilder` object also passed, though they are connected in the proxy method. Note, both Obfuscapk and AAMO chose to pass string literals to the proxy method as opposed to adding them in directly in the proxy method. Had the authors of these obfuscations chosen this approach we would have only two new string computations.

HelloFacebookSample Original

```
} else if (!bounds.contains(dodgeRect)) {
    throw new IllegalArgumentException("Rect should be within the child's bounds. Rect:"
    + dodgeRect.toShortString() + " | Bounds:" + bounds.toShortString());
}
```

HelloFacebookSample Obfuscated

```
} else if (!ZcSeEVmvtoGfsUz(pePkvJbXQLMNmhB, AhwZxspbYeELPyC)) {
    StringBuilder sb = new StringBuilder();
    GdIcfKsaVSUMoXb(sb, "Rect should be within the child's bounds. Rect:");
    NFVBtyQxIEbWskp(sb, glCPehUFMfiLGpW(AhwZxspbYeELPyC));
    LYpSIENaklzRWxm(sb, " | Bounds:");
    mJqdauIzWkLnRCM(sb, kdEohLQPbVNigBl(pePkvJbXQLMNmhB));
    throw new IllegalArgumentException(ihsLegEHAVfCrkR(sb));
}
```

Figure 6.9: Modifications Performed on String Computations by Obfuscation

We can correct both these issues using the persisted string computation graphs. At first, we attempted to correct the issues using only SPARQL queries and its update operator to correct links in the string computations. This proved too difficult and a separate algorithm was created to correct these issues and create a string computation isomorphically identical to that found in the original. We present the logic of the algorithm in Figure 6.10. The algorithm accepts a graph model and finds all string computations that are candidates for graph correction. A key aspect is that the candidate string computations have interprocedural calls to other string computations that are in the same Java class. Other obfuscators outside of Obfuscapk and AAMO could implement an indirection obfuscator that moves proxy methods to classes outside the originator class - requiring more examination to determine if a semantic correction can legitimately be applied. The algorithm reconstitutes the string computation found in the original app. However, it does not attempt to remove the newly introduced string computations. This task could be implemented as well. Application of the algorithm as a post-processing step was able to semantically corrected hundreds of string computations in each of the four apps obfuscated by both Obfuscapk and AAMO.

The semantic graph equivalence algorithm demonstrates the resilience of string com-

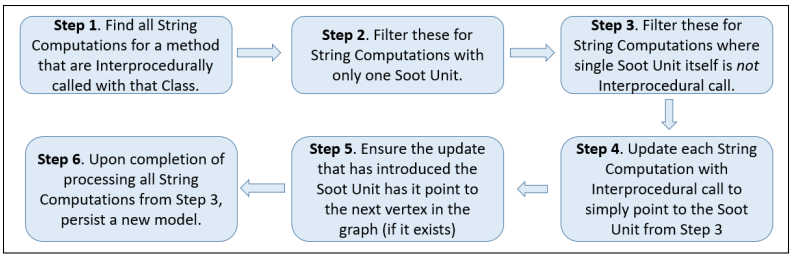

Figure 6.10: Semantic Remediation of String Computations Modified by Obfuscation

putations as a feature undeterred by obfuscation. With an investigation of the obfuscators functionality, we were able to create a simple algorithm to modify graph computations for isomorphic and semantic equivalence across the original and obfuscated form.

Chapter 7

Obfuscation Impacts Classification Features

7.1 Features

The analysis performed in Section 6.1 examined how obfuscation applied by two state-of-the-art obfuscators changed the control flow of applications and that change's impact on generated string computations. This provides an initial perspective as to how our feature sets will change as a result of the application of different obfuscators. Our goal in this chapter is to provide *explainability* for classification results in terms of changes to the generated feature sets resulting from application of the obfuscators reflection, goto, and indirection. The analysis advances the current state of the art for determination of whether a malware detection tool is obfuscation resilient. The typical approach taken by researchers [53, 110] who develop Android malware detection is: (1.) Apply obfuscation to the set of apps examined, (2.) Rerun the classification algorithm used, and (3.) Explain results in terms of accuracy alone as opposed to accuracy and a detailed assessment of the impact the obfuscator had on the feature set generated, especially those features most important for classification. We focus on the explainability of machine learning in terms of impact to the

features most critical to classification. We advance the current state-of-the-art by introduction of metrics for importance feature retention and removal rates for those features most critical to classification. These metrics present a new and important mechanism for evaluation of obfuscation resilience that both explain dips in accuracy when specific obfuscators are applied as well as provide indicators as to whether malware developers could easily thwart malware detection algorithms with further customizations to obfuscators. That is, could an obfuscator like indirection be changed to significantly drop the accuracy of a genre of malware detection algorithms without impact to the running time or system resources used by the obfuscated app.

Our research into classification of the string computations led us to use gradient boost (GB) as it consistently performed better than SVM of k-NN clustering algorithms for classification of our malware and benign sets. GB involves creating a number of hypotheses h_t. and combining them to form a more accurate composite hypothesis [92]. When supplied an input feature of *x* the boosted classifier will output the additive form:

$$H(x) = \sum_t \alpha_t h_t(x).$$

Here, α_t represents the weight to be given to each decision tree or hypothesis, h_t. An example of the decision trees are shown in Figure 7.1 and provides a simplified example of decision trees of depth three. Each non-leaf node represents a feature and some decision related to it. Here, the decisions are how often the features are present in the input between a set of ranges. For example, the root node may represent the decision "Is feature *n* greater than zero." If the answer is yes, the appropriate path is taken in the decision tree and leads to the next decision "Is feature *m* between 10 and 50." The GB algorithm evaluates 1000s of potential trees with the output model utilizing 100's of best performing decision trees. The typical depth of a tree is 10 and the number of leaf nodes per tree between 8 to 32 nodes.

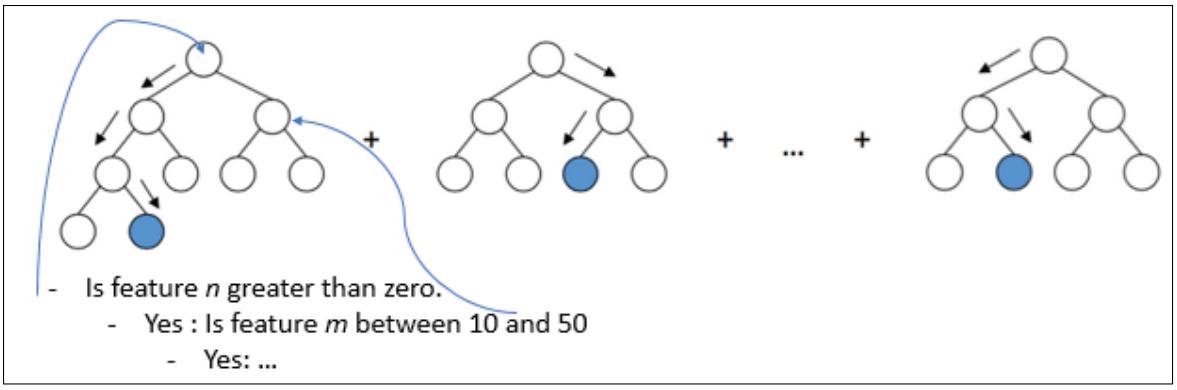

Figure 7.1: Examples of GB Decision Trees

7.1.1 Feature Importance

Our analysis was performed on the University of Buffalo's Center for Computation Research (CCR). This resource provides the ability to run up to 1,000 concurrent processes. We developed scripts to test the different obfuscators and for each obfuscator different sets of hyperparameters. The hyperparameters evaluated included number of trees, tree depth, and learning rate. Learning rate controls the weighting of new trees added to the model [25] and requires particular care as selection of learning rates at high values may yield high accuracy but an overtrained model. In general, learning rates higher than 0.1 have the potential for overtrained models and we utilized learning rates 0.1 or lower. We ran 100s of scripts on CCR that ran GB with different hyperparameter configurations and reports results in terms of the hyperparameters that offered the highest accuracy.

A common complaint about ML models is that they are black boxes - excellent at the assigned classification tasks they are developed for but sub-par at explaining *why* it classified. A recent push in ML application across all sciences is to include an explainability assessment for results[88]. GB offers build in explainability for its generated models. It provides an importance metric the ranks individual features that have the most influence on a models decision trees, providing this ranking from most influential to least influential. An important feature is typically present across multiple trees and its impact on classifying an arbitrary app as malware or benign can be specified as a percentage in relation to all

the other features. Specifically, importance is calculated for a single decision tree by the amount that each attribute split point improves the performance measure, weighted by the number of observations the node is responsible for [25].

We examined the GB models with the best accuracy for classification of malware and benign apps for both the string computations and RevealDroid features sets. We identified the top 16 important features based on weight in terms of number of correct observations the feature is responsible for. This provides us with the top 16 most important features output by our string computations and RevealDroid. The models were trained and tested on the same set of apps to guarantee commonality for later analysis between string computations and RevealDroid.

Table 7.1 provides the top 16 most important string computations, ranked in descending order, for the GB model that best classified the un-obfuscated apps described in Section 5.2. A description of the string constructed by the string computation is provided. We see that the string computation types used most often for successful classification span a wide spectrum and tell the story of how malware utilizes strings under the hood across different malware families. These include string computations weighted toward collection and storage of sensitive information. For example, features 1, 4, 5, 8, 13, 14, and 16 all involve `StringBuilder` operations that append content to strings. This content is written to log files, or Json objects, or to harvest information from an HTML response. We also see string computations focused on the Android device itself. For example, accessing the `PackageManager` to find out package info, obtaining phone information using the `TelephonyManager` class, or identifying the CPU based on OS build.

Note, the size of the string computations themselves can vary. We provide two examples in Figure 7.2. To the left we see a larger string computation, linked to feature 13, that writes content to a logging file. To the right is a much smaller string computation that accesses the `PackageManager` to find information about an installed package.

We next examined the top 16 most important RevealDroid features used by its GB

Table 7.1: Top 16 Most Important String Computation Features Summarized

Feature	Description
1	Sequence of multiple appends to StringBuilder
2	String used to obtain Package Manager and identify installed apps
3	String to identify CPU based on OS build
4	Sequence of appends to StringBuilder for JSON object
5	StringBuilder operations adding multiple char objects to String
6	Examination of String to identify if CharSequence is present in String
7	String that helps to determine the external storage state
8	Sequence of appends to StringBuilder byte
9	Use of TelephonyManager to obtain phone information
10	String object created via initialization utilizing byte[] array
11	String object created via initialization utilizing a char[] array
12	String object that identifies system bootloader
13	Sequence of appends to StringBuilder supplied to logging util
14	StringBuilder operations involving Jsoup library – used to parse HTML
15	String .equals() operations
16	Sequence of appends to StringBuilder focused on int values and booleans

model to most successfully classify malware and benign applications, provided in Table 7.2. We see that these features tend to deal with sensitive operations - for example identifying the location of a phone, accessing network or Bluetooth information, or accessing the phone's file system to write or read information.

RevealDroid importance features for our developed model fall into two categories. The basic form for each category is shown in Figure 7.3. To the left is the token `android.net.wifi`. This feature is a pure count of the following. For each class in that namespace, shown toward center, each API call for the class is counted for a app. For example, if the app uses the `ScanResult` class and calls the `getWifiStandard()` method two times and `isPasspointNetwork()` three times, it will have a count of two - indicating that two methods or fields from the class are used. RevealDroid also performs a basic form of information flow analysis where it attempts to identify sinks and sources. This is what is embodied to the right by the token `BLUETOOTH_INFORMATION`. RevealDroid maintains a listing of API calls that are likely source or sink points for for exfiltration of sensitive information. A subset of these API calls are shown to the right. If an app exhibits one of these

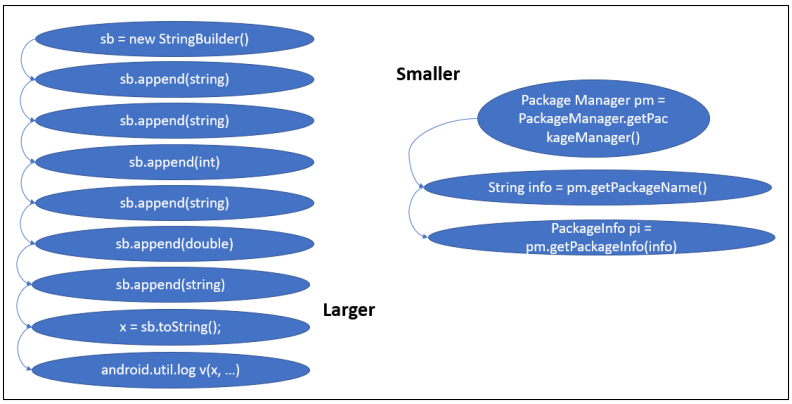

Figure 7.2: Examples of String Computation Graphs for Importance Features

calls - it is presumed to have a source in this case. RevealDroid collects what it believes are potential sinks and counts these. We see the importance features for RevealDroid are a mixture of the pure API calls and potential sources and sinks of critical information.

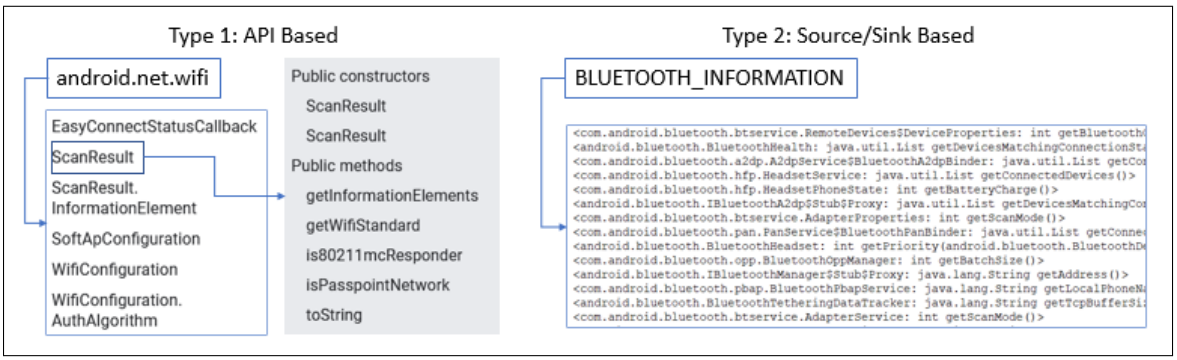

Figure 7.3: Examples of RevealDroid Importance Features

7.1.2 Accuracy of String Computation and RevealDroid when Obfuscated

We next took the unobfuscated apps and obfuscated them using reflection, goto, and indirection for both Obfuscapk and AAMO. In addition, we ran Obfuscapk's advanced reflection obfuscator. We then classified the obfuscated apps using two approaches. Both have been used in previously published research and serve as baseline metrics with which to measure how obfuscation impacts models trained on unobfuscated apps [53, 110]. The

Table 7.2: Top 16 Most Important RevealDroid Features

Feature	Description
1	android.view
2	android.location
3	android.net.wifi
4	LOG
5	android.content
6	NETWORK
7	android.view.inputmethod
8	android.bluetooth
9	android.opengl
10	FILE
11	AUDIO
12	NETWORK_INFORMATION
13	android.support.v4.net
14	BLUETOOTH_INFORMATION
15	android.support.v7.text
16	android.speech.tts

methodologies are:

- **Train on Original, Test on Transformed:** Trained classifiers on a dataset that contains the original apps and then tested those classifiers on the obfuscated versions of the same apps.
- **Train on Original, Test on Unseen Transformed:** Refrain from training classifiers on any apps that we transformed (i.e., original, malicious apps before obfuscation); however, we test on the transformed, obfuscated apps.

The intuition is that the first case is easy and the second case hard. If the classification rates for the first case significantly drop when tested on apps the model was trained on - then clearly the feature set used is not obfuscation resilient. The second case is significantly harder. How will the model perform when it needs to classify apps it has never seen that have been obfuscated? This is the true measure of obfuscation resilience. Table 7.3 reports results for both methodologies. The first row of results is the accuracy of the best model for the *unobfuscated* apps. The next two sets of rows test each methodology fore each

individual obfuscator, that is for reflection, goto, and indirection. We report results in terms of accuracy and provide the train and test sizes. Note - it was difficult to get exactly the same test and train sizes across each type of obfuscator as AAMO and Obfuscapk failed on certain applications but the test and train sets are relatively close in size across all obfuscators.

Table 7.3: Accuracy of String Computations and RevealDroid on Different Obfuscators

Test Set	Obfuscation Type	Accuracy SC	Accuracy RV	Train Size	Test Size
	Un-Obfuscated	97.31%	94.11%	8,600	2,150
Orig	**AdvRef Obfuscapk**	96.70%	93.48%	8,340	2,085
	Ref Obfuscapk	96.11%	93.89%	8,542	2,136
	Ref AAMO	96.62%	94.11%	8,221	2,055
	Goto Obfuscapk	96.02%	93.71%	8,601	2,150
	Goto AAMO	96.79%	94.01%	8,477	2,119
	Indir Obfuscapk	95.90%	93.92%	8,312	2,078
	Indir AAMO	95.28%	93.44%	8,437	2,109
New	**AdvRef Obfuscapk**	95.80%	89.20%	8,340	2,085
	Ref Obfuscapk	96.25%	92.90%	8,542	2,136
	Ref AAMO	95.45%	93.14%	8,221	2,055
	Goto Obfuscapk	96.10%	93.89%	8,601	2,150
	Goto AAMO	95.98%	93.45%	8,477	2,119
	Indir Obfuscapk	92.12%	91.40%	8,312	2,078
	Indir AAMO	94.98%	94.82%	8,437	2,109

The results demonstrate that, as hoped, accuracy remains close to the original for the first methodology. We see a slight drop off for both the string computations and RevealDroid but nothing that exceeds greater than two percent. However, the results for the second methodology tell a different story. We see that for certain obfuscators, accuracy can dip as much as 5% for both tools when a higher bar for obfuscation resiliency is presented.

We leverage the importance features reported by GB to develop a novel metric that enables us to directly correlate this dip in accuracy to the most important features used for classification. Our methodology to develop this metric is as follows. We run the obfuscation tools with each obfuscator. We record the names of the apps that were in the test set, represented as test $test$. We then develop two functions, the first, 1 is the number of features

removed, $RM(x)$ where x represents one of the 16 importance features. For each feature we sum the number of times the features is present in the unobfuscated app subtracted for the times present in the obfuscated app. We then calculate a percent retained, 2 by dividing that total by the sum of the number of times the feature is present in the unobfuscated app.

$$RM(x) = \frac{\sum_{test}(origcount(x, test)) - \sum_{test}(obfcount(x, test))}{\sum_{test}(origcount(x, test))}.(1)$$

$$RT(x) = 1 - RM(x).(2)$$

In the following sections we examine each obfuscator and report results on the retention and removal rates for both string computations and RevealDroid. We report for each of the 16 importance features the % of the feature that was removed by obfuscation and the % that was retained in terms of the test of apps that was supplied for classification. These sections report a mixture of both anticipated and unanticipated results. Critically, the results demonstrate that a proclamation of obfuscation resilience requires a thorough examination of the obfuscation output to understand the true impact on the feature sets.

7.1.3 Advanced Reflection Analysis

Our analysis starts with an examination of the impact Obfuscapk's advanced reflection obfuscator has on the 16 importance features for both the string computations and RevealDroid. This obfuscator applies reflection but focuses on a subset of critical or sensitive Android API specific operations to obfuscate. Figure 7.4 shows the result on the malware set and Figure 7.5 shows the result on the benign set. The blue portion of each bar indicates the percent of that feature that was retained after obfuscation and the red portion indicates the percent that was lost. Of the 16 importance features only one is impacted - feature 9. This features creates a string that harvest device information utilizing the Android `TelephonyManager` class. We consulted the listing of method signatures Obfuscapk's

advanced reflection targets for reflection obfuscation which is available on its git site. This listing indicates that `TelephonyManager` and its method call `getDeviceId()` are a target for obfuscation. We see that occurrences of this feature are completely removed in the obfuscated malware sample and removed in nearly 60% of the obfuscated benign sample. The feature itself is heavily weighted toward being present in malware, with over 500 malware apps exhibiting the string computation feature while only 10 of the benign apps had it present. This example answers the critical question of how the obfuscation impacts the feature set and, importantly, those features most important for classification. It should be noted that this obfuscation did not significantly impact the ability to classify obfuscated apps using string computations as features.

We next review the impact of advanced reflection on the RevealDroid set of importance features. Figures 7.6 and 7.7 show the results for the malware and benign sets respectively. We see that three importance features are severely degraded by advanced reflection. This keeps in line with the classifier results where accuracy dipped to 89% for RevealDroid in the presence of advanced reflection. A review of the importance features explains why. Figure 7.8 depicts why the degradation occurs. The left shows names spaces whose classes advanced reflection targets via the obfuscation. The right shows the RevealDroid importance features most impacted by advanced reflection. The sets of API calls that RevealDroid focuses on are the exact calls that advanced reflection obfuscates. Obfuscapk's creators intended for this to work exactly this way - tricking malware detection tools by removing some of the calls to those classes' sensitive methods.

7.1.4 Reflection Analysis

Next we review the impact of reflection obfuscation as applied by both Obfuscapk and AAMO. Figures 7.9 and 7.10 show Obfuscapk's reflection impact on string computations while Figures 7.10 and 7.11 show AAMO's impact. Both obfuscation tools apply reflection liberally with no specific classes or methods targeted. AAMO has the limitation that

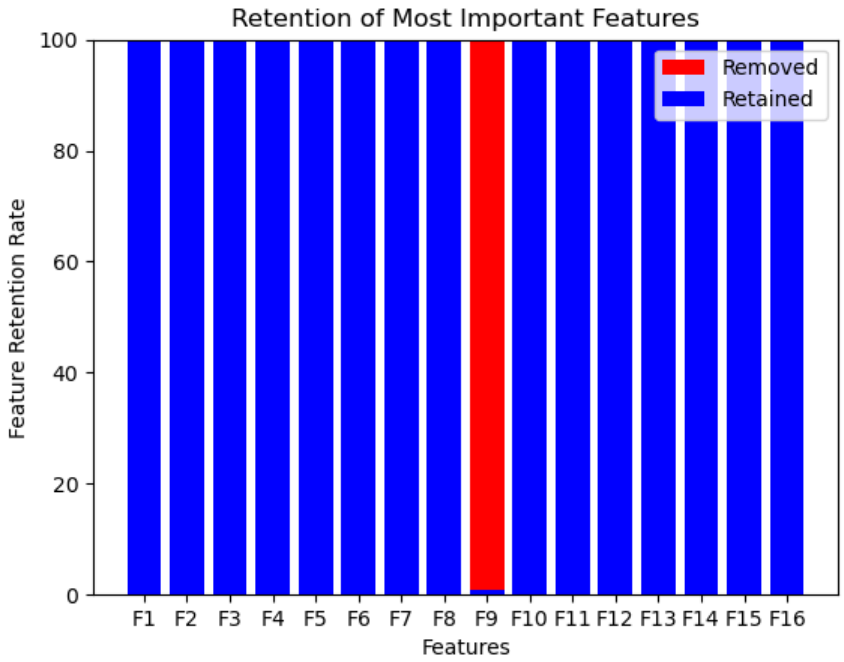

Figure 7.4: Obfuscapk Advanced Reflection Impact on String Computation Malware Importance Features

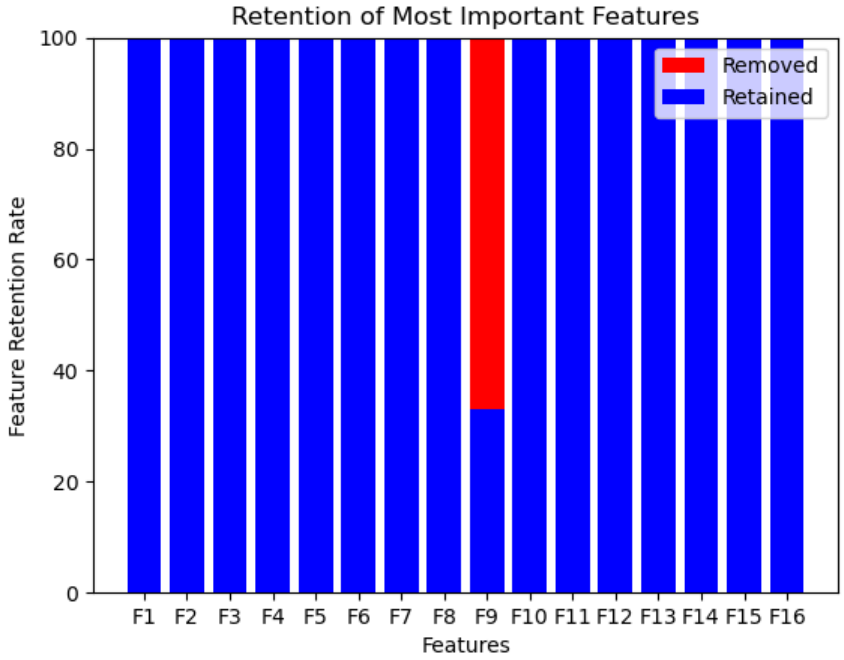

Figure 7.5: Obfuscapk Advanced Reflection Impact on String Computation Benign Importance Features

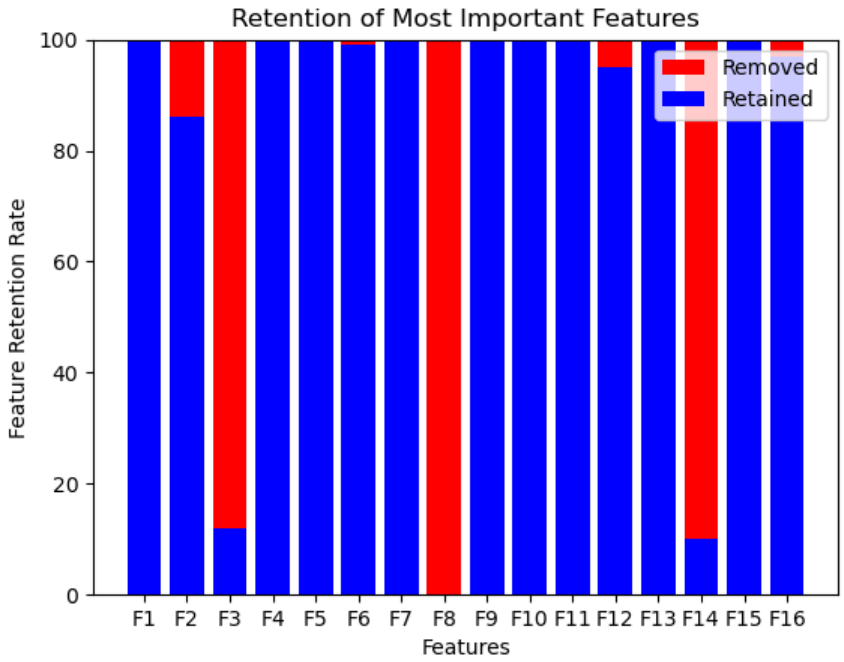

Figure 7.6: Obfuscapk Advanced Reflection Impact on RevealDroid Malware Importance Features

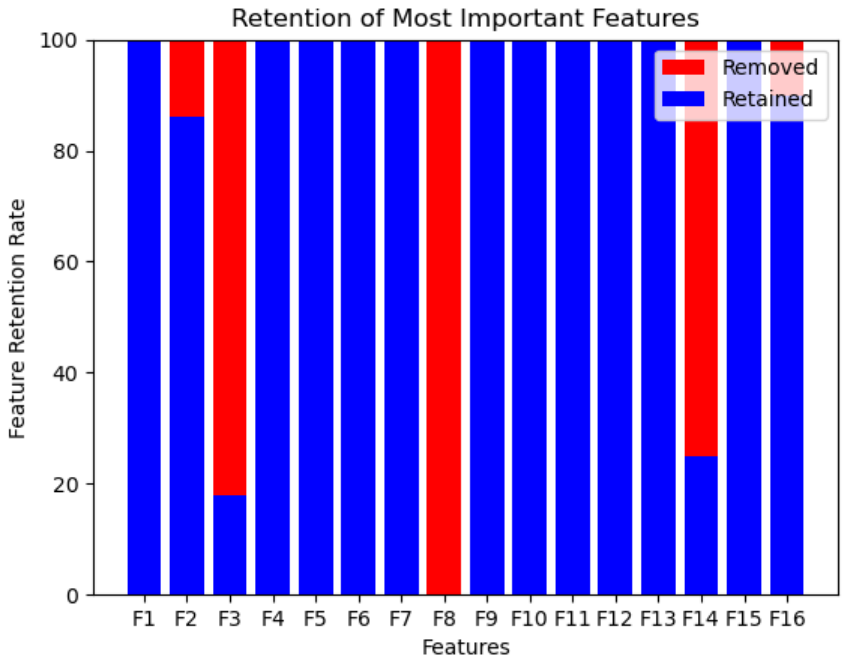

Figure 7.7: Obfuscapk Advanced Reflection Impact on RevealDroid Benign Importance Features

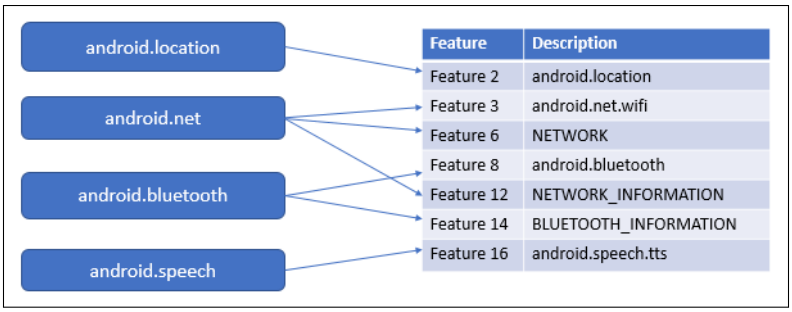

Figure 7.8: Degraded Reveal Droid Importance Features Mapped to Advanced Reflection Targets

it can only perform reflection upon statically defined methods. The result of reflection obfuscation is that some string computations are lost due to reachability issues. The features removed are from methods no longer reachable by the call graph and therefore removed from examination. This is the intent of reflection obfuscation; confuse the execution of the control flow graph to make the code less understandable. Also note that Obfuscapk reflection limits the amount of reflection introduced to ensure that apps are still runnable and fast with respect to usability. Impact on the classification rates is minimal - with at most a two percent reduction in overall accuracy for string computations.

We next turn to RevealDroid and examine how it performs in the presence of Obfuscapk and AAMO reflection. Figures 7.13 and 7.14 show performance when Obfuscapk is applied. In some cases reflection introduction has moderate impacts on some of the important RevealDroid features, for example features 1 and 5. Figures 7.15 and 7.16 show performance when AAMO is applied. RevealDroid features do better, with only slight degradation. All of these results are in line with what was initially expected; the obfuscators apply reflection liberally and are not targeting any one feature. The classification results therefore show only a slight dip in performance with neither string computation or RevealDroid showing more than a 2 percent dip in classification performance.

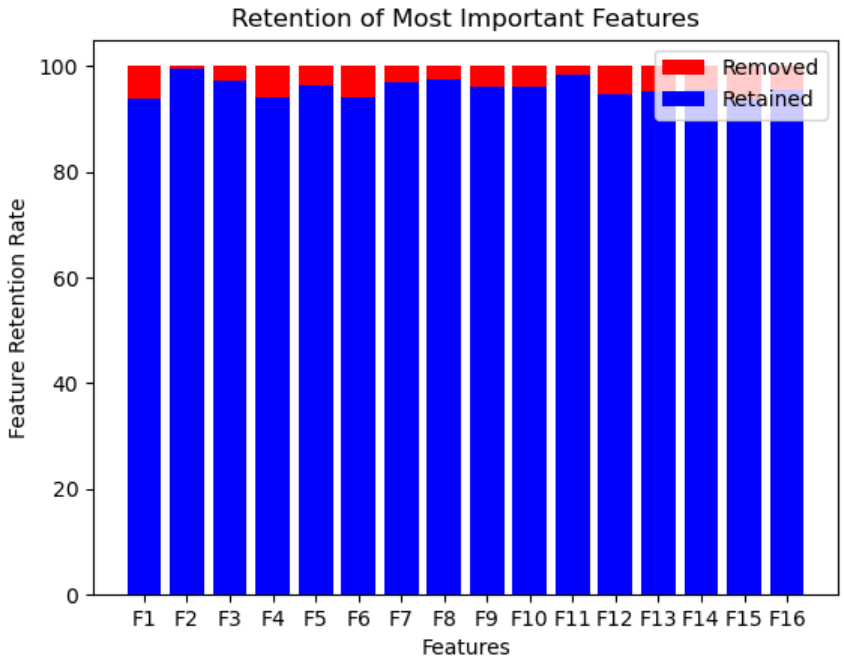

Figure 7.9: Obfuscapk Reflection Impact on String Computation Malware Importance Features

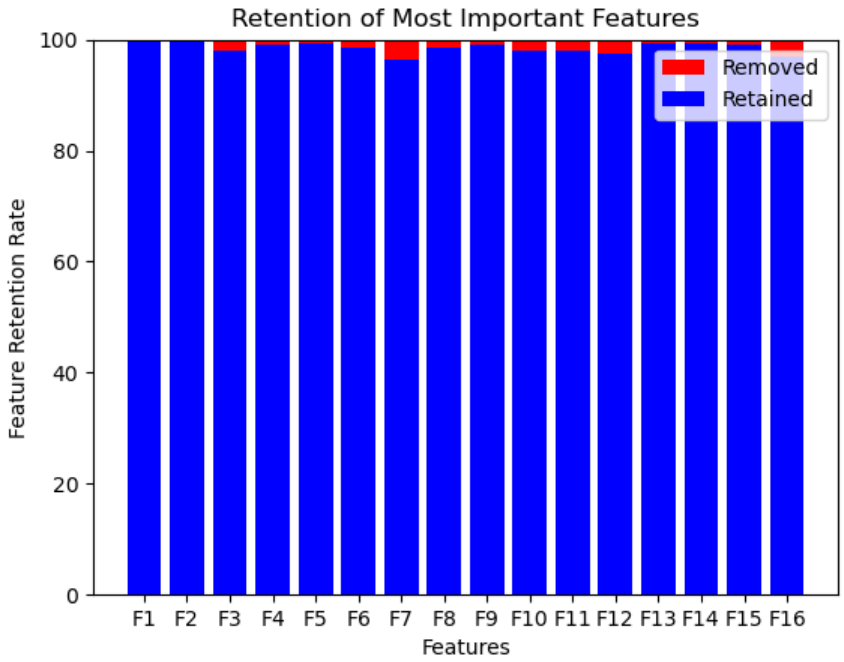

Figure 7.10: Obfuscapk Reflection Impact on String Computation Benign Importance Features

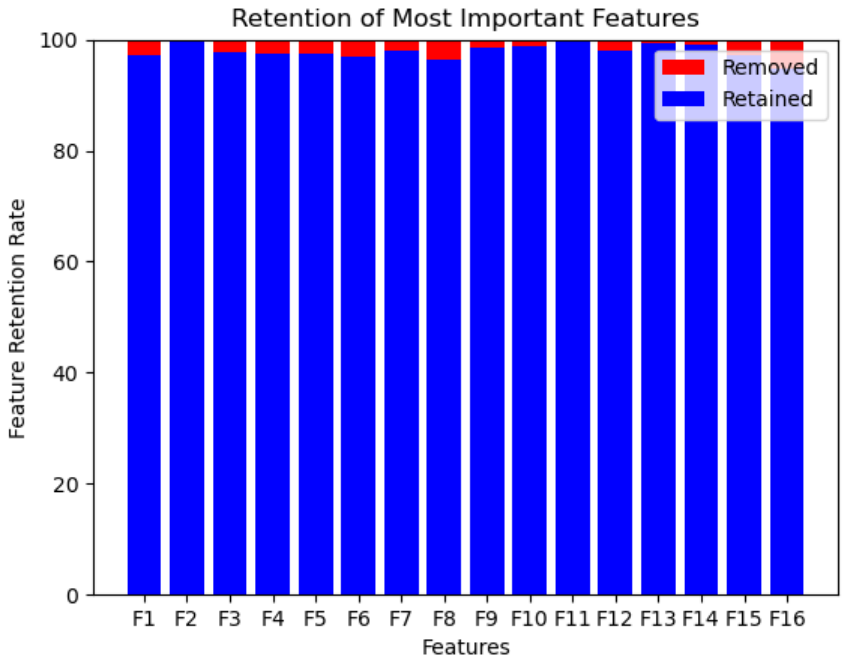

Figure 7.11: AAMO Reflection Impact on String Computation Malware Importance Features

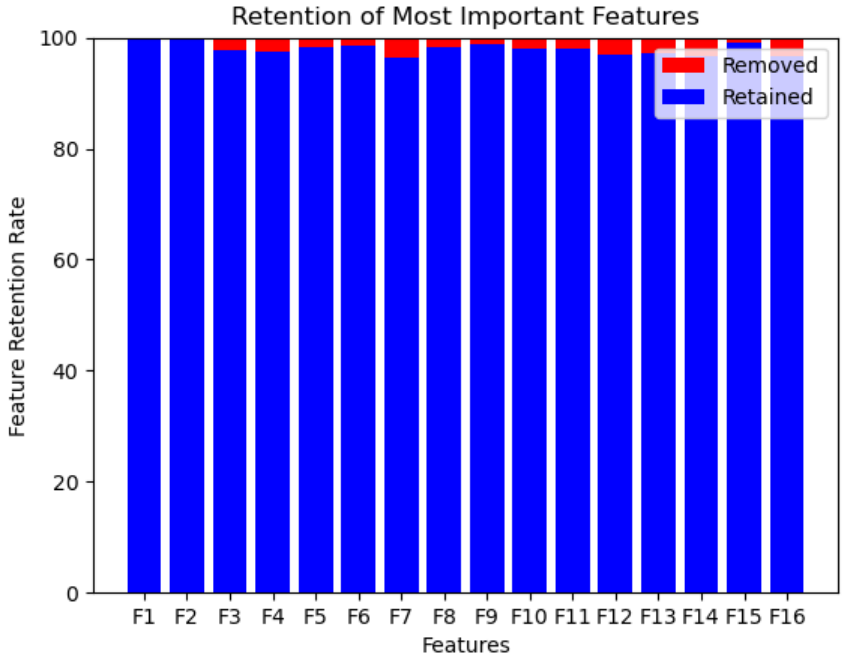

Figure 7.12: AAMO Reflection Impact on String Computation Benign Importance Features

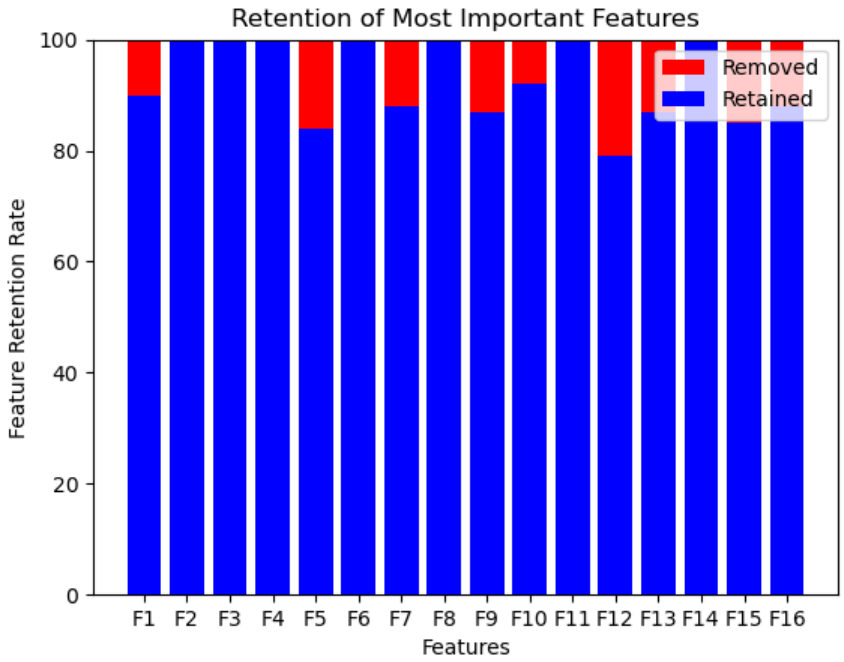

Figure 7.13: Obfuscapk Reflection Impact on String Computation Malware Importance Features

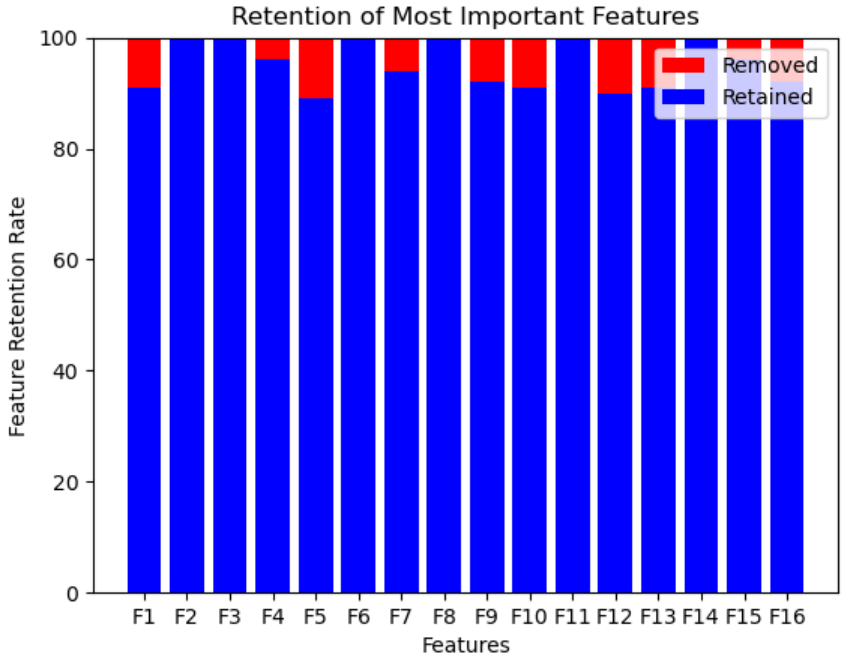

Figure 7.14: Obfuscapk Reflection Impact on String Computation Benign Importance Features

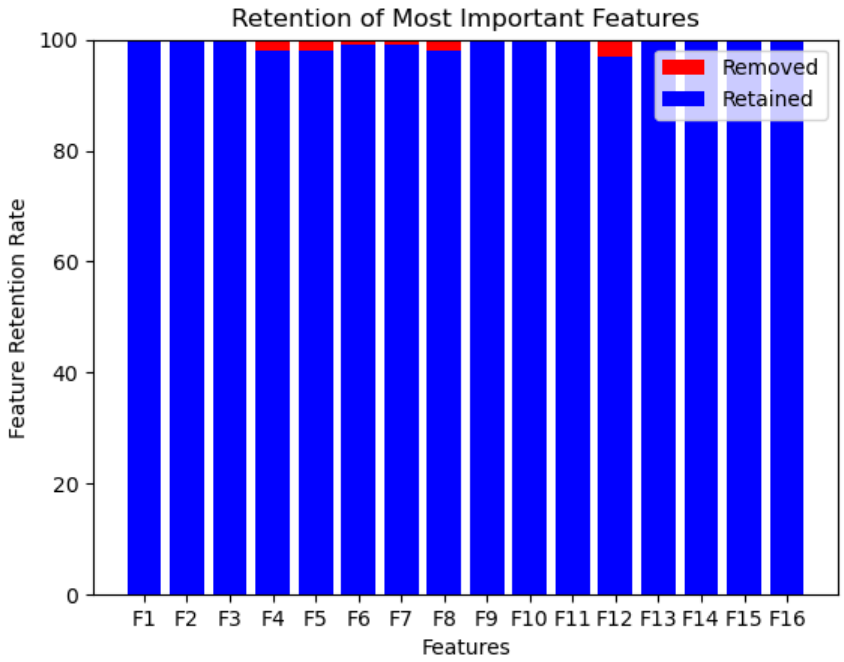

Figure 7.15: AAMO Reflection Impact on RevealDroid Malware Importance Features

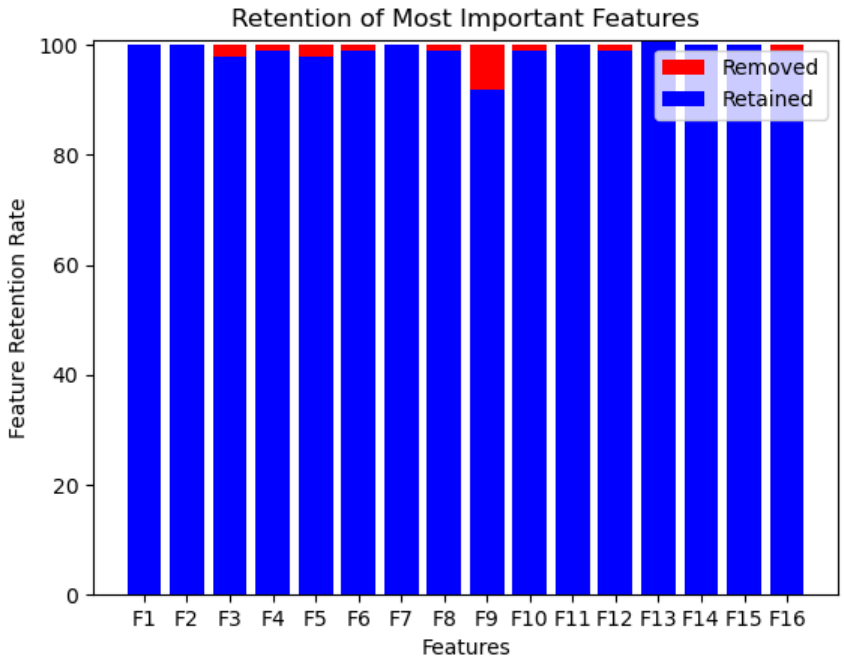

Figure 7.16: AAMO Reflection Impact on RevealDroid Benign Importance Features

7.1.5 Goto Analysis

We next move to goto analysis. In our deep dive analysis of the goto obfuscator for the four example applications, we saw no impact on string computations. There in fact no impact on the feature sets extracted neither across string computations or RevealDroid for both Obfuscapk and AAMO. This was discussed earlier where it was identified that the manner in which goto is implemented by both AAMO and Obfuscapk has no impact on string computations or RevealDroid. Our first set of analysis results did show slight dips for some of the goto features and we believed we had identified an interesting edge case. However, these dips were indicative of apps whose processing was stopped by the CCR cluster before they could complete due to jobs exceeding their time allotment. These cases were easily identified and removed from our analysis across all the obfuscators examined.

7.1.6 Indirection Analysis

Our last set of analyses focuses on how indirection impacts the importance features. Indirection has the potential to be problematic as it introduces new method calls into the control flow specifically to evade signature based detection techniques that focus on the call graph. We first examine the impact of of Obfuscapk in Figures 7.17 and 7.18 and AAMO in Figures 7.19 and 7.20 on the string computations. Rather than report results in terms of removed features we report in terms of broken features and changing the bar coloration to gray. We do this because the string computations are not completely removed due to reachability, rather they are broken by the introduction of indirection with single string computation graphs being broken into 1 or more string computations. In Section 6.1.3 we provide an algorithm to remediate most of the broken string computations. Indirection creates the greatest problem for classification by string computations as it breaks enough critical features to dip accuracy by 4% versus the unobfuscated apps. However, these broken string computations can be reconstituted using the semantic remediation al-

gorithim as a last step prior to persisting the graph database output by our string analysis algorithm. Addition of such a post processing step makes sense as we seek to put string computations into a semantically equivalent form - which semantic remediation does. Correcting the broken string computations would likely increase the accuracy of the classifier when examining apps obfuscated by indirection.

We next move to an analysis of indirection obfuscation's impact on the RevealDroid importance features. We see that accuracy for indirection dips by nearly 4% for Obfuscapk indirection. The graphs yield an interesting finding. Intuitively, we would expect little to no variance in the number of features extracted by RevealDroid as indirection simply introduces new methods - not obfuscate API calls. However, we see that feature counts both go up and go down.

We focus first on the increase in feature counts - what is happening? RevealDroid counts API calls to sensitive Android APIs but Obfuscapk is not creating new APIs. However, consider the code snippet of included in Figure 7.21 which is an actual indirection call added by Obfuscapk. Point 1 of the Figure identifies the package name, `androidx.appcompat.widget`. This is a package structure RevealDroid harvests. Point 2 identifies the class, `AppCompatCheckedTextView`, a class that RevealDroid is interested in. At this point, RevealDroid will harvest any calls to methods of that class. This means it will also count the Point 3 method - `EcPGfVmAIy1uVYD()`. This method is added by Obfuscapk indirection and decidedly *not* a part of the `AppCompatCheckedTextView` class API. However, RevealDroid is not aware of this and simply counts it as yet another method invocation of that class. In some classes, hundreds of new methods are introduced by indirection leading to wildly inaccurate count totals in terms of RevealDroid's harvest targets.

We also note a decrease in the count totals of some features and examined what might contribute to this. A review of RevealDroid's logs indicates a common message as its Soot implementation attempts to process methods added by indirection - "Method has no active

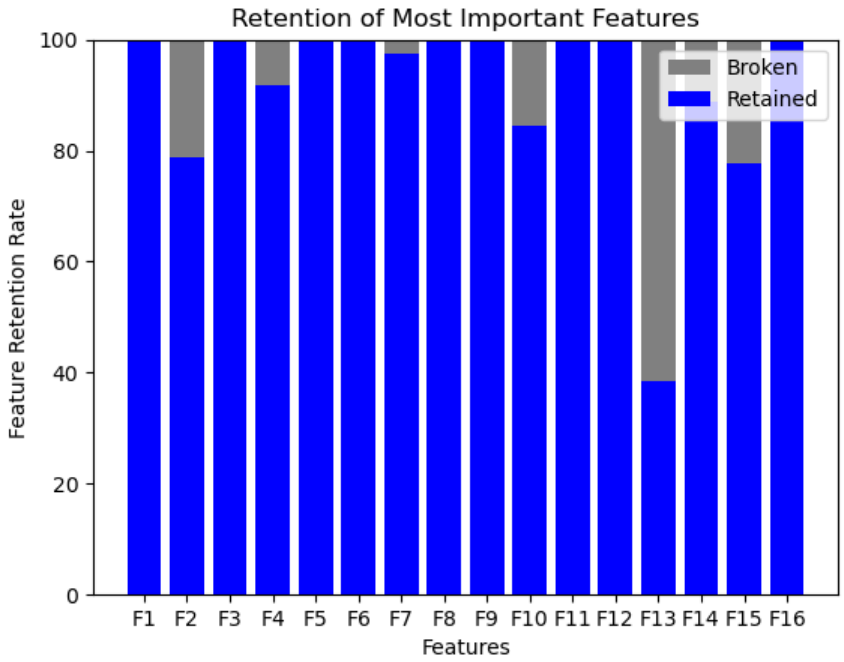

Figure 7.17: Obfuscapk Indirection Impact on String Computation Malware Importance Features

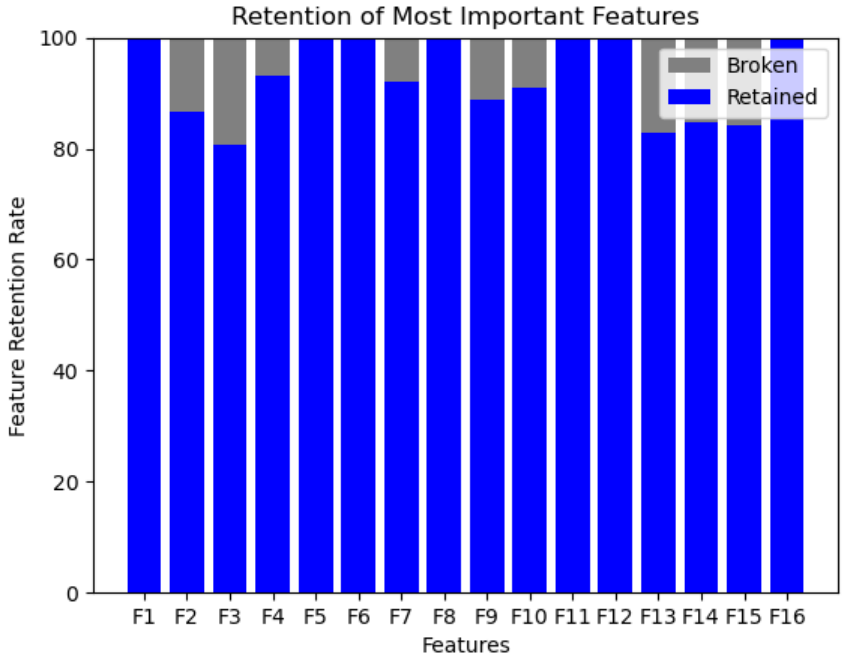

Figure 7.18: Obfuscapk Indirection Impact on String Computation Benign Importance Features

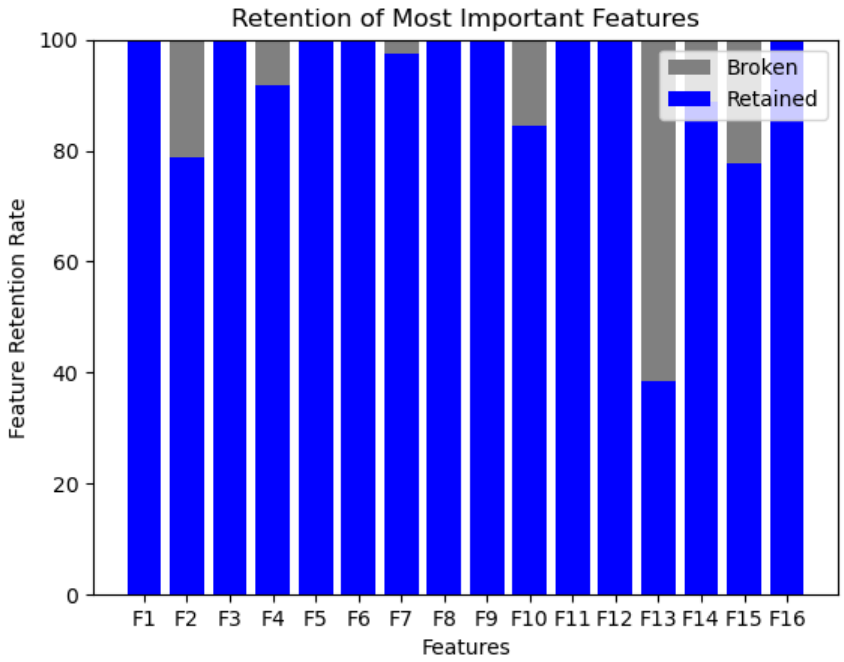

Figure 7.19: AAMO Indirection Impact on String Computation Malware Importance Features

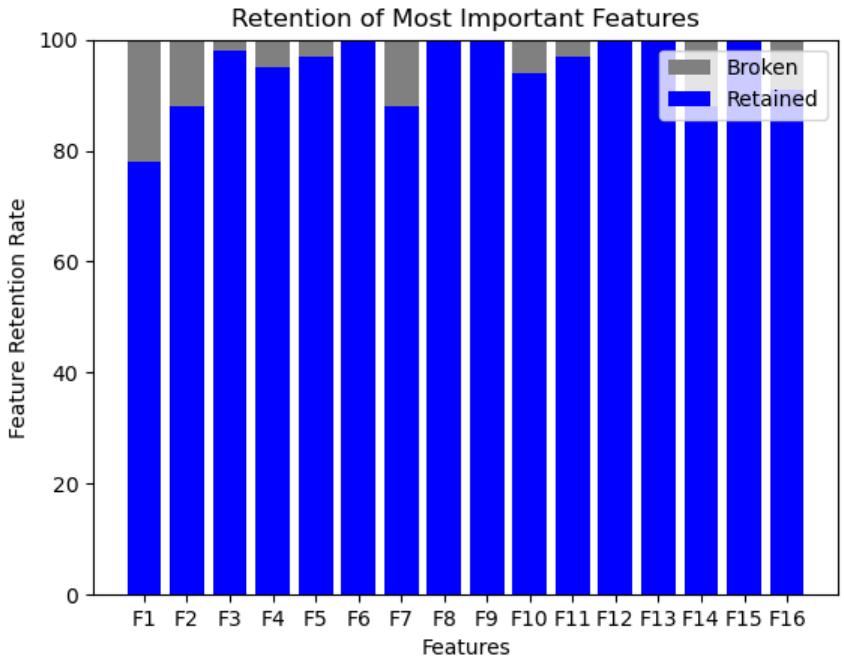

Figure 7.20: AAMO Indirection Impact on String Computation Benign Importance Features

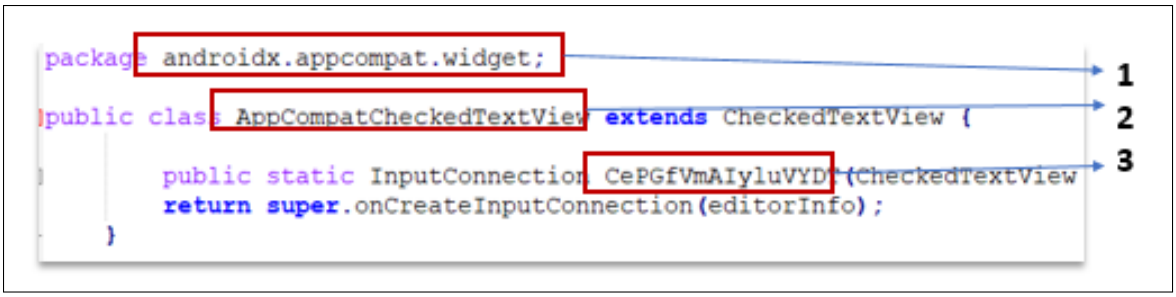

Figure 7.21: Example of Indirection Increasing Number of Sensitive API Calls

body." This is a message well known to even novice Soot users and is indicative of Soot being aware of a method in the call graph but unable to find its source code. This was not a problem for our implementation of Soot which which was able to successfully find and process the indirection method bodies. It may be the case that RevealDroid needs to upgrade to the most recent version of Soot.

Indirection introduced by AAMO demonstrates the same characteristics as that introduced by Obfuscapk, but not as extreme. We see the reduction in importance variable and a slight increase.

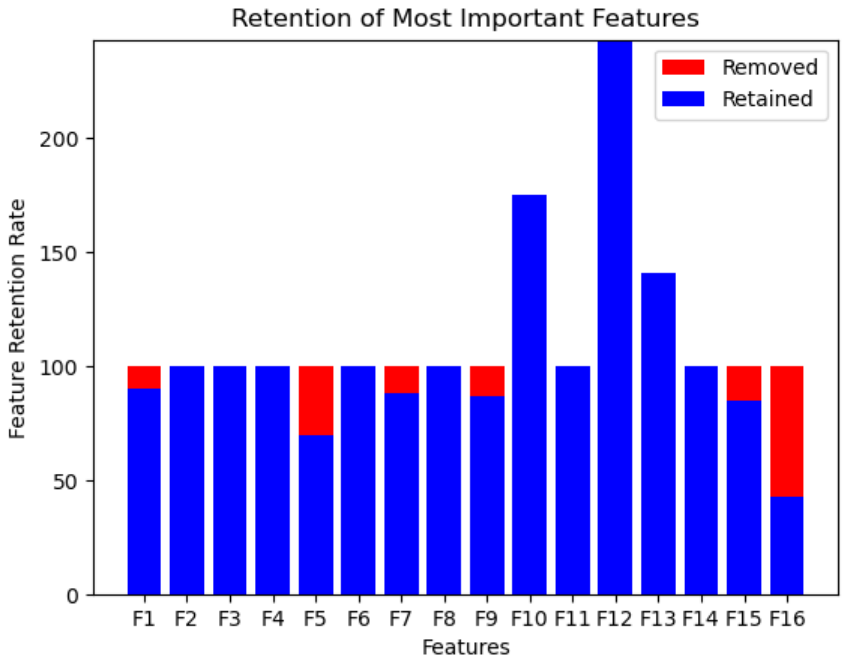

Figure 7.22: Obfuscapk Indirection Impact on RevealDroid Malware Importance Features

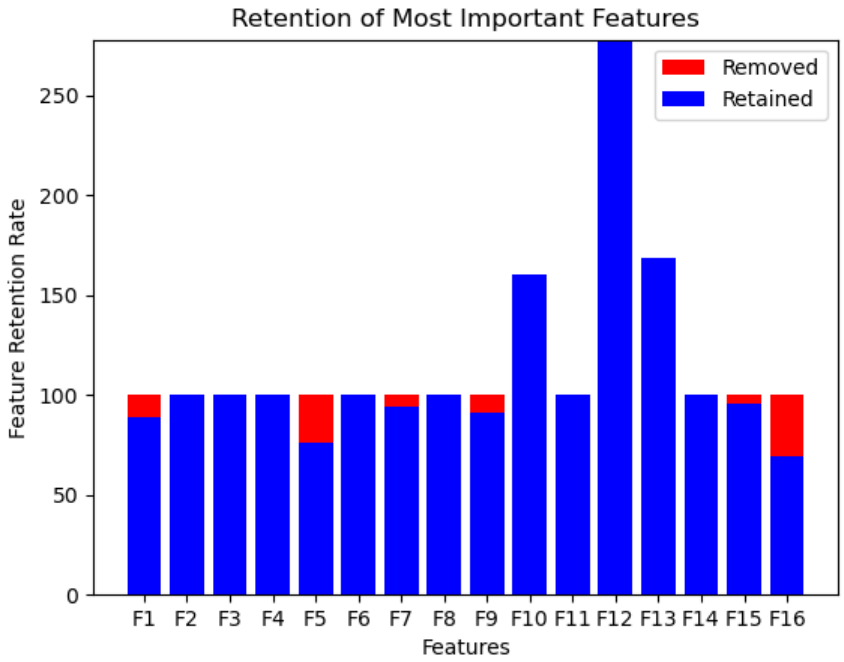

Figure 7.23: Obfuscapk Indirection Impact on RevealDroid Benign Importance Features

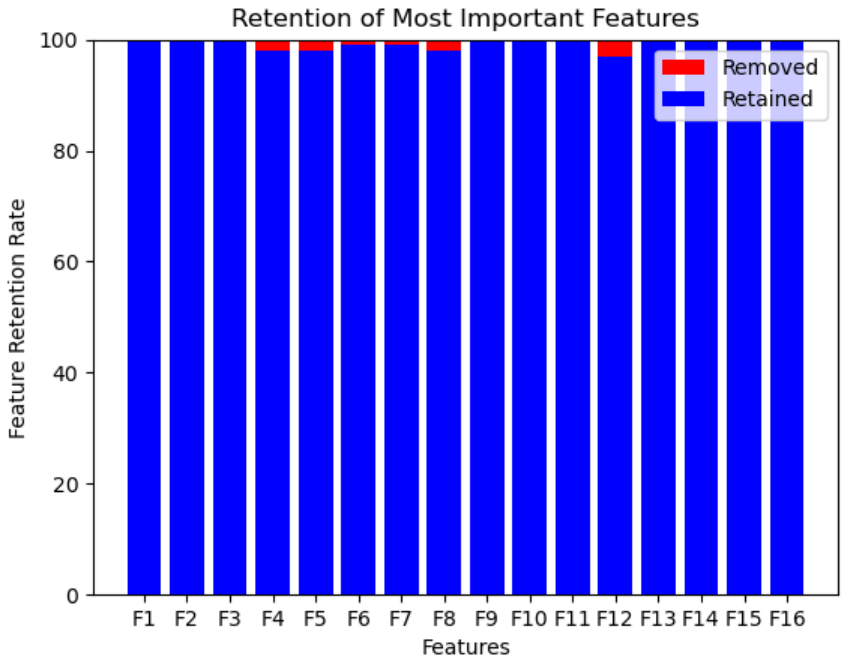

Figure 7.24: AAMO Indirection Impact on RevealDroid Malware Importance Features

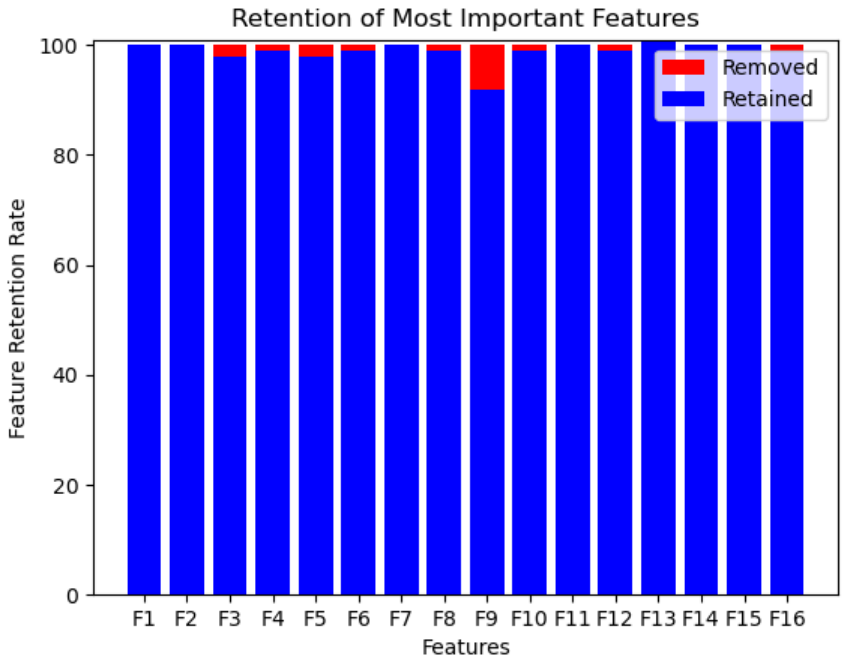

Figure 7.25: AAMO Indirection Impact on RevealDroid Benign Importance Features

Chapter 8

Conclusions

We have identified three major findings as a part of this research. These are summarized following.

8.1 String Computations are a Useful Feature for Classification and Obfuscation Resilient

String computations are incredibly resilient as a feature for classification of malware and benign apps. The computations are a snapshot of an app's control flow with respect to string creation and this snapshot is useful in the classification of malware and benign apps. The results demonstrate clear differences with respect to how malware and benign applications employ strings. In the case of malware these string computations often insect with sensitive information on Android devices, from using strings as a repository to store critical information before exfiltration from the device to capture of information about the device hardware done in order to help running processes evade detection by virus detection software.

String computations are also obfuscation resilient. We see that across multiple types of obfuscators (reflection, goto, and indirection) string computations remained in tact. In cases where the computations were broke by the introduction of obfuscation such as indirection,

it was reasonably easy to reconstruct their original form *solely* from the persisted string computations. This point is vitally important as it can be expanded in the future to create fuzzy matching between graph patterns to identify semantic equivalence between mostly similar string computations for a more robust feature set. Additionally, in the case where string computations were removed entirely due to reachability, as is the case for reflection, enough string computations remained in tact to keep accuracy rates close between when ML models were tested on unobfuscated and obfuscate apps. This indicates that string computation differences between benign and malware apps exist across the application, and are not centralized in a few, sensitive methods.

Taken as whole, string computations tell the story of an app and its behaviour. Reading a listing of string computations for an app quickly provides insight into what it is doing and how risky it is. Throughout the research we commonly scanned through the listing of string computations for malware and benign apps and easily identified differences in behavior simply through this manual inspection. An interesting future research task would be translation of the string computations themselves into a narrative that is human readable for virus detection developers to review. Such content could be added to a the VirusTotal description of a malware app and give more semantic information about behavior than is currently provided.

Persisting graph computations is also incredibly useful. Traditional static analysis approaches compute data flows of interest during the static analysis, answer specific questions, and discard the analysis itself. However, there is *immense* value in persisting the graph computations as a host of add on questions can quick and easily be asked and answered without the need to rerun the complete static analysis. It also allows us to understand how string computations are connected across different apps. We noted in our analysis that string computations began to become connected across apps when string computations were stored in a single, persisted graph store.

8.2 Obfuscation Resilience Requires an Examination of Feature Impact

Any claim of obfuscation resilience by a malware detection tool requires both accuracy results to show resiliency and a thorough examination of the impact of features used for classification. This analysis of importance features must report *how* those features are impacted by the obfuscation.

Understanding how features change in the presence of obfuscation is exactly what malware developers do. They will download malware detection tools, understand the features exhibited in their applications that enable detection and perform trial and error with different types of obfuscation until those features are sufficiently masked. This is exactly the case for Obfuscapk where the reflection obfuscator is refined to make it more effective. Developers of malware detection algorithms must now go an additional step and understand how obfuscation changes the features and whether a competent obfuscator developer could use this information to create a more effective obfuscator.

8.3 Obfuscation is Applied Greedily

The obfuscators evaluated apply obfuscation greedily, one step at a time. They disassemble an app's code and iterate through it one line at a time. When an opportunity to apply a specific type of obfuscation, it is taken. However, the obfuscation's application is with respect to that single line, a fact that has significant implications.

First, it leads to obfuscated apps that may on the surface look significantly opaque but in reality are not. The apparent complexity of the application is the result of a single operation performed on a single line of code iteratively, across the entire application. When one understand *how* these single operations are applied, it is reasonably easy to translate them back to their original form. Even when multiple obfuscators are executed in a sequence, it

is still straight forward to reverse the code back to the original form. We demonstrated this with our semantic remediation which easily corrected broken string computations once the logic of the underlying obfuscation was understood. However, we did need to perform a rigorous analysis of the obfuscation tool to gain this understanding of how obfuscation was applied.

Second, the obfuscation tools we examined dud not try to maintain the state of the code to create truly complex obfuscation. That is, the tools did not maintain the control flow graph for a method to understand larger patterns that may be obfuscated upon. This would be an interesting research question related to obfuscation and a potential area where machine learning could be employed to learn large segments of code where more complex obfuscation may be applied.

Bibliography

[1] Yousra Aafer, Wenliang Du, and Heng Yin. "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*. 2013, pp. 86–103.

[2] Kevin Allix et al. "AndroZoo". In: (2016), pp. 468–471. DOI: 10.1145/2901739.2903508.

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

[4] E. J. Alqahtani, R. Zagrouba, and A. Almuhaideb. "A Survey on Android Malware Detection Techniques Using Machine Learning Algorithms." In: (2019), pp. 110–117.

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

[6] " Androguard ". In: (2020). URL: http://github.com/androguard/androguard.

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

[8] Simone Aonzo et al. “Obfuscapk: An open-source black-box obfuscation tool for Android apps”. In: *SoftwareX* 11 (2020), p. 100403. ISSN: 23527110. DOI: 10.1016/j.softx.2020.100403. URL: https://doi.org/10.1016/j.softx.2020.100403.

[9] “ Apktool ”. In: (2020). URL: https://ibotpeaches.github.io/Apktool.

[10] Axelle Apvrille and Ruchna Nigam. “Obfuscation in Android malware, and how to fight back”. In: July (2014), p. 10. URL: https://www.virusbtn.com/virusbulletin/archive/2014/07/vb201407-Android-obfuscation%7B%5C#%7Dcitation.22.

[11] Daniel Arp et al. *Drebin: Effective and explainable detection of android malware in your pocket*. 2014.

[12] Daniel Arp et al. *Drebin: Effective and explainable detection of android malware in your pocket*. 2014.

[13] Steven Arzt and Eric Bodden. “StubDroid: Automatic Inference of Precise Data-flow Summaries for the Android Framework”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 725–735. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884816. URL: http://doi.acm.org/10.1145/2884781.2884816.

[14] Steven Arzt et al. “FlowDroid : Precise Context , Flow , Field , Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps”. In: *PLDI ’14 Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014), pp. 259–269. ISSN: 15232867. DOI: 10.1145/2594291.2594299.

[15] Steven Arzt et al. “FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps”. In: *Proceedings of the 35th*

ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14. Edinburgh, United Kingdom: ACM, 2014. ISBN: 978-1-4503-2784-8.

[16] Steven Arzt et al. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Appstion in TCB Source Code". In: *PLDI '14*. Edinburgh, UK, 2014.

[17] Vitalii Avdiienko et al. "Mining apps for abnormal usage of sensitive data". In: *Proceedings - International Conference on Software Engineering*. 2015. ISBN: 9781479919345. DOI: 10.1109/ICSE.2015.61.

[18] Eran Avidan and Dror G. Feitelson. "From Obfuscation to Comprehension". In: *IEEE International Conference on Program Comprehension* 2015-August (2015), pp. 178–181. DOI: 10.1109/ICPC.2015.27.

[19] Michael Backes et al. "On demystifying the android application framework: Revisiting android permission specification analysis". In: *Proceedings of the 25th USENIX Security Symposium* (2016), pp. 1101–1116.

[20] Shikha Badhani and Sunil K. Muttoo. "Analyzing Android Code Graphs against Code Obfuscation and App Hiding Techniques". In: *Journal of Applied Security Research* 14.4 (2019), pp. 489–510. DOI: 10.1080/19361610.2019.1667165. eprint: https://doi.org/10.1080/19361610.2019.1667165. URL: https://doi.org/10.1080/19361610.2019.1667165.

[21] Vivek Balachandran et al. "Control flow obfuscation for Android applications". In: *Computers and Security* 61.2016 (2016), pp. 72–93. ISSN: 01674048. DOI: 10.1016/j.cose.2016.05.003. URL: http://dx.doi.org/10.1016/j.cose.2016.05.003.

[22] Paulo Barros et al. "Static analysis of implicit control flow: Resolving Java reflection and Android intents". In: *ASE 2015: Proceedings of the 30th Annual Interna-*

tional Conference on Automated Software Engineering. Lincoln, NE, USA, Nov. 2015.

[23] Dirk Beyer and Ashgan Fararooy. "A simple and effective measure for complex low-level dependencies". In: *IEEE International Conference on Program Comprehension* (2010), pp. 80–83. DOI: 10.1109/ICPC.2010.49.

[24] Tevfik Bhultan et al. *String Analysis for Software Verification and Security*. Cham, Switzerland: Springer, 2017.

[25] Jason Brownlee. *No Title*. 2016. URL: https://machinelearningmastery.com/feature-importance-and-feature-selection-with-xgboost-in-python/.

[26] Christian Bunse. "On the Impact of Code Obfuscation to Software Energy Consumption". In: *From Science to Society*. Springer, 2017, pp. 239–249.

[27] Iker Burguera, Urko Zurutuza, and Smin Nadjm-Tehrani. "Crowdroid: behavior-based malware detection system for Android". In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. SPSM '11. Chicago, Illinois, USA: ACM, 2011. ISBN: 978-1-4503-1000-0.

[28] Zhenquan Cai and Roland H.C. Yap. "Inferring the detection logic and evaluating the effectiveness of android anti-virus apps". In: *CODASPY 2016 - Proceedings of the 6th ACM Conference on Data and Application Security and Privacy* January 2009 (2016), pp. 172–182. DOI: 10.1145/2857705.2857719.

[29] Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. "Code defactoring: Evaluating the effectiveness of Java obfuscations". In: *Proceedings - Working Conference on Reverse Engineering, WCRE* (2012), pp. 71–80. ISSN: 10951350. DOI: 10.1109/WCRE.2012.17.

[30] Saurabh Chakradeo et al. “MAST: Triage for Market-scale Mobile Malware Analysis”. In: *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*. WiSec ’13. Budapest, Hungary: ACM, 2013, pp. 13–24. ISBN: 978-1-4503-1998-0. DOI: 10.1145/2462096.2462100. URL: http://doi.acm.org/10.1145/2462096.2462100.

[31] Tianqi Chen. *Introduction to Boosted Trees*. URL: https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf.

[32] S. Chidamber and C. Kemerer. “Towards a metrics suite for object oriented design”. In: *OOPSLA ’91: Conference proceedings on Object-oriented programming systems, languages, and applications* (1991), pp. 197–211.

[33] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. “Precise Analysis of String Expressions”. In: int id (2003), pp. 1–18. ISSN: 03029743. DOI: 10.1007/3-540-44898-5_1. URL: http://link.springer.com/10.1007/3-540-44898-5%7B%5C_%7D1.

[34] Mihai Christodorescu et al. “Semantics-Aware Malware Detection”. In: *Proceedings of the 2005 IEEE Symposium on Security and Privacy*. SP ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 32–46. ISBN: 0-7695-2339-0. DOI: 10.1109/SP.2005.20. URL: http://dx.doi.org/10.1109/SP.2005.20.

[35] Georgiu Claudiu. “ Obfuscapk ”. In: (2019). URL: https://github.com/ClaudiuGeorgiu/Obfuscapk.

[36] C Collberg, C Thomborson, and D Low. “A taxonomy of obfuscating transformations”. In: *Computer Science Technical Reports* 148 (1997), pp. 36–48. ISSN: 11733500. DOI: 10.1.1.38.9852.

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

[38] Oracle Corporation. *The Java Tutorials*. [Online; accessed 15-March-2017]. 2018. URL: https://docs.oracle.com/javase/tutorial/java/data/index.html.

[39] A. M. Corral. "DroidAPIMiner". In: *Servicios de Publicación UCM* 9.1 (2013), pp. 67–94. ISSN: 18678211. DOI: 10.1007/978-3-319-04283-1_6.

[40] " Dex2Jar ". In: (2020). URL: https://bitbucket.org/pxb1988/dex2jar.

[41] " DexDump ". In: (2020). URL: https://android.googlesource.com/platform/dalvik/+/09239e3/dexdump/DexDump.c.

[42] Shuaike Dong et al. "Understanding android obfuscation techniques: A large-scale investigation in the wild". In: *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST* 254 (2018), pp. 172–192. ISSN: 18678211. DOI: 10.1007/978-3-030-01701-9_10. arXiv: 1801.01633.

[43] William Enck, Machigar Ongtang, and Patrick McDaniel. "On Lightweight Mobile Phone Application Certification". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. CCS '09. Chicago, Illinois, USA: ACM, 2009, pp. 235–245. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653691. URL: http://doi.acm.org/10.1145/1653662.1653691.

[44] Michael D. Ernst et al. "Collaborative Verification of Information Flow for a High-Assurance App Store". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014. ISBN: 978-1-4503-2957-6.

[45] Adrienne Porter Felt et al. "Android Permissions Demystified". In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: ACM, 2011. ISBN: 978-1-4503-0948-6.

[46] Yu Feng et al. “Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis”. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 576–587. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635869. URL: http://doi.acm.org/10.1145/2635868.2635869.

[47] Yu Feng et al. “Apposcopy: semantics-based detection of Android malware through static analysis”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014* (2014), pp. 576–587. DOI: 10.1145/2635868.2635869. URL: http://dl.acm.org/citation.cfm?doid=2635868.2635869.

[48] Felix Freiling, Mykola Protsenko, and Yan Zhuang. “An Empirical Evaluation of Software Obfuscation Techniques Applied to Android APKs”. In: Dec. 2015. DOI: 10.1007/978-3-319-23802-9_24.

[49] Jerome Friedman. “Stochastic Gradient Boosting”. In: *Computational Statistics and Data Analysis* 38 (Feb. 2002), pp. 367–378. DOI: 10.1016/S0167-9473(01)00065-2.

[50] Xiang Fu et al. “A static analysis framework for detecting SQL injection vulnerabilities”. In: *Proceedings - International Computer Software and Applications Conference* 1.Compsac (2007), pp. 87–94. ISSN: 07303157. DOI: 10.1109/COMPSAC.2007.43.

[51] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. *SCanDroid: Automated Security Certification of Android Applications*.

[52] Joshua Garcia, Mahmoud Hammad, and Sam Malek. “Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware”. In: *ACM Transactions on Software Engineering and Methodology* 26.3 (2018), pp. 1–29. ISSN:

1049331X. DOI: 10.1145/3162625. URL: http://dl.acm.org/citation.cfm?doid=3177743.3162625.

[53] Joshua Garcia et al. *Obfuscation-Resilient, Efficient, and Accurate Detection and Family Identification of Android Malware*. Tech. rep. URL: http://cs.gmu.edu/703-993-1530.

[54] Google. *Shrink, obfuscate, and optimize your app*. 2020. URL: https://developer.android.com/studio/build/shrink-code%7B%5C#%7Dconfiguration-files.

[55] Michael Grace et al. "RiskRanker: Scalable and Accurate Zero-day Android Malware Detection". In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. MobiSys '12. Low Wood Bay, Lake District, UK: ACM, 2012, pp. 281–294. ISBN: 978-1-4503-1301-8. DOI: 10.1145/2307636.2307663. URL: http://doi.acm.org/10.1145/2307636.2307663.

[56] Kent Griffin et al. "Automatic Generation of String Signatures for Malware Detection". In: *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*. RAID '09. Saint-Malo, France: Springer-Verlag, 2009, pp. 101–120. ISBN: 978-3-642-04341-3. DOI: 10.1007/978-3-642-04342-0_6. URL: http://dx.doi.org/10.1007/978-3-642-04342-0_6.

[57] Ben Gruver. *No Title*. 2020. URL: https://github.com/JesusFreke/smali.

[58] William G J Halfond and Alessandro Orso. "Amnesia". In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering - ASE '05* 5 (2005), p. 174. ISSN: 20760930. DOI: 10.1145/1101908.1101935. arXiv: 1203.3324. URL: http://doi.acm.org/10.1145/1101908.

1101935%7B%5C%%7D5Cnhttp://portal.acm.org/citation.cfm?doid=1101908.1101935.

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

[60] Behnaz Hassanshahi and Roland H.C. Yap. "Android Database Attacks Revisited". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS '17*. 2017. ISBN: 9781450349444. DOI: 10.1145/3052973.3052994.

[61] Pieter Hooimeijer et al. "Fast and precise sanitizer analysis with BEK". In: *Proceedings of the 20th USENIX Security Symposium* (2011), pp. 1–16.

[62] Shohreh Hosseinzadeh et al. "Diversification and obfuscation techniques for software security: A systematic literature review". In: *Information and Software Technology* 104 (2018), pp. 72–93. ISSN: 09505849. DOI: 10.1016/j.infsof.2018.07.007. URL: https://doi.org/10.1016/j.infsof.2018.07.007.

[63] Jianjun Huang, Xiangyu Zhang, and Lin Tan. "Detecting Sensitive Data Disclosure via Bi-directional Text Correlation Analysis". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: ACM, 2016, pp. 169–180. ISBN: 978-1-4503-4218-6. DOI: 10.1145/2950290.2950348. URL: http://doi.acm.org/10.1145/2950290.2950348.

[64] Wei Huang et al. "Scalable and Precise Taint Analysis for Android". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: ACM, 2015, pp. 106–117. ISBN: 978-1-4503-3620-8.

DOI: 10.1145/2771783.2771803. URL: http://doi.acm.org/10.1145/2771783.2771803.

[65] Richard Killam, Paul Cook, and Natalia Stakhanova. “Android Malware Classification through Analysis of String Literals”. In: (2014).

[66] Bodong Li et al. “APPSPEAR: Automating the hidden-code extraction and reassembling of packed android malware”. In: *Journal of Systems and Software* 140 (2018), pp. 3–16. ISSN: 01641212. DOI: 10.1016/j.jss.2018.02.040. URL: https://doi.org/10.1016/j.jss.2018.02.040.

[67] Ding Li et al. “Automated energy optimization of HTTP requests for mobile applications”. In: *Proceedings of the 38th International Conference on Software Engineering - ICSE ’16*. 2016. ISBN: 9781450339001. DOI: 10.1145/2884781.2884867. arXiv: arXiv:1508.06655v1.

[68] Ding Li et al. “String Analysis for Java and Android Applications”. In: ().

[69] Li Li et al. “IccTA: Detecting Inter-Component Privacy Leaks in Android Apps”. In: *2015 International Conference on Software Engineering (ICSE)*. To appear. 2015. URL: http://www.bodden.de/pubs/lbb+15iccta.pdf.

[70] Zhiqiang Li et al. “Obfusifier: Obfuscation-Resistant Android Malware Detection System”. In: *Security and Privacy in Communication Networks*. Ed. by Songqing Chen et al. Cham: Springer International Publishing, 2019, pp. 214–234. ISBN: 978-3-030-37228-6.

[71] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. “MARVIN: Efficient and Comprehensive Mobile App Classification through Static and Dynamic Analysis”. In: *Proceedings - International Computer Software and Applications Conference* 2 (2015), pp. 422–433. ISSN: 07303157. DOI: 10.1109/COMPSAC.2015.103.

[72] K. Liu et al. “A Review of Android Malware Detection Approaches Based on Machine Learning”. In: *IEEE Access* 8 (2020), pp. 124579–124607.

[73] Long Lu et al. “CHEX: statically vetting Android apps for component hijacking vulnerabilities”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. CCS ’12. 2012.

[74] Davide Maiorca et al. “Stealth attacks: An extended insight into the obfuscation effects on Android malware”. In: *Computers and Security* 51.March 2014 (2015), pp. 16–31. ISSN: 01674048. DOI: 10.1016/j.cose.2015.02.007. URL: http://dx.doi.org/10.1016/j.cose.2015.02.007.

[75] Thomas J Mccabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* 4 (1976), pp. 308–320.

[76] O. Mirzaei et al. “ANDRODET: An adaptive Android obfuscation detector”. In: *Future Generation Computer Systems* 90 (2019), pp. 240–261. ISSN: 0167739X. DOI: 10.1016/j.future.2018.07.066.

[77] “Mobile Malware Analysis : Tricks used in Anubis”. In: (2019). URL: https://eybisi.run/Mobile-Malware-Analysis-Tricks-used-in-Anubis/.

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

[79] Veelasha Moonsamy, Moutaz Alazab, and Lynn Batten. “Towards an understanding of the impact of advertising on data leaks”. In: *Int. J. Secur. Netw.* 7.3 (Mar. 2012). ISSN: 1747-8405.

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

[81] Damien Octeau et al. “Effective Inter-component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis”. In: *Proceedings of the 22Nd USENIX Conference on Security*. SEC’13. Washington, D.C.: USENIX Association, 2013. ISBN: 978-1-931971-03-4.

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

[83] Sebastian Poeplau et al. “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, Feb. 2014.

[84] Mila Dalla Preda and Federico Maggi. “Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology”. In: *Journal of Computer Virology and Hacking Techniques* 13.3 (2017), pp. 209–232. ISSN: 22638733. DOI: 10.1007/s11416-016-0282-2.

[85] “Protecting software through obfuscation: can it keep pace with progress in code analysis?” In: *ACM Computing Surveys* 49.1 (2016), pp. 1–40. ISSN: 15577341. DOI: 10.1145/2886012.

[86] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. “DroidChameleon: Evaluating Android Anti-Malware against Transformation Attacks”. In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS ’13. Hangzhou, China: Association for Computing Machinery, 2013, pp. 329–334. ISBN: 9781450317672. DOI: 10.1145/2484313.2484355. URL: https://doi.org/10.1145/2484313.2484355.

[87] Thomas Reps. “Program analysis via graph reachability”. In: *Information and Software Technology* 40.11-12 (1998), pp. 701–726. ISSN: 09505849. DOI: 10.1016/s0950-5849(98)00093-7.

[88] R. Roscher et al. “Explainable Machine Learning for Scientific Insights and Discoveries”. In: *IEEE Access* 8 (2020), pp. 42200–42216.

[89] Feng Shen et al. “Android Malware Detection using Complex-Flows”. In: *Proceedings of The 37th IEEE International Conference on Distributed Computing Systems*. ICDCS ’17.

[90] Feng Shen et al. “Information Flows As a Permission Mechanism”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE ’14. Vasteras, Sweden: ACM, 2014. ISBN: 978-1-4503-3013-8.

[91] Abdullah Sheneamer, Swarup Roy, and Jugal Kalita. “A detection framework for semantic code clones and obfuscated code”. In: *Expert Systems with Applications* 97 (2018), pp. 405–420. ISSN: 09574174. DOI: 10.1016/j.eswa.2017.12.040. URL: https://doi.org/10.1016/j.eswa.2017.12.040.

[92] Mahsa Shoaran et al. “Energy-efficient classification for resource-constrained biomedical applications”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8.4 (2018), pp. 693–707. ISSN: 21563357. DOI: 10.1109/JETCAS.2018.2844733.

[93] Rocky Slavin et al. “Toward a Framework for Detecting Privacy Policy Violations in Android Application Code”. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: ACM, 2016, pp. 25–36. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884855. URL: http://doi.acm.org/10.1145/2884781.2884855.

[94] Dual Space. 2018. URL: https://apkpure.com/dual-space-multiple-accounts-app-cloner/com.ludashi.dualspace.

[95] Guillermo Suarez-Tangil et al. “DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware”. In: Mar. 2017. DOI: 10.1145/3029806.3029825.

[96] Guillermo Suarez-Tangil et al. “DroidSieve: Fast and accurate classification of obfuscated android malware”. In: *CODASPY 2017 - Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*. Association for Computing Machinery, Inc, Mar. 2017, pp. 309–320. ISBN: 9781450345231. DOI: 10.1145/3029806.3029825.

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

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

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

[100] Virus Total. 2020. URL: https://www.virustotal.com/gui/file/01ef8a109507c6e3b253ee7f7815382fab5b539f088ed385d971ac3076fc8d23.

[101] Raja Vallée-Rai et al. “Soot - a Java bytecode optimization framework”. In: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. CASCON ’99. Mississauga, Ontario, Canada: IBM Press, 1999.

[102] Justin Del Vecchio et al. “String Analysis of Android Applications (N)”. In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ASE ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 680–685. ISBN: 978-1-5090-0025-8. DOI: 10.1109/ASE.2015.20. URL: http://dx.doi.org/10.1109/ASE.2015.20.

[103] Wei Wang et al. “DroidEnsemble: Detecting Android Malicious Applications with Ensemble of String and Structural Static Features”. In: *IEEE Access* 6 (2018), pp. 31798–31807. ISSN: 21693536. DOI: 10.1109/ACCESS.2018.2835654.

[104] Yan Wang and Atanas Rountev. “Who Changed You? Obfuscator Identification for Android”. In: *Proceedings - 2017 IEEE/ACM 4th International Conference on*

Mobile Software Engineering and Systems, MOBILESoft 2017 (2017), pp. 154–164. DOI: 10.1109/MOBILESoft.2017.18.

[105] Michael S. Ware and Christopher J. Fox. "Securing Java code: heuristics and an evaluation of static analysis tools". In: *Proceedings of the 2008 workshop on Static analysis*. SAW '08. Tucson, Arizona: ACM, 2008, pp. 12–21. ISBN: 978-1-59593-924-1. DOI: 10.1145/1394504.1394506. URL: http://doi.acm.org/10.1145/1394504.1394506.

[106] Gary Wassermann and Zhendong Su. "Sound and precise analysis of web applications for injection vulnerabilities". In: *ACM SIGPLAN Notices* 42.6 (2007), p. 32. ISSN: 03621340. DOI: 10.1145/1273442.1250739.

[107] Mark N. Wegman and F. Kenneth Zadeck. "Constant Propagation with Conditional Branches". In: *ACM Trans. Program. Lang. Syst.* 13.2 (Apr. 1991), pp. 181–210. ISSN: 0164-0925. DOI: 10.1145/103135.103136. URL: http://doi.acm.org/10.1145/103135.103136.

[108] Shengqian Yang et al. "Static Control-flow Analysis of User-driven Callbacks in Android Applications". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 89–99. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818768.

[109] Wei Yang et al. "AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 2015, pp. 303–313. ISBN: 978-1-4799-1934-5. URL: http://dl.acm.org/citation.cfm?id=2818754.2818793.

[110] Mu Zhang et al. "Semantics-aware Android malware classification using weighted contextual API dependency graphs". In: *Proceedings of the ACM Conference on*

Computer and Communications Security (2014), pp. 1105–1116. ISSN: 15437221. DOI: 10.1145/2660267.2660359.

[111] Yuan Zhang et al. “Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS ’13. Berlin, Germany: ACM, 2013. ISBN: 978-1-4503-2477-9.