Android Malware Clustering through Malicious Payload Mining

Yuping Li[1], Jiyong Jang[2], Xin Hu[3], and Xinming Ou[1]

[1]University of South Florida, [2]IBM Research, [3]Pinterest

ABSTRACT

Clustering has been well studied for desktop malware analysis as an effective triage method. Conventional similarity-based clustering techniques, however, cannot be immediately applied to Android malware analysis due to the excessive use of third-party libraries in Android application development and Android application repackaging techniques. For example, two Android malicious apps from different malware families may share high level of overall similarity if both apps include the same popular libraries or both apps are repackaged based on the same original app.

In this paper, we propose novel malicious payload mining techniques to efficiently perform Android malware clustering. In particular, we design a robust method to precisely exclude legitimate library code from Android malware while retaining malicious code segments, even if the malicious code is injected under popular library names. We design and implement an Android malware clustering approach through iterative mining of malicious payload and checking whether malware samples share the same version of malicious payload. Our approach utilizes traditional hierarchical clustering technique and an efficient fuzzy hashing fingerprint representation. We also develop three optimization techniques to significantly improve the scalability, and our performance evaluation confirms the applicability of our approach in analyzing a large scale of malware families with little or no accuracy impact. To evaluate the overall performance, we first leverage VirusTotal reports, clustering techniques, and manual efforts to separate collected malware samples into 260 sub-families; then constructed 10 testing datasets by shuffling the sub-families and randomly select 30 sub-families for each dataset. When applying the proposed clustering approach on the 10 testing datasets constructed as described above, the experimental results demonstrate that the proposed clustering approach achieves average precision of 0.984 and recall of 0.959.

Keywords

Android Malware, Clustering, Fingerprints, Malicious Payloads, Fuzzy Hashing

1. INTRODUCTION

Similar to the trends that have been observed in the desktop malware analysis domain, security companies are receiving an increasing amount of potentially malicious Android apps everyday [24]. It becomes infeasible for security analysts to manually study Android malware families with a large number of samples, and automatic malware scanning typically requires considerable amount of computing resources and may not be able to detect the latest evolving threats. Recent research has shown that clustering analysis can be an effective approach to triage incoming samples. For example, multiple static [13, 15, 17, 18, 28], dynamic [1, 22, 26], and hybrid [14, 27] analysis based clustering techniques have been proposed in the desktop malware domain.

However, conventional similarity-based clustering techniques cannot be immediately adopted for the Android malware analysis domain due to several challenges faced in Android domain. Android applications often include a variety of third-party libraries to implement extra functionalities in a cost-effective way. We measure the library code proportion of 19,738 labeled malware samples, and note that the ratio of library code to every malware sample is at least 31.5%, this means a large portion of Android applications belongs to libraries. Directly applying overall similarity analysis on those applications can generate imprecise results: malicious samples from different malware families may be clustered together simply because they share the same libraries.

A common approach for handling legitimate libraries during Android malware analysis is to use a naming based white-list [3, 4, 5, 8, 11] to exclude all libraries code. However, as demonstrated in this paper, such solution is still problematic in that attackers can disguise their malicious payloads under popular library names to take advantage of the popularity of legitimate libraries. For example, we note that malware authors inject their payload under popular library names, such as `com.google.ssearch`, `com.android.appupdate`, and `com.umeng.adutils`. Consequently, the white-listing approach would unintentionally remove certain real malicious payloads together with legitimate libraries code from analysis. Our analysis confirmed that about 30% of Android malware families actually try to inject their malicious payload under popular libraries.

Ubiquitous library usage and the readily available repacking tools bring in additional challenges for Android malware analysis due to the relative small code size of the malicious payloads. We analyze the ratio of the core malicious payloads[1] to the entire apps for the labeled 19,738 malware samples, and observe that such ratio is between 0.1% and 58.2%. This means that the real content of the malicious payloads is comparatively small, which makes traditional similarity analysis based clustering approach less effective. For example, two malicious samples from different families

[1]Malicious payload identification and extraction are discussed in Section 2.3.2.

can be repackaged based on the same original benign app, thus presenting high level of overall similarity. Likewise, Android malware variants with the same malicious payload of one family can be repackaged on different original benign apps, thus presenting low level of overall similarity.

Moreover, it is also challenging to achieve precise malware family labeling and to prepare ground truth labeled datasets for clustering analysis [21]. For example, we collect 247,932 potentially malicious Android samples, each of which has at least one malicious flag according to VirusTotal [25] reports as of April 18, 2016; however, only 19,738 (8%) of them have consistent anti-virus labels indicating their corresponding malware family names. We frequently see the following generic labels reported by various anti-virus products: `Suspicious`, `Artemis!XXX`, `Unclassified-Malware`, `PUA (Potentially Unwanted Application)`, and `Trojan.AndroidOS.Generic`. In practice, anti-virus products can detect the maliciousness of target apps by checking for the presence of specific behaviors; however, different Android malware families may present similar malicious behaviors (e.g., hiding app icon and sending out premium SMS messages). Therefore, we need to consolidate multiple antivirus scanning results to achieve precise malware family labeling.

In this work, we design a robust method to precisely exclude legitimate library code from Android malware while retaining malicious code segments, even if the malicious code is injected under popular library names. We design and implement an Android malware clustering approach through iterative mining of malicious payload and checking if the malware samples share the same version of the malicious payloads. For simplicity, we refer to the core malicious code segments of an Android malware sample as *malicious payloads*, and a payload can be an added/modified part of the repackaged Android malware. For standalone Android malware, we consider its majority code, except for legitimate library code, as its malicious payload since standalone malware is often based on the same skeleton code and the malicious functionalities may only be triggered under specific context.

Our main contributions are summarized as follows:

- We design a novel method to exclude legitimate library code from Android apps. It is well-known that attackers can inject their malicious payload under existing libraries, however there is no existing solution to reliably distinguish between a legitimate library and a bogus library that share the same library name. Our techniques precisely remove legitimate libraries from an app and still preserve the malicious payloads even if they are injected under popular library names. It can also be used to pinpoint the differential parts to extract the core malicious code segments.

- We propose an Android malware clustering solution by checking if they share the same version of the malicious payloads. By providing the shared malicious payloads within a payload cluster and payload-to-app association information, our approach offers efficient Android malware app clustering along with fundamental insights of malware grouping. The experimental results demonstrate the effectiveness of our solution in practice.

- We conduct extensive experiments to evaluate the consistency and robustness of the proposed clustering solution. We first leverage VirusTotal reports, clustering techniques, and manual efforts to separate collected malware samples into 260 sub-families; then construct 10 testing datasets by shuffling the sub-families and randomly selecting 30 sub-families for each dataset. Our experimental results demonstrate that our clustering approach achieves average precision of 0.984 and recall of 0.959 with regard to the testing datasets.

2. OVERVIEW AND BASIC TECHNIQUES

In this section, we describe the necessary technical background of our work and the overall workflow to perform Android malware clustering.

2.1 Overall Workflow

As illustrated in Figure 1, the overall workflow for conducting Android malware app clustering is as follows:

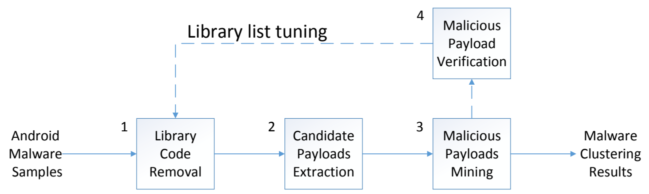
Library list tuning
4
Malicious Payload Verification
Android Malware Samples
1
Library Code Removal
2
Candidate Payloads Extraction
3
Malicious Payloads Mining
Malware Clustering Results

Figure 1: Overall workflow for app clustering

Step 1: Library Code Removal: We design a novel approach to remove legitimate library code from Android apps. Unlike the simple naming based white-list method, our approach can safely exclude legitimate library code while retaining malicious code segments, even if the malicious code is injected under popular library names. We discuss the details of the library code removal technique in Section 2.3.1.

Step 2: Candidate Payload Extraction: After removing the popular library code, we consider the shared code between each malware sample pair as one version of candidate malicious payload. At the same time, the payload-to-app association information is recorded.

Step 3: Malicious Payload Mining: We then apply clustering analysis of the candidate payloads and iterative mining to obtain meaningful shared code patterns that are most likely to be malicious payloads according to several factors, such as code size, sharing frequency, and so on. The malicious payload mining results and the payload-to-app association information are combined to derive app clustering output.

Step 4: Malicious Payload Verification: A less popular library code may be included in the malicious payload mining results, then certain apps may be clustered together because of their shared usage of the library. We generate a summary output of the major classes and functions for each malicious payload cluster, which is then used as supportive evidences for app clustering results. Through manual verification results, the official library list is updated accordingly.

The same procedure is conducted on benign Android applications to acquire the initial library list, and the process will terminate when no meaningful shared code can be further mined.

2.2 Fuzzy Hashing Fundamentals

We utilize nextGen-hash [20] to represent an Android app, including library code and a malicious payload, as a bit-vector fingerprint using n-gram features and feature hashing techniques. The overall procedure to generate a fuzzy hashing fingerprint of an Android app is illustrated in Figure 2 where we list 3 simplified 5-gram features of disassembled Dalvik opcodes sequences.

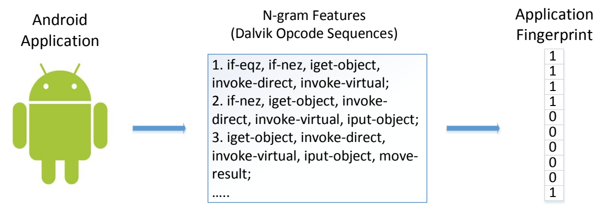
Android Application
N-gram Features (Dalvik Opcode Sequences)
1. if-eqz, if-nez, iget-object, invoke-direct, invoke-virtual;
2. if-nez, iget-object, invoke-direct, invoke-virtual, iput-object;
3. iget-object, invoke-direct, invoke-virtual, iput-object, move-result;
.....
Application Fingerprint
1
1
1
1
0
0
0
0
0
1

Figure 2: Fuzzy hash fingerprint generation procedure

Specifically, we use Dexdump [7] to disassemble `classes.dex` into Dalvik bytecode, and extract n-gram features from disassembled bytecode sequences, then apply feature hashing [17] to map each n-gram feature into one bit in a large bit-vector, and the final bit-vector fingerprint is used to represent an app, and we use the bit-wise intersection analysis capability provided by the bit-vector to achieve library code removal, malicious payload extraction and malware sample clustering. We also keep track of feature-to-bit mapping information, and use it to reconstruct the original payload code from the target bits.

We extract various representative information as a feature set, and use bytecode opcode sequences as main features. Within the disassembled bytecode sequences, "function invocation" instructions are the basis to construct a complete function call graph. We include their corresponding class names, invoked popular function names, and input and output signatures as another source of features. In order to prevent overfitting, we restrict the class names to widely shared entries, such as the classes that are defined under Android core libraries and Java core libraries, and only include the function names that are defined under such popular classes because they do not usually change among applications. For the rest of classes and functions, we replace the concrete names with placeholders considering that they may easily change. Besides widely shared class names under popular core libraries, we also collect the Java VM's representation of type signatures as features. For example, `Z` for Boolean, `B` for byte, and `[` for array types, and so on. These signatures are often located within function invocation instructions and "object field operation" instructions, such as the operand for `iget`, `iput`, `sget` and `sput` opcodes. For the rest of opcodes, we also include the string value of `const-string` based on the observation that strings used by a particular family tend to be distinctive.

After generating the bit-vector fingerprints, we then use bit-wise Jaccard similarity [17] function to check the similarity of fingerprints, and use the 1-bits containment ratio [16] between two fingerprints as containment analysis function.

2.3 Fingerprint based Basic Operations

In this paper, we extensively exploit the bit manipulation capability provided by the bit-vector fingerprint representation as the technical foundation for subsequent analysis. Particularly, we design the following two fingerprint based basic operations (a) to remove the library related bits from an app fingerprint and (b) to extract the shared bits as one version of candidate payload fingerprint.

2.3.1 *Library code removal*

Android apps often use various modularized libraries, such as Android system libraries, Java common libraries, and third party advertisement libraries. Existing malware analysis techniques either take no special actions or ignore all library code based on whitelisted library names. However, when an attacker injects malicious payloads under popular libraries, such as `android.ad.appoffer`, a naïve whitelisting approach may result in imprecise analysis result. In order to reliably extract malicious payloads from labeled malware samples, we precisely exclude legitimate library code from each app while keeping potentially malicious payloads that are injected under the same legitimate library names.

Overall, we exclude legitimate library code from each app by removing the "library-mapped" bits from the app bit-vector fingerprint. For each legitimate libraries, we collect the official jar file and disassemble them into Dalvik bytecode sequences; then use the same feature hashing technique to map the n-gram features of the library code sequences into a 256KB fuzzy hashing fingerprint.

In our implementation, each library is represented with an individual fingerprint while multiple versions of the same library are encoded together. For example, a single library fingerprint of `twitter4j` eventually contains multiple versions of `twitter4j` libraries. For each Android sample, we examine the application and get the contained library name list, and then remove the corresponding library bits from the app fingerprint.

Similar to application fingerprint generation, we map the library n-gram features into bit 1 in a bit-vector fingerprint fp_{lib} that is initialized with bits 0. We then flip all the bits in the library fingerprint to get $\bar{fp_{lib}}$. Since the same library features contained in Android application are mapped to bit 1 in the app fingerprint, the bit-flipped library fingerprint representation enables us to exclude "library-mapped" bits from app fingerprint through intersection analysis between the app fingerprint and library fingerprint. The rest 1-bits in library fingerprint ensure all other non-library bits of app fingerprint intact.

Figure 3 demonstrate the overall procedure to safely remove legitimate `twitter4j` library code from a malware sample. The cells in the fingerprint are either bit 0 or bit 1, indicating whether the corresponding application has a specific feature or not; the underlying number indicating the bit indexes, a feature is mapped to a specific bit index according to feature hashing results. Whenever the malware sample uses any version of the legitimate `twitter4j` library, the intersection analysis between the app fingerprint and library fingerprint will safely remove the corresponding library code from app, while still retaining the potentially malicious code injected under the library namespace. The intersection analysis is only applied when we found the app indeed contains code segments that are defined under `twitter4j` library namespace.

The official libraries are collected incrementally through statistical analyzing of the 19,738 labeled malware samples and clustering analysis of 20000 randomly selected benign apps. We prepare an initial set of libraries by statistically

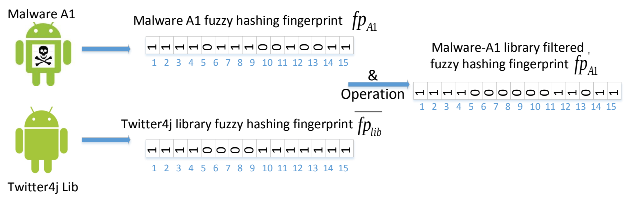

Figure 3: Example procedure to safely remove legitimate "twitter4j" library code

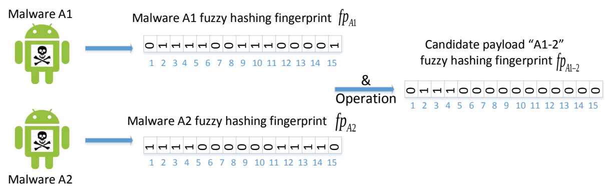

Figure 4: Extracting a candidate payload from two malware applications

analyzing malware samples and manually select the top 60 widely shared libraries. Our Android malware clustering approach is based on malicious payload mining results. The less popular libraries may be also shared by multiple samples and eventually extracted as one version of payload. To remedy the problem, we apply the clustering procedure on benign apps to extract significant shared code patterns and add them to the library list since the majority shared payloads among benign apps are likely legitimate libraries. The clustering procedure is repeatedly applied on 1000 sampled benign apps until convergence[2] where no more significant payload could be extracted.

We collect the majority of official libraries through the library mining process on benign apps except for a few new libraries. This iterative library extracting process also enables us to extract certain generic and legitimate code patterns such as the base64 encoding and decoding routine. Since it is infeasible to cover all of the existing legitimate libraries especially for analyzing unseen Android malware samples, we include an optional malicious payload verification process as shown in Figure 1.

It is likely that multiple versions of the same library are used by different Android applications. The specific version of a library can be located by searching its representative class or function names, examining the contained library code structure, or checking its version related definitions. Once we identify a new version of the library is being used, we map the new version library code to the existing library fingerprint that has the same library name.

Note that the added new library code will be checked against the samples that indeed contain one version of the library, and the potential feature collision may happen between the application code and irrelevant versions of the library code. However, since different versions of the same library typically share high level of code similarity due to code reuse, and the size of the single library is often smaller than the entire application, the collision rate between the application code and irrelevant versions of the library code is negligible.

2.3.2 *Candidate Payload Extraction*

The next fingerprint based basic operation is to extract malicious payload from labeled malware samples. We consider the shared code segments (after excluding legitimate libraries) between each malware sample pair to be a candidate malicious payload.

For the target samples, we first convert the samples into bit-vector fuzzy hashing fingerprints, and then apply the library code removal process whenever needed. We conduct intersection analysis between every library-excluded sample fingerprint pair, and consider the shared 1-bits between the sample fingerprints as a candidate payload fingerprint. Note that the resulting candidate payload fingerprints may do not contain any bit 1 if two underlying sample fingerprints have no shared code.

Figure 4 describes the intersection analysis procedure for extracting one candidate malicious payload at the high level. From two malware samples from malware family `A`, we first build fuzzy hashing fingerprints of both samples and exclude the legitimate library bits from app fingerprints, and then pinpoint their shared 1 bits (e.g., bits index 2, 3, and 4) as potentially malicious bits (e.g., malicious payload mapped) and represent them in a separate payload fingerprint.

After extracting the malicious bits from app fingerprints, we eventually reconstruct the corresponding malicious payload code by checking the feature-to-bit mapping information that are stored during feature hashing process. Particularly, we map an identified bit 1 to an n-gram feature, locate the n lines of code where the n-gram feature are extracted, and reconstruct complete malicious code sequences by stitching the identified n lines of code segments together. This payload code reconstruct procedure helps to recover n lines of code from each feature for every identified 1-bit, to certain extent it compensates feature hashing collisions (e.g., resulting in missing n-grams) since each n-gram feature can be used to recover n line of original code sequences.

Since we can generate a candidate payload from each malware app pair, the candidate payload extraction procedure will create a total of $\frac{n\times(n-1)}{2}$ candidate fingerprints for n malware samples. During the candidate payload extraction procedure, we keep track of the association information between the candidate payload (e.g, `A1-2`) and the corresponding samples (e.g., `A1` and `A2`). We subsequently use the payload-to-app association information and the malicious payload mining results to group malware samples.

3. ANDROID MALWARE CLUSTERING

We conduct hierarchical agglomerative clustering on all of extracted candidate payload fingerprints, so that all similar versions of candidate payload fingerprints will be grouped together. In order to let the real malicious payloads standing out from the rest of extracted candidate payloads, we design a iterative malicious payload mining strategy based on our malware analysis experiences.

Figure 5 shows the overall clustering analysis procedure among five malware samples. Within the Figure, "Library code removal" is applied during app fingerprint generating process whenever needed, "Candidate payload verification"

[2]The resulting cluster fingerprint has less than 70 1-bits or the largest app cluster contain less than 10 entries.

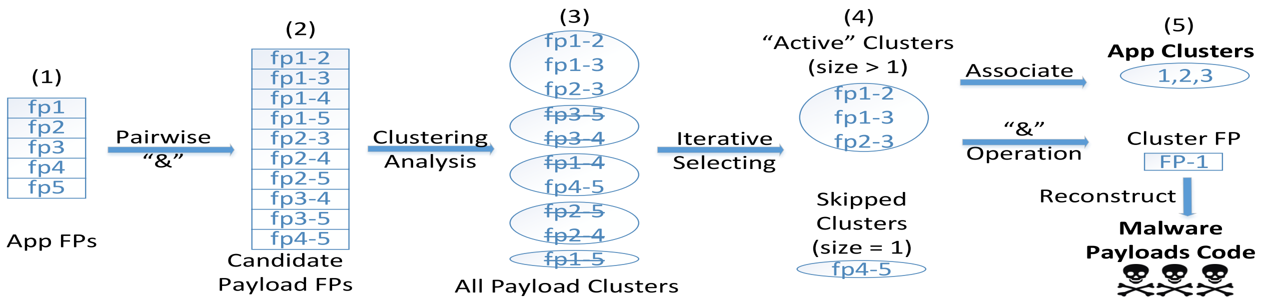

Figure 5: Overall clustering analysis procedure among five malware samples.

is conducted on the reconstructed malicious payload code, they are omitted in the Figure for simplicity. We describe the major steps in more details as follows:

Step 1: we convert malware samples into bit-vector fuzzy hashing fingerprints, and represent them as fp1, fp2, fp3, fp4, fp5 accordingly. The library removal process is applied to fingerprint generation if the sample uses any legitimate libraries.

Step 2: we conduct intersection analysis to extract the shared 1-bits between every malware fingerprint pair, and consider each version of the shared 1-bits as a candidate payload fingerprint. The payload to sample association information are recorded during the analysis. For example, fp1-2 indicates this candidate payload is extracted from malware sample 1 and 2.

Step 3: we perform hierarchical clustering analysis on all generated candidate malware payload fingerprints. As shown in the Figure, candidate payload fingerprints fp1-2, fp1-3, fp2-3 are grouped together as the largest payload cluster based on overall similarity.

Step 4: we iteratively select representative malicious payload clusters using the selecting strategy discussed earlier. For instance, candidate payload cluster containing fp1-2, fp1-3, fp2-3 is selected first because it has the largest cluster size. Candidate payload fingerprints fp1-4, fp1-5, fp2-4, fp2-5, fp3-4, fp3-5 are then skipped as "inactive" entries, since we only expect to extract one version of malicious payload from each malware sample, and the malicious payload for malware sample 1, 2 and 3 has already been recovered through previously selected representative payload cluster. fp4-5 is skipped since it is the only entry in the payload cluster after the updates.

Step 5: we retrieve the malware apps associated with each payload cluster and group them as one app cluster, in which they share the same version of malicious payload. For example, malware samples 1, 2, 3 are grouped together based on candidate payload cluster that contains fp1-2, fp1-3, fp2-3. The corresponding cluster fingerprint for this version of malicious payload is the intersection of all the candidate payload fingerprint entries in the cluster.

We automatically reconstruct the malicious payload code (e.g., Dalvik code sequences with corresponding class and function name) according to the cluster fingerprint and the feature-to-bit mapping information stored during sample fingerprint generation process. The reconstructed malicious payload code can be manually verified to make sure they are indeed the malicious components. Each version of the extracted payload usually has similar classes names and Dalvik code sequences, we note that the maliciousness of the extracted payload can usually be spot by checking the extracted class names, more detailed malicious payload verification results are included in 4.3. In case if the class names are not enough to make a decision, we then go through the reconstructed code segments and check if there are any suspicious activities or behaviors such as stealthily sending out premium SMS, etc.

3.1 Malicious Payload Mining

In practice, thanks to the low collision property of feature hashing, malware samples fingerprints will not contain large number of shared 1-bits unless they do share certain common features (e.g., payload code snippets). Likewise, if the target dataset contains malware samples that do share the same version of malicious payload, then the candidate payloads fingerprints extracted from those samples will contain similar 1-bits, thus will be automatically clustered into the same group. After removing legitimate library code from each app, similar malicious payloads will have highest opportunity to form a comparatively larger cluster than the payloads that related to less popular libraries or completely random shared code segments. And compared with other randomly shared code segments, similar malicious payloads will have a larger shared code base because of "legitimate" reason of code reuse in the same malware family, and the fingerprints for those malicious payloads will have larger number of shared 1-bits.

After obtain hierarchical clustering results with predefined threshold $\theta = 0.75$, we design the following strategies to iteratively select the representative candidate payload clusters that are most likely to contain really malicious payloads based on the above mentioned observations:

- We count the candidate payload fingerprint entries in each cluster, and maximize the possibility of extracting core malicious payloads by selecting the clusters with the largest number of payload fingerprint entries. Payload cluster size l is good indicator for popularity of the shared code segments between malware samples, and the popular shared code is a good candidate for being one version of malicious payload, since we have already filtered out the popular legitimate library code.

- We calculate the distinct apps m that contribute to generating candidate payload fingerprints of each cluster, and select the clusters with largest number of distinct apps if they have same number of payload entries. Payload clusters that contain large number of unique payload entries are usually associated with large number of distinct apps, and we use this information to break the tie in case if the number of cluster entries are the same, since distinct apps can be viewed as another sign of comparative popularity.

- We obtain the intersection bits k of payload fingerprint entries in each cluster as the cluster fingerprint. If two clusters are associated with the same number of distinct apps, we then select the one with the larger number of 1-bits in its cluster fingerprint. In this way, we can extract payload with larger code size, and it helps to increase the likelihood of getting malicious payloads together with shared libraries, and we can subsequently exclude libraries later.

- During the cluster selection procedure, we keep track of which apps have been used to generate candidate payload fingerprints in previously selected clusters, and consider already selected apps as "inactive". We update the remaining payload clusters by removing candidate fingerprint entries that are associated with "inactive" apps. Skipping such fingerprints forces us to extract one version of malicious payload from each app, and helps us to reduce the probability of extracting other payload that are mainly related to less popular libraries or other randomly shared code segments.

- We omit the payload cluster if the corresponding cluster fingerprint contains minimum (e.g. $k = 70$) number of 1-bits, this means the extracted code segments are very small. It forces the algorithm to break current large payload cluster into smaller size clusters, and prevent the situation in which multiple malware families are clustered together and we only extract the very small size shared code between different malware families. We also skip the candidate payload cluster if it becomes empty after the updates in last step or if the number of payload fingerprint entries (e.g., $l = 1$) are too small. This is because clusters with only one candidate payload entry could not provide additional payload sharing information and are more likely to be less popular libraries or other randomly shared code snippets, we consider malware samples associated with such payload clusters as unclustered.

It is worth to mention that the malicious payload mining procedure employs several thresholds, such as clustering threshold θ, the minimum payload cluster size l, minimum cluster fingerprint 1-bits k. The specific values for those thresholds are a trade-off between false positives and false negatives of finding meaningful (e.g., large enough in size) and correct (e.g., non-random shared code snippets) version of shared payload, and we opted for accuracy of the final results by allowing unclustered samples. The optimal values are obtained from the library extraction procedure when applied on benign apps.

Generally speaking, the shared payload between Android samples may be library code, malicious payload, copy pasted code segments, or other randomly shared code segments, .etc. The above malicious payload mining strategy enables us to select candidate payload groups that are most likely to be malicious.

3.2 Optimize Fuzzy Hashing based Clustering

According to the previously discussed malicious payload mining procedure, we will generate $\frac{n \times (n-1)}{2}$ versions of candidate payload fingerprints given n malware samples, and the hierarchical clustering algorithm has a quadratic complexity with respect to the number of analyzing targets. Due to the overall quartic complexity of the algorithm, directly using it to analyze large number of samples becomes a time-consuming task. Therefore, we further develop three methods to improve the scalability of the malicious payload mining procedure, and hereafter refer them as Opt-1, Opt-2, and Opt-3.

3.2.1 Opt-1*: Reduce Unnecessary Computation*

With hierarchical agglomerative clustering, each object starts in its own cluster, and two objects are compared and merged into one cluster if their similarity score is above predefined threshold θ.

We reduce unnecessary pairwise comparisons by exploiting the Jaccard similarity function and clustering analysis procedure at the same time. The Jaccard similarity is defined as the size of intersection divided by the size of the union of two sets. Suppose that we have two sets A and B, and they have n and $k \times n$ *($k \geqslant 1$)* elements respectively. Since A and B have different number of elements, the Jaccard similarity between two sets is maximized when all the elements of smaller set A are contained in set B, where the intersection size becomes n and the union size becomes $k \times n$. The maximum Jaccard similarity is therefore $\frac{1}{k}$. For example, if the size ratio of two sets is greater than 2 or smaller than $\frac{1}{2}$, then their Jaccard similarity score will be lower than 0.5; thus we do not need to compare them for clustering with threshold 0.5.

Since our fuzzy hashing fingerprint is essentially a set where each 1-bit represents a particular n-gram feature, we determine that we only need to compare fingerprint pairs whose 1-bit size ratio is between θ and $\frac{1}{\theta}$ when clustering with threshold θ, because the Jaccard similarity score will be lower than θ for the other pairs. Based on the above reasoning, Opt-1 will not impact accuracy at all.

3.2.2 Opt-2*: Optimize Each Pairwise Computation*

Another approach to speedup the overall fuzzy hashing based clustering process is to optimize each pairwise computation. Broder proposed minHash [2] to quickly estimate the Jaccard similarity of two sets without explicitly computing the intersection and the union of two sets. By considering our bit-vector fingerprint as a set, we apply minHash to further transform a large fingerprint into a smaller size signature, and calculate the similarity of minHash signatures to estimate the Jaccard similarity of the original fingerprints.

To apply minHash, we define a minHash function value of our bit-vector fingerprint $h(FP)$ to be the first non-zero bit index on a randomly permutated bits order of the fingerprint. We then apply the same minHash function to two fingerprint FP_a and FP_b. We have the same value when the two fingerprints have the same bit index set to 1 and the bit index is the first non-zero bit for the current permutation. Since the probability that the firstly encountered bit is a non-zero bit for both FP_a and FP_b is the same as $Similarity(FP_a, FP_b)$ [19], we use the probability to estimate the original Jaccard similarity $Pr[h(FP_a) = h(FP_b)] \approx Similarity(FP_a, FP_b)$.

The probability estimation becomes more accurate if more independent minHash functions are used together. Formally, we define a minHash signature $sig(FP)$ to be a set of k minHash function values extracted from k round of random permutations over the fingerprint, and represent it as follows: $sig(FP) = [h_1(FP), h_2(FP), ..., h_k(FP)]$. We denote the similarity of two minHash signatures as the ratio of equal elements between $sig(FP_a)$ and $sig(FP_b)$.

Instead of maintaining k random permutations over the bit-vector, we follow a common practice for using minHash technique and use k different hash functions to simulate k random permutations, where each hash function maps a bit index to a value. In order to create k hash functions, we first generate k random numbers, then use FNV [10] hash algorithm to produce a basic hash output for each bit index, and finally apply XOR operation between each random number and the hash output to get the k hash outputs. For each hash function, we select the smallest hash value (to simulate the first non-zero bit index) over all of the bit indexes of the fingerprint as the final hash output. Note that the FNV hash value and the k random numbers are all 32 bits unsigned integers, and they can be used to safely simulate random permutation over 512MB bit-vector fingerprint. In practice, the k value usually needs to be larger than 100 to generate good enough results [19]. We set k to be 256 in our experiments, and thus convert each 256KB fuzzy hashing fingerprint into a 1KB minHash signature.

In order to evaluate the potential impact of Opt-2 on accuracy, we conduct two experiments on the smallest 50 malware families[3]: one experiment (Exp-1) using Opt-1 only, and another experiment (Exp-2) using both Opt-1 and Opt-2. We used the clustering output from Exp-1 as a reference, and measured the precision and recall of the clustering output from Exp-2. The precision and recall indicate how similar the two experiments results are, and are used to check the impact on accuracy brought by Opt-2. Our experiments showed that on average Exp-2 took less than 83% time to complete compared to Exp-1 for each family, and the average precision and recall for the analyzed 50 families were 0.993 and 0.986. This demonstrates that Opt-2 has almost zero accuracy penalty.

3.2.3 Opt-3: *Employ approximate clustering*

The previous two speed improvements are still not sufficient for using the algorithm to analyze large scale malware samples. For instance, when analyzing with 2,121 samples, the algorithm will create 2,248,260 shared payloads, and it results in approximately 2.5×10^{12} pairwise comparison. Even 1% of the total comparison (after the previous two speed optimizations) still takes lots of computation resources. To resolve the scalability issue for a large dataset input, we further develop prototype-based clustering technique to achieve approximate clustering.

Specifically, we divide the target samples into small size (e.g., 150) groups. For each group, we apply hierarchical clustering analysis on the shared payload within the group, and create a prototype fingerprint for each payload cluster by applying intersection analysis (to obtain all the shared 1-bit) among the payload fingerprints in each cluster. We then conduct hierarchical clustering analysis on all the collected prototype fingerprints. In this way, we represent a group of similar payload fingerprints with a single prototype fingerprint, and the algorithm proceeds with approximate clustering analysis using the prototype fingerprints instead of the original payload fingerprints. Note that Opt-1 and Opt-2 are complementary to the prototype-based clustering analysis.

We design two experiments to evaluate the impact of Opt-3 on accuracy: one experiment (Exp-3) using Opt-1 and Opt-2, and another experiment (Exp-4) using Opt-1, Opt-2, and Opt-3. Due to the quartic complexity of the original algorithm, the overall analysis (using Opt-1 and Opt-2 only) will get dramatically slower for analyzing larger number of malware samples, and it quickly gets worse when the target samples are more than 2000. For instance, we found it takes about one day to analyze 2000 samples and more than ten days to analyze 3000 samples for Exp-3. In order to conduct the evaluation within reasonable amount of time, we randomly select 70% of labeled samples from the largest 4 malware families and conduct the two experiments for each family. We used the clustering output generated by Exp-3 as reference, and measured the precision and recall of the clustering output generated by Exp-4 to evaluate the accuracy impact brought by Opt-3.

Table 1: Accuracy impact of prototype based clustering

Family Name	Analyzing Size	Time taken		Precision	Recall
		Exp-3	Exp-4		
`Plankton`	1208	20h42min	54min	0.986	0.970
`Adwo`	1891	22h11min	1h8min	0.945	0.921
`Fakeinst`	2197	24h35min	32min	0.932	0.915
`Dowgin`	2296	28h24min	1h16min	0.957	0.923

The results for analyzing the largest four families were summarized in Table 1. Clustering quality of Exp-4 was comparable (precision of 0.932 and recall of 0.915 at the lowest) to Exp-3, and it significantly reduced the analysis time. These optimizations make it feasible to apply our algorithm to analyze a bigger scale of malware families while providing a desirable trade-off option between speed and accuracy.

4. EXPERIMENTS

In this section, we describe the data preparation procedure, and report malware clustering results and key findings of our experiments. All our experiments were conducted on a Ubuntu 14.04 Server, which was equipped with two 2.4GHz Intel Xeon E5-2695 v2 processors and 256GB memory.

4.1 Data Preparation

We obtained a large collection of potentially malicious Android apps (ranging from 2010 to 2016) from various sources, include Google Play, VirusShare and third party security companies. During our analysis, we observed that some labeled-malicious apps in our datasets were actually benign while some labeled-benign Google Play apps were malicious. In order to prepare "cleaner" datasets, we queried the collected apps against VirusTotal, and used the scanning results to filter out the potentially ambiguous apps. Malware samples submitted to VirusTotal are scanned by more than 50 anti-virus products. We assumed the detection techniques used by different anti-virus products were independent and we can prepare the malware dataset according to multiple anti-virus scanning results so that it is not biased by a

[3]we select those families since their maximum family size is under 100 and all the experiments for those families can be finished within 1 hour

particular anti-virus scanner.

From VirusTotal scanning results, previous research [23] showed that different number of malicious flags could be an indicator of different malware qualities, e.g., a malware sample detected by only one scanner was likely to be a false positive of the particular product. Since it is desirable to have reliable datasets to build analysis model and measure the performance, we restricted our malicious dataset to the malware samples that were detected by at least 25 different anti-virus scanners for conservativeness.

4.1.1 Clearly Labeled Malware

We used VirusTotal reports to derive the malware family labels. Specifically, we first tokenized VirusTotal scanning results and extracted English keywords, and then recorded the total count of each keyword. We converted keywords to lowercase except the first letter to normalize several variations. Generic keywords (e.g., `Virus`, `Trojan`, and `Malicious`) were ignored because they were too generic to be used as representative family names. Due to the inconsistent labeling across different anti-virus products, we also measured the edit distance between the keywords and aggregated the similar ones into one keyword. For example, `Nickyspy`, `Nickspy`, `Nicky`, and `Nickibot` were all consolidated into `Nickispy`. Note that the edit distance measurement serves as enhanced normalization for identifying various malware family aliases.

Table 2: Malware samples with dominant keywords

App ID	Keywords and counts		
M1	Plankton (10)	Startapp (4)	Apperhand (4)
M2	Wapsx (15)	Dowgin (5)	Frupi (2)
M3	Youmi (12)	Wooboo (4)	Adrads (2)

To assign a family name for each sample, we selected the dominant keywords from the scanning results. In particular, we considered a keyword as a dominant keyword if it satisfied the following two conditions: (a) the count of the keyword was larger than a predefined threshold t (e.g., t=10), and (b) the count of the most popular keyword was at least twice larger than the counts of any other keywords. Table 2 shows three examples of dominant keywords. We excluded samples that were not clearly labeled with a dominant keyword since the main objective was to prepare a reliable ground truth dataset with the most consistent family names. We also collected a certain amount of relatively recent malicious Android malware samples, such as `SlemBunk`, `Triada` and `RuMMS`. However, we observed that those recent apps usually did not have enough consensus labels across different anti-virus scanning results so that they were not included in the labeled dataset.

In summary, we collected 19,738 labeled malware samples from 68 different families, and the detailed breakup of the malware samples is shown in Table 3.

4.1.2 Candidate Repackaging App Pair

We also collected 945,786 benign apps that were scanned in VirusTotal and had no malicious flags as of April 18, 2016; and the benign apps were used to identify candidate repackaging app pairs. We designed an alternative approach to extract malicious payloads from the candidate repackaging app pairs, which then were used to verify the malicious payload extracted by the clustering analysis based approach.

Table 3: Clearly Labeled Malware Families

Name	Size	Name	Size	Name	Size
Dowgin	3280	Fakeinst	3138	Adwo	2702
Plankton	1725	Wapsx	1668	Mecor	1604
Kuguo	1167	Youmi	790	Droidkungfu	561
Mseg	245	Boqx	214	Airpush	183
Smskey	166	Kmin	158	Minimob	145
Gumen	145	Basebridge	144	Gingermaster	122
Appquanta	93	Geinimi	86	Mobidash	83
Kyview	80	Pjapps	75	Bankun	70
Nandrobox	65	Clicker	58	Golddream	54
Androrat	49	Erop	48	Andup	48
Boxer	44	Ksapp	39	Yzhc	37
Mtk	35	Adflex	32	Fakeplayer	31
Adrd	30	Zitmo	29	Viser	26
Fakedoc	26	Stealer	25	Updtkiller	24
Vidro	23	Winge	19	Penetho	29
Mobiletx	19	Moavt	19	Tekwon	18
Jsmshider	18	Cova	17	Badao	17
Spambot	16	Fjcon	16	Faketimer	16
Bgserv	16	Mmarketpay	15	Koomer	15
Vmvol	13	Opfake	13	Nickispy	12
Uuserv	12	Svpeng	12	Steek	12
Spybubble	12	Fakeangry	12	Utchi	11
Ramnit	11	Lien	11		

Given a repackaging app pair (an original benign app and a repackaged malicious app), we applied the library code removal technique to exclude the original benign app code from malware and extract the malicious payload. The malicious payload extracted in this step served as an alternative source for malicious payload verification. To identify candidate repackaging app pairs, we compared the labeled malicious dataset against entire benign datasets, and identified the candidate repackaging app pairs through the following procedure.

- We compared each malware fingerprint against every benign fingerprint, and filtered out dissimilar app pairs based on the number of set bits of both fingerprints by using the optimization `Opt-1` in Section 3.2. Specifically, we ignored app pairs that had a size ratio larger than 2, as their similarity would be lower than 0.5, and thus less likely to be the repackaged Android malware and their corresponding repackaging origins.

- In order to prepare more confident repackaging app pairs, we required the collected app pairs to satisfy at least one of the following conditions: (a) the edit distance between the package names of two apps was less than 4; (b) the over similarity of the directory names contained in two apps was larger than 0.5; (c) the over similarity of the file names contained in two apps was larger than 0.5; (d) the over similarity of the top level class name set in two apps was larger than 0.5; (e) the file size ratio of `classes.dex` of two apps was within 2; or (f) the ratio of the number of functions in two apps was within 2.

- Finally, we performed containment analysis to select the candidate repackaging app pairs with the containment threshold of 0.9. The rationale behind using containment analysis as a decision function was that removing existing code sequences from a benign app was less likely to happen than adding new code sequences, as the existing code base might have code dependency requirements.

Note that we deliberately tuned the parameters towards selecting the app pairs that were highly likely to be the repackaged malware and the repackaging origins. Within the labeled malware dataset, we eventually found 516 malicious samples, each of which had at least one candidate repackaging origin.

4.2 Clustering Results

4.2.1 *Alternative Malicious Payload Verification*

In this work, we achieve malware sample clustering by checking if multiple samples share similar version of malicious payloads. The quality of the extracted malicious payload will largely impact the sample clustering results. We design a new way to precisely extract malicious payloads from candidate repackaging app pairs to verify our malicious payload mining.

Specifically, we extracted the malicious payload from the malware sample of a candidate repackaging app pair by using the library code removal technique to exclude the original benign app code from the repackaged malicious app.

We create a normal bit-vector fuzzy hashing fingerprint for the malware sample by mapping the extracted n-gram features into 1-bits, and map the n-gram features that extracted from benign app into 0-bits for the benign app fingerprint generation. By applying intersection analysis on the two fingerprints, we automatically exclude the benign app mapped 1-bits from malicious app fingerprint, thus locate the corresponding malicious payload. For simplicity, we call the "clustering analysis" based malicious payload extraction as the main payload extraction approach, and call the new method described in this section as an alternative payload extraction approach.

For the 516 labeled malware samples which had candidate repackaging origins, we found 34 of them were not been clustered due to falling into payload clusters that had only one entry. We compared the payload extraction results of the main approach against the alternative approach for the rest 482 malware samples. Since the main approach extracts malicious payloads through two rounds of intersection analysis (e.g., sample intersection analysis, and candidate payload intersection analysis), the extracted payloads are more precise and contain little or no library code; while the alternative approach extracts malicious payloads by removing popular library code and original benign app code, and the extracted payloads may contain certain less popular library code.

For these 482 malware samples, the experiment result showed that on average 91.5% of the malicious payloads extracted by the main approach were also extracted by the alternative approach, which means both approaches almost all get the same core malicious payload from each malware sample, and the malicious payload extracted by the main approach has higher quality and indeed contain the majority of core malicious payload.

4.2.2 *Clustering Results for Individual Families*

As discussed earlier, we can derive malware family labels from VirusTotal scanning results, but we can not directly get the subversion information. In practice, malware samples labeled with the same family name could contain completely different versions of malicious payloads. For example, the `Cova` malware family has two main versions of malicious payloads and they use almost completely different code bases. In order to obtain specific malware family labels and version information, it is desirable to separate them into different clusters.

In order to get the subversion information for each malware family, we conducted malware sample clustering analysis within each malware family. The overall sample clustering results for each malware families is described in Table 4. As shown in the table, we obtained 260 sample clusters in total for the labeled malware samples from 68 families, each sample cluster corresponded to one version of malicious payloads. Out of 19,738 labeled malware samples, 634 (3.2%) samples were skipped by the clustering algorithm. The malware family that contained the largest number of sample clusters was `Fakeinst`, for which we extracted 42 versions of malicious payloads.

Table 4: Clustering Results for Individual Families

Family Name	Clusters Count	Skipped Samples	Family Name	Clusters Count	Skipped Samples
Dowgin	23	134	Fakeinst	42	105
Adwo	33	100	Plankton	6	20
Wapsx	11	49	Mecor	2	0
Kuguo	4	19	Youmi	18	75
Droidkungfu	7	24	Mseg	2	12
Boqx	3	2	Airpush	4	3
Smskey	3	1	Kmin	3	0
Minimob	2	1	Gumen	1	0
Basebridge	6	4	Gingermaster	8	3
Appquanta	1	0	Geinimi	1	1
Mobidash	2	2	Kyview	3	4
Pjapps	2	3	Bankun	3	3
Nandrobox	2	0	Clicker	2	2
Golddream	2	2	Androrat	1	3
Erop	1	2	Andup	5	1
Boxer	1	0	Ksapp	2	1
Yzhc	2	4	Mtk	3	3
Adflex	1	1	Fakeplayer	3	10
Adrd	2	2	Zitmo	2	0
Viser	2	2	Fakedoc	1	4
Stealer	1	0	Updtkiller	1	0
Vidro	1	0	Winge	2	2
Penetho	1	1	Mobiletx	1	2
Moavt	2	0	Tekwon	1	2
Jsmshider	1	1	Cova	2	0
Badao	1	0	Spambot	2	2
Fjcon	1	0	Faketimer	2	4
Bgserv	2	0	Mmarketpay	1	1
Koomer	1	0	Vmvol	1	0
Opfake	2	3	Nickispy	1	2
Uuserv	1	1	Svpeng	1	0
Steek	1	0	Spybubble	1	2
Fakeangry	1	3	Utchi	1	2
Ramnit	2	4	Lien	2	0

In practice, the clustering analysis procedure conducted on top of individual malware families can be considered as a training procedure for preparing the legitimate library list. Note that such training is not the same as the conventional training process often used in machine learning, in which an analysis model is built from the training dataset and used for future detection.

4.2.3 *Clustering Results for Multiple Families*

In order to evaluate the practical usage of clustering analysis procedure, we performed malware sample clustering analysis across multiple malware families. We considered the VirusTotal family labels together with the manually verified subversion information as ground truth datasets since VirusTotal scanning results did not contain subversion information. Since different versions of malicious payloads usually contained few shared code segments (otherwise, they would be grouped into the same payload cluster), we considered the samples with different subversion as different families, such as `Droidkunfu-1`, `Droidkungfu-2`, `Droidkungfu-3`, and so on. As a result, we had 260 malware families that were manually verified as shown in Table 4.

We prepared 10 experiment datasets for evaluation where each dataset contained 30 families. For each dataset, we randomly selected 30 families from the entire ground truth dataset, then mixed the corresponding samples together. The resulting datasets had different overall sizes as each individual family had different number of samples. We used the classical precision and recall [1, 13, 14, 15, 17, 18, 20, 22, 26, 27, 28] measurements to evaluate the accuracy of clustering results.

Table 5: Clustering Results for Multiple Families

Datasets	Samples Count	Resulting Clusters	Precision	Recall
D1	1064	33	0.977	0.972
D2	1462	27	0.987	0.964
D3	1708	29	0.985	0.978
D4	1039	31	0.971	0.960
D5	2277	29	0.988	0.989
D6	1066	30	0.971	0.919
D7	1256	29	0.985	0.981
D8	1680	29	0.985	0.980
D9	2074	31	0.996	0.858
D10	1612	31	0.992	0.989

The detailed dataset sizes and sample clustering results for multiple malware families are presented in Table 5. On average, the sample clustering algorithm separated the input malware samples into 29.9 clusters, which was extremely close to the reference set (i.e., 30 families). For the 10 experiment datasets, the clustering algorithm achieved average precision of 0.984 and average recall of 0.959. As shown in the table, the worst precision and recall for clustering multiple malware families were 0.971 and 0.858, which suggests that our approach generated very consistent and reliable outputs.

4.3 Key Findings

Significant library code ratio: Library code ratio in Android apps is indeed significant. From our datasets, we found that at least 31.5% of code in every malware was library code, and more than 50% of code was library code in 43% of malware samples. This highlights that existing similarity analysis of Android malware becomes ineffective without considering library code. We also note that more and more malware families inject their malicious payloads under popular library names, and even interpose them as subcomponents of existing libraries. By representing Android apps and legitimate libraries as fuzzy hashing fingerprints, we identify legitimate libraries and exclude them from an app while including bogus libraries for further analysis.

Limited versions of malicious payload: Practical Android malware contains limited version of malicious payload. During our experiments, we acquire 260 versions of malicious payloads from 68 VirusTotal labeled malware families through the malicious payload extraction procedure. Among 68 malware families, 27 families have only one version of malicious payload, and 5 families have more than 10 different versions of malicious payload. For example, `Dowgin` is the largest malware family and has 23 version of malicious payload extracted. For the detailed payload version count information, please refer to Table 4. The above results show that in practice Android malware only contains limited versions of malicious payloads and malware authors often reuse the same version of malicious payload to create new malicious samples. Our Android malware app clustering solution exploits such operational practices to group related malware samples.

Malicious payload under popular namespaces: Malware authors indeed try to hide payload under popular namespaces. We conducted manual analysis on the extracted malicious payloads, and identified that significant amount of Android malware families tried to hide their malicious payload under popular namespaces, such as "com.google" and "com.android". Since these namespaces are the main class names used by Android Open Source Project and Google Mobile Services, such malicious payloads can easily get overlooked during analysis. Besides the above popular namespaces, attackers can also hide malicious payload under third-party library names. For example, `Gumen` malware tried to inject their malicious payload under `com.umeng.adutils`, which looks like a genuine sub-class for the official `Umeng` advertisement library. Our malicious payload extraction method leads us to discover that there are about 29% of malware families disguising their malicious payloads under popular library names. The detailed information for such library name usages are illustrated in Table 6 of Appendix.

5. RELATED WORK

Android malware research has boomed in recent years thanks to the worldwide popularity of the Android platform. In this section, we describe Android application security research work that is most related to our work from two specific perspectives, and discuss the strengths and fundamental limitations of existing research work and the distinctive contributions of our work.

5.1 Android Application Similarity Analysis

It is well-known [32] that significant amount of existing Android malware belongs to repackaged apps. To detect repackaged Android malware, similarity analysis based techniques have been proposed. For example, Juxtapp [12] utilized Dalvik bytecode level n-gram features and feature hashing techniques [17] to represent an app, and focused on detecting code reuse and repackaged malware through app similarity analysis. DroidMoss [31] employed a fuzzy hashing technique to effectively localize and detect the changes by app-repackaging behavior. Androguard [6] leveraged semantic Normalized Compression Distance (NCD) measures to verify the similarity percentage for existing third party apps. AndroSimilar [9] provided a syntactic foot-printing mechanism to find the regions of statistical similarity with known malware to detect unknown zero-day samples. ViewDroid [29] utilized a feature view graph to capture users' navigation behavior across app views, and used a graph similarity algorithm to detect a repackaged app. The combination of n-gram features and feature hashing techniques was studied as the main building block for designing a reliable fuzzy hashing algorithm and conducting efficient and accurate code similarity analysis [20].

In this work, we employ a fuzzy hashing algorithm called nextGen-hash while incorporating the techniques to analyze Android applications at the disassembled bytecode level, and extensively exploit the bit manipulation capability provided by the bit-vector fuzzy hashing fingerprint representation as the technical foundation for subsequent analysis. Even though both Juxtapp [12] and our approach use n-gram features of Dalvik bytecode and feature hashing techniques, the main objectives are different so that the ways of using fingerprints are also different. For example, we directly process Dalvik bytecode sequences within each function and include operand features, such as Java primitive types, for precise similarity analysis (e.g., avoiding overestimating similarity) whereas Juxtapp processes Dalvik bytecode at the basic block level and discard most operands.

Similarity analysis is essential for clustering. Existing Android application similarity analysis techniques were mainly designed to detect repackaged apps [6, 12, 29, 31], and they cannot be directly applied to app clustering due to the challenges discussed earlier. We build our solution based on an efficient fuzzy hashing algorithm, and empirically demonstrate that it provides crucial capabilities for Android malware analysis, such as removing legitimate library code, extracting malicious payloads from Android malware, and performing Android malware clustering.

5.2 Android Malicious Payload Analysis

Malicious payload identification and extraction is essential for Android malware analysis. Zhou and Jiang [32] manually analyzed malicious payloads of Android malware and summarized the findings in the Android Malware Genome project. DroidAnalytics [30] presented a multi-level signature based analytics system to examine and associate repackaged Android malware. MassVet [4] analyzed graph similarity at the function level and extracted the shared non-legitimate functions as malicious payloads through commonality analysis and differential analysis.

In this work, we use a fuzzy hashing fingerprint based approach to extract the core malicious payload from Android malware, and conduct application clustering analysis by checking if the analyzed samples share the same version of malicious payloads. MassVet [4] is close to our work in that both extract malicious payloads from Android malware. However, similar to existing Android malware analysis work [3, 4, 5, 8, 11], MassVet simply used library name based whitelists to ignore popular library code, which can result in the failure of malicious payload extraction, and lead to false negatives in malware detection if malicious payloads are injected into popular library namespaces. In addition, our approach operates at the instruction level while MassVet operates at the function level, and our finer-grained granularity features allow us to precisely identify one version of malicious payload from each Android malware. Precise and robust malicious payload extraction in our approach is a key to provide a more complete view of the malicious payloads.

6. CONCLUSION

In this paper, we provide a practical solution for conducting Android malware clustering analysis based on an efficient fuzzy hashing algorithm. As shown in the paper, our approach can safely remove legitimate library code and automatically extract malicious payload. Particularly, our solution can be used to distinguish an old version library and a bogus library that share the same library name; it can precisely locate the differential part and extract the corresponding malicious code segments. Compared with existing malicious payload extraction system, our approach can extract malicious payload even if they are injected under popular library namespaces or under existing benign functions, and it provides a more complete picture of the whole malicious payload. Unlike traditional clustering techniques which directly examine the overall similarity, we achieve Android malware clustering by checking whether malware samples share the same version of malicious payload, and our experimental results demonstrated that the clustering algorithm can generate consistent and reliable outputs.

References

[1] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, “Scalable, behavior-based malware clustering.” in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[2] A. Z. Broder, “On the resemblance and containment of documents,” in *Compression and Complexity of Sequences 1997. Proceedings.* IEEE, 1997, pp. 21–29.

[3] K. Chen, P. Liu, and Y. Zhang, “Achieving accuracy and scalability simultaneously in detecting application clones on android markets,” in *Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014, pp. 175–186.

[4] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, “Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale,” in *USENIX Security*, vol. 15, 2015.

[5] J. Crussell, C. Gibler, and H. Chen, “Attack of the clones: Detecting cloned applications on android markets,” in *Computer Security–ESORICS 2012.* Springer, 2012, pp. 37–54.

[6] A. Desnos, “Androguard reverse engineering tool,” https://github.com/androguard, 2013.

[7] “Dexdump,” http://developer.android.com/tools/help/index.html, 2015.

[8] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, “An empirical study of cryptographic misuse in android applications,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* ACM, 2013, pp. 73–84.

[9] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, “Androsimilar: robust statistical feature signature for android malware detection,” in *Proceedings of the 6th International Conference on Security of Information and Networks.* ACM, 2013, pp. 152–159.

[10] K.-P. V. Glenn Fowler, Landon Curt Noll, “Fnv hash,” http://www.isthe.com/chongo/tech/comp/fnv/, 2015.

[11] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, “Riskranker: scalable and accurate zero-day android malware detection,” in *Proceedings of the 10th international conference on Mobile systems, applications, and services.* ACM, 2012, pp. 281–294.

[12] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2013, pp. 62–81.

[13] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *ACM Conference on Computer and Communications Security*, 2009.

[14] X. Hu and K. G. Shin, "DUET: integration of dynamic and static analyses for malware clustering with cluster ensembles," in *Annual Computer Security Applications Conference*, 2013.

[15] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna, "A static, packer-agnostic filter to detect similar malware samples," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013.

[16] J. Jang, A. Agrawal, and D. Brumley, "Redebug: finding unpatched code clones in entire os distributions," in *2012 IEEE Symposium on Security and Privacy.* IEEE, 2012, pp. 48–62.

[17] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: feature hashing malware for scalable triage and semantic analysis," in *Proceedings of the 18th ACM conference on Computer and communications security.* ACM, 2011, pp. 309–320.

[18] D. Kirat, L. Nataraj, G. Vigna, and B. S. Manjunath, "SigMal: a static signal processing based malware triage," in *Annual Computer Security Applications Conference*, 2013.

[19] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of massive datasets.* Cambridge University Press, 2014.

[20] Y. Li, S. C. Sundaramurthy, A. G. Bardas, X. Ou, D. Caragea, X. Hu, and J. Jang, "Experimental study of fuzzy hashing in malware clustering analysis," in *Proceedings of the 8th USENIX Conference on Cyber Security Experimentation and Test.* USENIX Association, 2015, pp. 8–8.

[21] R. Perdisci and M. U, "VAMO: towards a fully automated malware clustering validity analysis," in *Annual Computer Security Applications Conference*, 2012.

[22] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.

[23] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara, "Experimental study with real-world data for android app security analysis using machine learning," in *Proceedings of the 31st Annual Computer Security Applications Conference.* ACM, 2015, pp. 81–90.

[24] B. Snell, "Mobile threat report, what´s on the horizon for 2016," http://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf, 2016.

[25] "Virustotal," https://www.virustotal.com, 2015.

[26] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[27] G. Yan, N. Brown, and D. Kong, "Exploring discriminatory features for automated malware classification," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013.

[28] Y. Ye, T. Li, Y. Chen, and Q. Jiang, "Automatic malware categorization using cluster ensemble," in *ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, 2010.

[29] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu, "Viewdroid: towards obfuscation-resilient mobile application repackaging detection," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks.* ACM, 2014, pp. 25–36.

[30] M. Zheng, M. Sun, and J. Lui, "DroidAnalytics: A signature based analytic system to collect, extract, analyze and associate Android malware," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on.* IEEE, 2013, pp. 163–171.

[31] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party Android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy.* ACM, 2012, pp. 317–326.

[32] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on.* IEEE, 2012, pp. 95–109.

7. APPENDIX

Table 6: Malicious payload under popular libraries

Family	Popular Class Names Used
Nickispy	com.google.android.info.SmsInfo com.google.android.service.UploadService
Uuserv	com.uuservice.status.SysCaller.callSilentInstall com.uuservice.status.SilenceTool.MyThread.run
Fjcon	com.android.XWLauncher.CustomShirtcutActivity com.android.XWLauncher.InstallShortcutReceiver
Yzhc	com.android.Base.Tools.replace_name com.android.JawbreakerSuper.Deamon
Gumen	com.umeng.adutils.AdsConnect com.umeng.adutils.SplashActivity
Basebridge	com.android.sf.dna.Collection com.android.battery.a.pa
Spambot	com.android.providers.message.SMSObserver com.android.providers.message.Utils.sendSms
Moavt	com.android.MJSrceen.Activity.BigImageActivity com.android.service.MouaService.InitSms
Zitmo	com.android.security.SecurityService.onStart com.android.smon.SecurityReceiver.sendSMS
Mseg	com.google.vending.CmdReceiver android.ad.appoffer.Copy_2_of_DownloadManager
Droidkungfu	com.google.ssearch.SearchService com.google.update.UpdateService
Dowgin	com.android.qiushui.app.dmc com.android.game.xiaoqiang.jokes.Data9
Fakeinst	com.googleapi.cover.Actor com.android.shine.MainActivity.proglayss_Click
Ksapp	com.google.ads.analytics.Googleplay com.google.ads.analytics.ZipDecryptInputStream
Bankun	com.google.game.store.bean.MyConfig.getMsg com.google.dubest.eight.isAvilible
Pjapps	com.android.MainService.SMSReceiver com.android.main.TANCActivity
Adwo	com.android.mmreader1030 com.google.ads.AdRequest.isTestDevice
Svpeng	com.adobe.flashplayer_.FV.doInBackground com.adobe.flashplayer_.FA.startService
Opfake	com.android.appupdate.UpdateService com.android.system.SurpriseService
Badao	com.google.android.gmses.MyApp com.android.secphone.FileUtil.clearTxt