Tackling Androids Native Library Malware with Robust, Efficient and Accurate Similarity Measures

Anatoli Kalysch
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU), Germany
anatoli.kalysch@fau.de

Oskar Milisterfer
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU), Germany
oskar.milisterfer@gmail.com

Mykolai Protsenko
Fraunhofer Institute for Applied
and Integrated Security
Garching, Germany
mykolai.protsenko@aisec.fraunhofer.de

Tilo Müller
Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU), Germany
tilo.mueller@cs.fau.de

ABSTRACT

Code similarity measures create a comparison metric showing to what degree two code samples have the same functionality, e.g., to statically detect the use of known libraries in binary code. They are both an indispensable part of automated malware analysis, as well as a helper for the detection of plagiarism (IP protection) and the illegal use of open-source libraries in commercial apps. The *centroid similarity metric* extracts control-flow features from binary code and encodes them as geometric structures before comparing them. In our paper, we propose novel improvements to the centroid approach and apply it to the ARM architecture for the first time. We implement our approach as a plug-in for the IDA Pro disassembler and evaluate it regarding efficiency, accuracy and robustness on Android. Based on a dataset of 508,745 APKs, collected from 18 third-party app markets, we achieve a detection rate of 89% for the use of native code libraries, with an FPR of 10.8%. To test the robustness of our approach against the compiler version, optimization level, and other code transformations, we obfuscate and recompile known open-source libraries to evaluate which code transformations are resisted. Based on our results, we discuss how code re-use can be hidden by obfuscation and conclude with possible improvements.

KEYWORDS

Code Similarity, Android Static Analysis, Reverse Engineering

ACM Reference Format:

Anatoli Kalysch, Oskar Milisterfer, Mykolai Protsenko, and Tilo Müller. 2018. Tackling Androids Native Library Malware with Robust, Efficient and Accurate Similarity Measures. In *ARES 2018: International Conference on Availability, Reliability and Security, August 27–30, 2018, Hamburg, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3230833.3232828

ARES 2018, August 27–30, 2018, Hamburg, Germany

ACM ISBN 978-1-4503-6448-5/18/08…$15.00
https://doi.org/10.1145/3230833.3232828

1 INTRODUCTION

Androids' ecosystem is inherently plagued by application clones transgressing on intellectual property rights, and malicious actors using native code to mitigate malware detection and hide payloads more efficiently.

Application clones are motivated by economic gain. "Cloning" in this context refers to the process of unpacking an APK and repackaging it, including possible tampering. Since Android's APK building process allows for fairly easy repackaging, its code and resources can be tampered with before repackaging. Attackers use repackaging to redistribute popular non-free apps for a cheaper price, but also free apps took a hit on their business model. Popular apps financed by ad-revenue can find clones of themselves uploaded by fraudsters with one simple modification: either the Google ads ID is replaced or the advertisements displayed belonged to another beneficiary. Although third-party markets are primarily affected, Google's Play Store, while implementing UI similarity and code similarity detection mechanisms, still struggles with repackaged apps [33, 37].

Particularly repackaged Android apps from third-party markets are often extended with malicious payloads. This is evident through reports from IT-security firms, e.g., Kasperky and McAfee, releasing trends in malware activity and analysis [26, 29]. March 2018 saw the release of Kasperky's yearly Mobile malware evolution report [26], once again reinforcing the impression of high native library usage among all trending types of mobile malware, including rooting malware, ransomware and banking trojans. Even malware performing relatively simple operations, such as premium SMS trojans, started using native code to thwart their detection and prolong their time to be in the wild [29].

These days, most Android reverse engineering and app analysis tools still only support the analysis of Dalvik bytecode, due to most Android apps having their core logic implemented in Java or Kotlin [22]. The need for more efficient and accurate similarity measures, especially for Android native code becomes apparent.

1.1 Contributions

In this paper, we present improvements to the *centroid approach*, a control-flow graph based method for code similarity detection,

and apply it to Android native libraries on the ARM architecture for the first time. With our improvements, implemented as an IDA Pro plug-in, the centroid approach gains higher efficiency while retaining its accuracy level. In detail, our contributions are:

- We propose algorithmic improvements to the computation of centroids reducing the space and improving runtime requirements for the computation.
- The improved algorithm is evaluated with the *Tigress* obfuscator to assess its robustness against automated code transformations, such as compiler versions and optimization levels.
- Using the centroid similarity metric, we investigate to what extent Android developers and malware authors rely on the re-use of native libraries in a large scale study of 508,745 apps.
- Comparing our approach with other implementations for malware detection, which are featured on *VirusTotal*, yields higher accuracy values for our approach detecting nearly twice as many malware samples.
- Our implementation is freely available at https://www1.cs.fau.de/content/centroid as an open-source plug-in for IDA Pro, published under the MIT license.

2 BACKGROUND

This section describes the building blocks necessary to understand the design and implementation of our code similarity detection. Readers familiar with domination relationship computation in CFGs (Section 2.1), database clustering algorithms (Section 2.2), or native C++ libraries on Android (Section 2.3) may safely skip these sections.

2.1 Domination Relationship

Domination is a principle for many code optimizations concerning CFGs. It is deeply rooted in compiler theory and also part of the computation of centroids. Algorithms to determine domination operate on the basis of relationships between the nodes of a CFG. Aho et al. describe the domination relationship between nodes, corresponding to basic blocks in case of a CFG, as follows [1]:

Definition 2.1 (CFG Dominance Relationships). Given a directed graph G incorporating one root, then a node $x \in G$ dominates another node $y \in G$ if and only if every path from the root node to y contains x. x is then called a dominator of y, and a strict dominator if $x! = y$. The dominance relationship is reflexive as well as transitive, and an immediate dominator is a dominator of a node which is next to this node.

Building on the dominance relationship one can define natural loops inside a CFG as edges from the dominator x of y to y as backward edge. Each backward edge defines a natural loop containing x, y and every node dominated by x and from which a path to y exists which does not traverse x [1].

However, naive attempts at computation of dominators, e.g., iterative fix-point algorithms performed rather slow in practical scenarios [10]. This prompted the creation of optimizations that require the introduction of an additional concept of semi-dominators. Semi-dominators form ancestors from the spanning tree T of graph G, and approximate the immediate dominators. They are formally defined as follows [27]:

Definition 2.2 (CFG Semi-Dominators). Given depth first graph node numbers, a path $P = (v_0, v_1, \ldots, v_{k-1}, v_k)$ is a semi-dominator path in the spanning tree T with $v_i > v_k$ for $0 < i < k$. A node's semi-dominator is the minimum of the starting points of all semi-dominators paths to the node.

Semi-dominators can be calculated quickly if done in reverse preorder because semi-dominator candidates of a node then will include the semi-dominators of its immediate CFG-predecessors that are part of the path to the node on the spanning tree T. On the basis of the semi-dominators the immediate dominators are derivable. As the semi-dominator is an ancestor of the node and all nodes on the semi-dominator path between the two are larger than the node, the immediate dominator must be an ancestor of the semi-dominator or the semi-dominator. The node with the smallest semi-dominator on the spanning tree path between the semi-dominator excluded and the node has the same immediate dominator as the node [27].

2.2 Clustering Algorithms

We refer to clustering as an analogy to unsupervised learning with the objective of grouping similar items together such that intra-cluster differences are maximized and inter-cluster differences minimized. The partitioning-based k-Means, k-Medoids and CLARA variants implement heuristics to approximate solutions of corresponding optimization problems for k clusters, and apply metrics like the sum of square distances [30]. These algorithms have in common, that the number of clusters needs to be known prior to the analysis.

Grid-based methods like Sting and Clique [4] define clusters based on densities of cells in a n-dimensional euclidean vector space. Hierarchical clustering in turn [4] defines nested clusters either through division or agglomeration. The more practical agglomerative algorithms initialize each sample as a cluster and then iteratively merge clusters with their nearest neighbor defined by a metric like minimum (single linkage) or maximal sample distances or centroids. This is repeated until a user-defined cluster difference threshold is exceeded and results in a tree of clusters called *dendrogram*.

Clusters can also be defined as connected dense regions [4] in accordance with a density metric. An advantage of this approach is that it can be performed in a single database scan and detect clusters of any shape. "Density-based"clustering algorithms include DENCLUE and DBSCAN [15]. The latter estimates probability density functions and defines clusters by their local maxima. DBSCAN in turn defines dense regions via *core points* with at least “minPts” other samples having a distance smaller or equal epsilon. During the main database scan each "core pointänd its neighbors define a new cluster which is extended by solely such neighbors of each cluster element which are also "core points". This algorithm applied with a “minPts” parameter of 1 results in exactly one dendrogram cut of an agglomerative single linkage clustering.

2.3 Native Libraries on Android

Android makes it quite easy to write a library's logic in C++ and compiling it afterwards through the use of Android Studio's Native

Development Kit (NDK) [17]. While Java libraries packaged alongside the applications code into the *classes.dex* file containing the bytecode application logic, native libraries are stored as additional files under an architecture dependent path. At runtime the functionality of native libraries can be queried for example through the use of the Java Native Interface (JNI). Native code executes with the same permissions as the underlying application and will adhere to Android's sandboxing principle.

Malware usually utilizes native code to thwart analysis. Assembly offers less meta information than Dalvik bytecode, and also the ability to tamper with the process address space comes in handy, as it allows powerful changes to the Android runtime itself, creating major hindrances for static analysis. Cryptographic operations enjoy computational advantages while several Android reverse engineering tools fail disassembling native code which is widely employed by malware, e.g., the DroidKungFu family [21].

Dynamic code loading [28] is often handled through the use of native code as well, and allows the extraction and decryption of network resources or disguised application resources into Dalvik or native code at runtime. Further use cases include spy ware, e.g., CarrierIQ which employed adware and spyware capabilities [18] and rooting exploits [13].

3 DESIGN AND IMPLEMENTATION

This section presents the design and implementation of our approach. We implemented the centroid approach as an IDA Pro 6.8 plug-in, building upon IDA's automated analysis API. Our open-source implementation is available at https://www1.cs.fau.de/content/centroid.

3.1 Centroid Approach

Broadly speaking, the centroid approach operates on a method-level CFG, encoding it to a 3D-CFG and then computing a mass center inside the 3D coordinate system. A 3D-CFG is a 3D vector where each basic block has a unique coordinate. The centroid of a method represents an encoding of this 3D-CFG, by assigning weight to 3D-nodes and computing a mass center accordingly.

3D-CFG. Chen et al. [7] suggest using *sequence*, *selection* and *repetition* to encode a classic CFG into a 3D-CFG due to their importance as basic building blocks of structured programming languages. The x coordinate hereby encodes the sequence number, defining the order in which a CFG node executes. Chen et al. start with a sequence number of one for the entry node of the method and increment the sequence number with every following basic block. If a branch is encountered, the path with the most nodes is sequenced first, and if two paths have the same number of nodes the number of instructions per branch becomes relevant. The representation of selection, or branch statements, is encoded by the y coordinate through the number of outgoing edges. Lastly, the z coordinate represents repetition by encoding the node's loop depth. The resulting structure consists of nodes or basic blocks connected by directed vectors in a 3D space.

In the 3D space an additional dimension can be introduced by including a weight π for the nodes of a 3D-CFG. In the model suggested by Chen et al. the weight corresponds to the number of statements in a node. Differing encoding schemes are possible as well, e.g., giving additional weight to invocation instructions.

Centroid Computation Theory. Chen et al. define a centroid of a 3D-CFG as a vector $\vec{c} =< c_x, c_y, c_z, \pi >$, with

$$c_x = \frac{\sum_{e(p,q)\in 3D-CFG}(\pi_p x_p + \pi_q x_q)}{\pi},$$

and c_y and c_z accordingly. The π coordinate is encoded as $\pi = \sum_{e(p,q)\in 3D-CFG}(\pi_p + \pi_q)$ where $e(p,q)$ refers to an edge in the 3D-CFG, which connects the two nodes p and q. $< x_p, y_p, z_p >$ encodes p's coordinates while π_p represents the number of statements in p.The resulting centroid of a method hence can be described as the mass center of the methods' 3D-CFG.

Chen et al. showed centroids to have monotonicity properties, ensuring that equal methods are always mapped to the same centroid and minor changes inside a method will only incur a minor effect on the methods' centroid. This is particularly useful since centroids are sortable, thereby enabling a faster method comparison after computation.

Centroid Similarity. The comparison of two centroids is performed through the computation of the Centroid Difference Degree (CDD), a normalized distance for each dimension, as follows [7]:

Definition 3.1 (Centroid Difference Degree). Given two centroids, $\vec{c}$ and $\vec{d}$, the CDD is computed as

$$CDD(\vec{c},\vec{d}) = \max\left(\frac{|c_x - d_x|}{c_x + d_x}, \frac{|c_y - d_y|}{c_y + d_y}, \frac{|c_z - d_z|}{c_z + d_z}, \frac{|\pi_c - \pi_d|}{\pi_c + \pi_d}\right).$$

3.2 Algorithmic Improvements

Our improvements to the centroid computation build heavily upon dominator computations and graph optimizations, as described in section 2.1. The computation of a node's z coordinates requires knowledge of the current loop depth. Compared with the x and y coordinates, the z coordinate is the most expensive in terms of computation resources. However, if the dominance relationships inside a CFG are known this computation can be handled a lot more efficiently.

Dominance Computation Algorithms. The most straightforward approach to calculate dominance relationships is an iterative fixpoint algorithm, which is the approach of Chen et al. [7]. Those algorithms are based on the observation that the dominator of a node must also dominate all predecessors of the node. The dominator set of the root is initialized to the root itself and the dominator sets of all other CFG nodes are initialized to the whole set of nodes. Thereafter iterations are performed in which each CFG node is visited after all of its predecessor have already been processed. The dominator set of each node is hereby set to the intersection of its previous set and the dominator sets of all predecessors. The algorithm stops when no single dominator set changes during one iteration.

Lengauer et al. published the Lengauer-Tarjan algorithm based on semi-dominators, which is still today considered as one of the fastest [27]. The Lengauer-Tarjan algorithm keeps track of candidate semi-dominator paths via a forest which is initialized as

Algorithm	libwilhelm.so	libjnlua5.1.so	libcore.so
Original Fixpoint	310	107.3	243.6
Boost Len.-Tar.	504.3	179	188.3
Custom Len.-Tar.	285.3	109.3	134.3
Cooper Fixpoint	256.6	94	115.6

Table 1: Runtimes in seconds of the centroid calculations for the ARM 32-bit libraries `libwilhelm.so`, `libjnlua5.1.so` and `libcore.so` with different dominance algorithms.

singletons for each node, and the node semi-dominators are initialized as the nodes themselves. After a semi-dominator of a node has been calculated, the node is linked to its parent in the spanning tree. This allows to determine semi-dominators as the minimum of those of the nodes returned by the fast *eval(v)* function on all immediate predecessors, which returns v if v is a root in the forest and the node with the smallest semi-dominator out of v and its ancestors excluding the roots in the forest otherwise. Thereby path compression is done what means that it is checked only once if a direct ancestor in the forest has a smaller semi-dominator and thereafter this ancestor is skipped.

The earlier described fixpoint algorithm can be accelerated via data structures that allow for faster intersections like proposed by Cooper et al. [10]. A single array can encode an immediate dominator tree, wherein per-node elements hold the indices of the respective immediate dominators and the entries are ordered by node postorder numbers. A fast algorithm thereon first calculates the postorder numbers and initializes the immediate dominator of the root to itself and all others to undefined. Then it iteratively performs intersections on the immediate dominator tree in reverse postorder. A new immediate dominator of each node is set to the one of an already processed predecessor, which has to exist because of the reverse postorder processing. This new entry is intersected with the entries of all other predecessors by a procedure with two pointers initialized to point to the two immediate dominators of the respective nodes [10].

To improve the runtime of the centroid computation, we decided to compare the prominent algorithms for dominance computation through runtime measurements. We selected three native Android libraries with differing sizes and different amounts of functions, namely `libwilhelm.so` with 164kB and 1236 functions (multimedia processing), `libjnlua5.1.so` with 46kB and 393 functions (lua language interpreter), and `libcore.so` with 42kB and 46 functions (Adobe AIR). For each library, the centroids were calculated several times, 10k times for `libwilhelm.so` and 15k times for the others. This procedure was first executed with the original fixpoint algorithm and then repeated with the Boost graph Lengauer-Tarjan algorithm, a self-programmed custom version of Lengauer-Tarjan, and Cooper's algorithm. The mean for each algorithm and library is shown in table 1. Overall the Cooper algorithm clearly performed best, prompting the decision to implement a variant of the Cooper algorithm.

Coordinate Calculations. With the algorithmic improvements introduced above, we compute the actual centroids. Aside from the 3D-CFG computation, the π coordinate needs to be computed for the centroid generation. Extending the explanation given in Section 3.1, we compute π's weight depending on the instructions inside a node and add additionally the number of `call` statements inside the basic block, giving more weight to function calls. This requires the instructions of a node to be known prior to the computation of the π coordinate.

Our implementation is implemented as a superordinate depth first search (DFS) [36]. In the beginning of each processing step, the instructions of a node are counted and scanned for call instructions and the outgoing edges for this node are retrieved, which correspond to the y coordinate.

To compute the x coordinate we perform a recursive node enumeration DFS, starting from the branch start nodes and ending at already visited nodes or the branching postdominator. This enables us to count the basic block instructions during the DFS visit to the node, and ensures the correctness of the sequence numbers. This increases the accuracy of our fingerprints, because the more differing two CFG branches are, the more their node's sequence numbers will differ.

Once dominance relationships are established, the calculations of 3D-CFG z coordinates (node loop depth) are possible. Those were implemented to be done in the main DFS as work between the visits of the successors of a node:The dominators of each node are efficiently searched for a successor via the immediate dominator tree array and if one successor dominates the node a backward edge is identified. The corresponding natural loop's nodes are then traversed via a DFS starting at the dominated node and stopping at the dominating successor. For each thus visited node's adjacent edges the π coordinates of the node are accumulated to the enumerator of the z coordinate. Two loops with the same loop head are not considered to be nested, thus already identified loop heads are memorized for each node. When the sums in the depth first search are calculated, the final divisions to obtain centroids coordinates are performed.

3.3 Implementation and Efficiency Considerations

Computational and algorithmic improvements aside, certain runtime considerations can be helpful in either improving the efficiency of the system as a whole or preventing unnecessary overhead during the initial computation of centroids or the later comparison phase. These runtime improvements can save considerable amounts of resources especially if applied to large datasets.

One such consideration posed the choice of database. The plug-in supports both, the relational database MySQL and the NoSQL solution MongoDB. To find an optimal solution we compared the runtime characteristics of both, MongoDB and MySQL, with schemes for both designed for maximum space efficiency. At first centroid fingerprints of 40 libraries with a total size of about 94 MB were inserted. Then reading data of centroids libraries with a total of about 500,000 functions was measured directly after system reboots to avoid memory caching-based result falsifications. Results showed

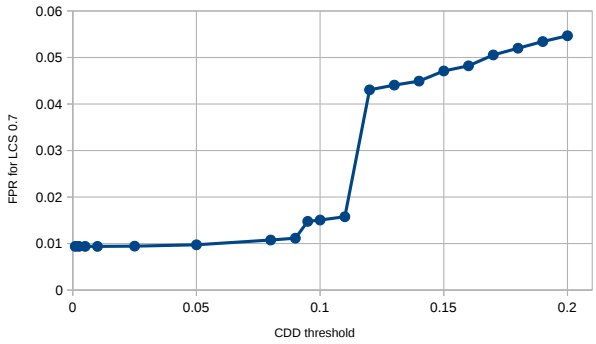

Figure 1: LCS-based FPR pre-analysis of centroids-based library variant detection with an LCS threshold of 0.7.

significantly faster insert and read performance for MongoDB and thus prompted the decision to build a MongoDB-based distributed database.

Fingerprint-based Parallel Clustering. A clustering of the Android native library centroids DB was intended for an analysis of fingerprint accuracy at a large scale and to investigate detection of malware samples in the database. After careful considerations we decided upon DBSCAN due to its ability to detect clusters with any shape and its more efficient calculations of one dendogram cut. The DBSCAN algorithm was parallelized via the implementation of a master-slave scheme, under which each thread processes preferably similar sample subsets and memorizes other partition's neighbors of an own core point by putting both in a cluster-specific list.

For fast CDD derivation complying to the definition of Chen et al.[7], the coordinate pairs were checked for dissimilarity in the order of the presumably most differing ones. This enables continuation to the next comparison function after each dissimilarity threshold - exceeding coordinate pair. An efficient coordinate order was derived from two million non-zero centroid comparisons on a test set which contained unrelated libraries and library versions. The results showed the largest differences for the π-coordinates, followed by the x,y, and z coordinate pairs in that order.

With respect to the applied centroids-based native library similarity measurement, the following library similarity degree (LSD) was used:

$$LSD(ncs_1, ncs_2) = \frac{|ncs_{1,2}|}{|ncs_2|},$$

where $ncs_{1,2}$ is the subset of the non-zero centroids set of the smaller of the two non-zero centroids library sets (any of the two if the sets are equal-sized) for which a similar entry in the second set exists and ncs_2 is the larger of the two sets (any of the two if the sets are equal-sized).

A fine tuning of the CDD and LSD were additionally performed. A longest common subsequence preanalysis yielded insights, that a CDD-threshold of 0.05 or less is needed for a FPR of less than 1% for function similarity. Then a library version detection analysis, which constituted a centroids accuracy investigation itself, lead to the decision to apply the CDD/LSD threshold-pair 0.01/0.7. Contrary to other pairs this CDD/LSD threshold-pair had not a single false positive on unrelated native libraries greater or equal 100 functions. While it performed worse at library version detection with a detection rate of only 54.6%, the minimal FPR was deemed more important. Other promising CDD/LSD threshold pairs included 0.01/0.5, 0.01/0.6 and 0.01/0.7 with FPRs of 0.008, 0.004 and 0.000 respectively, which classified 70.1%, 69.1% and 54.6% of the versions as variants. Figure 1 shows the FPR of a preanalysis of different varying CDDs for an LSD of 0.7.

Despite sophisticated tuning the clustering had to be constrained to native library pregroups for performance reasons. Test pregroups with certain numbers of libraries and average number of functions were runtime-measured to derive the expected runtime by taking into account a roughly quadratic increase by both number of samples and functions, due to the respective algorithmic properties.

4 EVALUATION

To evaluate the centroid approach for its effectiveness on native libraries, we computed the centroid fingerprints from a large data set of Android apps and stored them inside a database. For a graphical overview of the approach refer to figure 2. The hardware setup consisted of four workstations with a distributed MongoDB centroid database. Two of the workstations were Intel Core i7 (8 cores) with 16GB DDR3 RAM each and the other two were Intel Core i5 (4 cores) with 16 and 12GB DDR3 RAM respectively. Each workstation featured 1.5 TB hard drives for the distributed DB and the required software stack for similarity computations. To avoid overhead by DBMS-based sharding, app sharding via the number of native library functions was implemented, supporting a potential cluster pregrouping by the same key.

The app dataset was created from 18 largest third party app stores, Androidsapkfree, anruan, apkfiles, apkmirrordownload, apkpure, appsapk, aptoide, eoemarket, fdroid, freewarelovers, hiapk, mobileapkworld, mumayi, nduo, play_mob, shoujibaidu, slideme, up2down [23]. We used app store crawlers that download the most popular apps first to access the most popular apps [23].

With more than 2.7 TB source APK files, the batched DB build took the 24 CPU cores approximately 5 weeks and resulted in a DB containing data of 508,745 Android native libraries and 2,346,005,582 functions. After computation we stipulated about 200 pregroups in the database to increase centroid comparison efficiency. Pregroups represent a form of preclustering according to a functions size and ensue that centroids are not compared to each other if there is a significant size difference in the underlying functions.

4.1 Approach Accuracy

We used the detection of similar ARM 32-bit native libraries inside the evaluation dataset as a criterion for accuracy.

Library Versions. The comparisons of 1,500 unrelated library pairs served to determine FPRs. Afterwards, the inclusion of native library versions with differing ARM ABIs enabled additional investigation of the centroids fingerprints capabilities in identification of native library variants with different ABIs. The CDD threshold 0.01 showed FPRs of less than 1% for LSD thresholds over 0.4 and no single false positive for LSD thresholds greater equal 0.7. Noteworthy is the significantly worse performance for small libraries

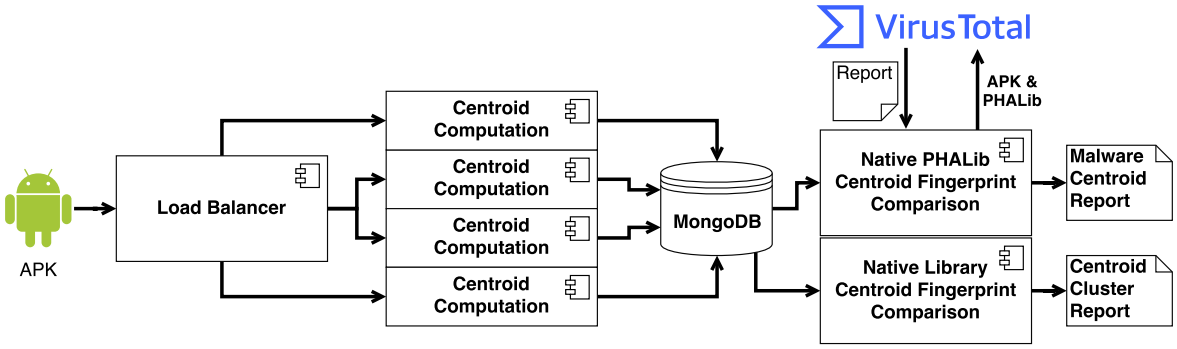

Figure 2: Overview of the evaluation system. To evaluate the accuracy and efficiency of the approach a fingerprint database of available samples from third-party markets was computed. Section 4 describes the setup and result gained by analyzing the resulting fingerprints and the respective clusters. Section 5 covers insights into the distribution of potentially harmful application libraries (PHALib) throughout the dataset.

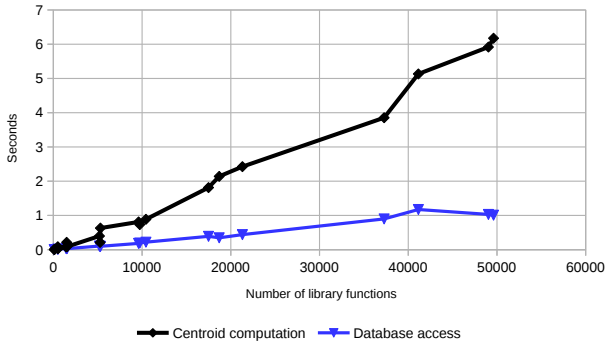

Figure 3: Time comparison of the centroid fingerprint computation and the DB interaction for libraries varying sizes.

containing less than 100 functions. For them the FPRs were 10% and more.

Database Clustering. The clustering of the native library centroids database enabled an analysis of the fingerprint's accuracy up to large scales. For similarity measurements we used the parallel DBScan variant with a 0.01/0.7 CDD/LSD threshold. We clustered 146,264 native libraries from 40 pregroups chosen at random, resulting in 4,201 clusters. At first, discriminative power was determined by measurement of up to 50 intra-cluster and 50 inter-cluster LSDs per cluster of the randomly sampled pregroups. This made for a total of 209,850 inter-cluster and 66,119 intra-cluster LSDs. 98.97% of them were smaller than 0.4, and only one of 209,850 was greater or equal 0.9. Further only 13 were greater or equal 0.8 and only 52 greater or equal 0.7. 97.45% of the intra-cluster LSDs were greater or equal to 0.6, with values between 0.7 and below belonging exclusively to libraries with a low function count.The average intra-cluster LSDs of 0.89 and inter-cluster LSDs of 0.08 show a clean separation of related and unrelated libraries.

To confirm the similarity inside clusters a name-based library comparison of over ten thousand randomly selected cluster samples was performed. It was observed that in 99.87% cases the sampled library was identical or a version of the predominant native library for their respective cluster. Manual analysis for the remaining cases yielded similar method-level CFGs for the sampled library and the predominant library. In that regard versions with different ABIs were also in the same clusters. These results indicate that the centroid fingerprints detected variants among a huge number of native libraries with high accuracy.

4.2 Approach Efficiency

Efficiency was a key aspect and performance enhancing runtime considerations were already presented in Section 3.3. We evaluate the efficiency of the fingerprint computation and the similarity analysis separately, each in their respective subsections.

Fingerprint Computation. A runtime cost analysis was performed several times on 24 Android native libraries of different sizes (104 to approx. 50,000 functions) on a Core i7 CPU and its results are displayed in Figure 3. The figure also includes the proportion of database read and write operations. The generation of centroids on existing CFGs libraries with less than 10,000 functions, which represented about 89% of all identified native libraries in our dataset, was in the order of hundreds of milliseconds. For large libraries with approx. 10,000 to approx. 50,000 functions, runtimes were up to seven seconds at most. Including IDA's autoanalysis to derive the CFGs, runtimes for libraries with under 10,000 functions were around a half up to six seconds, progressing super-linear with library size. The throughput of our implementation, including IDA's auto analysis, the database interactions and the centroid computation can be seen in Figure 4

Similarity Measurement. The second efficiency criteria was the similarity measurement runtime. Edge cases, e.g., method stubs referencing imports were disregarded during this analysis because they are encoded as zero vectors due to their single basic block nature.

Figure 5 shows the runtimes for similarity computation for versions of the same library and unrelated libraries separately. We additionally take database read operations for centroid fingerprint

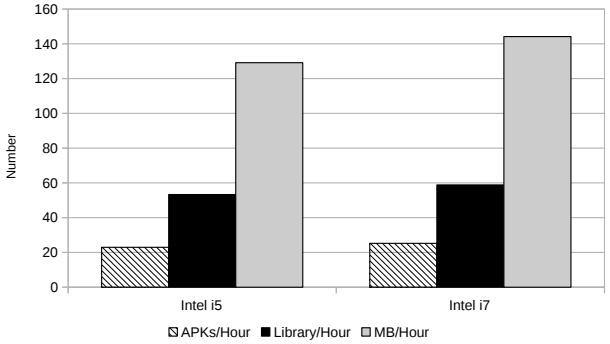

Figure 4: Throughput for the implementation of the centroid fingerprint computation on Intel i7 and i5 processors. These values include the centroid computation, database interaction, and IDA's auto analysis performed for each new binary.

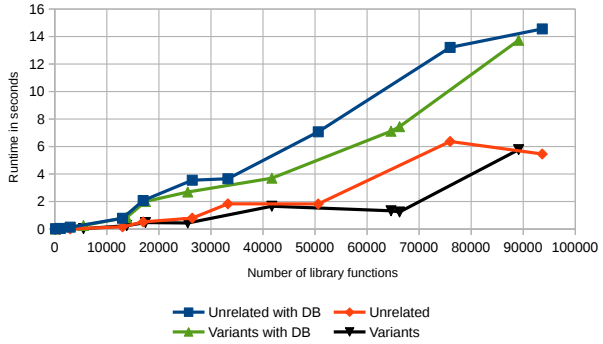

Figure 5: Runtime measurements for the similarity computation of centroid fingerprints.

retrieval into account. Similarity measurement on in-memory centroids of libraries with less than 10,000 functions was conductible within up to a few hundred milliseconds. On libraries with ten thousands of functions runtimes were several seconds and increased quadratically, which is in accordance with algorithmic properties. The library version comparisons were overall more expensive, and the MongoDB fingerprint readout times showed to be non-marginal proportions.

4.3 Obfuscation and Compiler Robustness

Obfuscation robustness is an additional ability which software variant detection metrics may or may not possess. Having high obfuscation robustness means a technique can withstand significant obfuscation attempts, i.e. code variations, and still detect a similarity relation between obfuscated and unobfuscated code. Android obfuscation techniques range from simple manually applicable changes, e.g., byte patching or renaming to fully automated and extensive techniques, such as APK packing or APK bytecode stripping [6].

We investigated the robustness of our improved centroid approach with selected Android obfuscation techniques that were often employed in the wild [39]. Specifically these include modifications to the APK meta data, native library relocation, native library renaming, variable name obfuscation, binary stripping, payload placement in native libraries, junk function insertion, literal and arithmetic encoding and basic block segment reordering. Additionally, we introduced control-flow altering obfuscation techniques, namely opaque predicates, function in- and outlining and control-flow flattening into our experiments. To research the effects of native library obfuscation we used open source libraries from the AOSP, including renderers, audio- and video processors, and compile them to ARM 32-bit libraries. The control-flow alterations were performed with *Tigress*, a `C` obfuscator by Collberg [9]. For each technique we obfuscated 53 methods with varying sizes from the AOSP libraries.

Native library obfuscation can prove effective, especially if advanced control-flow tampering techniques are involved. The introduction of junk functions and the inclusion of malicious payloads did not obstruct library similarity and CDDs of the available methods did not vary. String-based obfuscations had little to no effect at all. The effect of junk code insertions varies, depending on the number of additionally introduced instructions. They increase the weight of the CFG node or nodes they are introduced to and therefore shift the centroids mass accordingly, the CDDs' median for our test cases with varying degrees of junk code was 0.1. Put into context these values are already quite high, mitigating detection if CDD thresholds of 0.01 are considered which allow for library variant scanning with low FPRs. Similar results were observed for integer arithmetic encoding, resulting in a CDD median of about 0.3 and function outlining with a CDD of 0.5. Opaque predicates combined with junk code, and control flow flattening performed by far the worst. Their app changes the CFG quite considerably resulting in completely differing centroids. Both techniques caused CDDs of 1.0 for almost every native function, effectively disabling native library variant detection.

Our results show that APK structure and meta data changes, e.g., modified APK meta data, library path and extension changes as well as binary stripping can be resisted, because they operate on an abstraction layer that is removed during the centroid computation. Those results can be applied to differing compiler versions and optimization levels, too, which typically introduce smaller changes to the binary code than obfuscation.

5 FINDINGS

To conduct experiments on malware detection of the centroid approach, we use a database of about 140 GB of malicious Android apps. Samples were gathered from the preexisting datasets Drebin[3] and Contagio[31], and additionally new samples from the malware repositories AndroMalShare[5] and VirusShare[38] were used. The 36,908 malware samples resulted in 75,517 ARM 32-bit native libraries that were extracted and their centroid fingerprint introduced to the database.

Malware in the Dataset. We used results from malware analysis firms such as Symantec and Kaspersky to narrow down a list of

Market Name	Malicious NLs	Detected NLs
Androidapkfree	2	120
anruan	5	20
apkpure	5	160
playmob	26	1089
mumayi	36	151
baidu	44	368
apkmirror	80	3393
nduo	128	396
up2down	219	4195
apkworld	307	1880

Table 2: Malicious native libraries among ARM 32-bit native libraries. Non-listed markets were not found to be infected by the selected malware families' native components. Each detected malware family result was confirmed using the VirusTotal API.

promising malware families that abused native libraries to either introduce exploits or perform malicious actions. This sets the focus for our dataset similarity comparison to 29 malware families, each including several native library versions. After a similarity comparison on the fingerprint database clusters for all families were found, the ten largest belonging to Bios.A, DroidDream, GingerBreak, Godless, KungFu, OldBoot, Rootnik, TatooHack, VikingHorde, and Ycchar. The found malware clusters assembled family members while excluding known malware native libraries other than those of the considered families.

Comparison to VirusTotal. To compare the detection ratio of our approach with established malware detection tools, we tested how many of the malicious native libraries would have been detected by VirusTotal. We separately uploaded the APK and its native library flagged as malicious through our clustering to VirusTotal. Since VirusTotal has a limitation on uploads we randomly sampled applications from the detected clusters. A result was considered as malware recognition if at least two scanners identified the sample as malicious.

The randomly sampled applications contained 2,476 known malware native library samples of 1,795 apps. The native library samples were clustered into totally 249 clusters and 100 noise points. If the submitted file was an APK, 124 of those clusters were detected by VirusTotal to contain native library samples of newly recognized malware, and 1,957 of 9,877 previously undetected apps of the library samples in the 249 clusters were thereby recognized as malicious. Among the detections were families like SimpLocker, Rootnik and Kungfu, which were confirmed to use malicious native libraries through reports from malware analysis labs. Table 2 shows the distribution of all detected malware samples inside clusters across the sampled markets. Non-listed markets were not found to be plagued by the selected malware families native components.

On native library file level, the VirusTotal approach recognized only 9 out of 249 clusters as malicious. This points out that native libraries pose a significant attack vector since the detection ratio by commercial tools seems to be low.

6 RELATED WORK

The research landscape of code similarity measures offers a wide range of approaches, ranging from fragile hash-based value calculations [19, 42] to graph-based methods, e.g., API call graph [14, 40] and program dependency graph based [11, 12] methods.

Notable is the emergence of Android-specific approaches which utilize machine learning to detect functionality similarity through feature vector detection. PiggyApp [41], which was invented to detect piggybacked Android apps, is an exemplary tool relying on feature vectors, which are built from statically extracted semantic app features. Semantic app features are applicable to machine learning as performed by AndroTracker [25], and include requested permissions, intent types, kernel API calls and certificate content like authorship information. AndroSimilar [16] extracts features with a low a priori occurrence probability deduced from an empirical entropy study in a fuzzy hashing approach. The downside of machine learning-based techniques is the high susceptibility to code obfuscation, e.g., reflection or even renaming.

Current endeavors in code similarity detection can be categorized into hash value calculations and graph-based approaches, e.g., API call graphs (ACGs), program dependency graphs (PDGs) and control-flow graphs (CFGs). The centroid-based approach, which is categorized as the latter since it operates on CFGs, was first applied to app clone detection based on Dalvik bytecode in 2014 [7]. Examples for the former category on Android, namely hash value calculations, are presented by Juxtapp [19] and DroidMOSS [42]. While Juxtapp applies feature hashing on a sliding opcode window, DroidMOSS uses fuzzy hashing. Although being faster than other most approaches, hash-based approaches have an inherent weakness to even minor code variations.

DroidSIFT [40] and DroidLegacy [14] construct ACGs in order to perform static classification of Android malware behavior. DroidSIM [35] creates ACGs for code reuse detection. ACG-based malware tracking can have the drawback that some benign apps use malware-typical API call sequences, e.g., for dynamic code loading. The Android bytecode clone detector DNADroid [11] calculates subgraph isomorphism on PDGs at the cost of high computational effort. The same authors presented Andarwin [12], which also relies on PDGs but aims to speed up computations by stripping control flow dependent edges, splitting PDGs into connected components and representation of the components with vectors which count binary operation types.

Regarding Androids' event-driven OS, which allows for an easy obfuscation of API call patterns, and the missing adaptation for native code in PDGs, recently CFG-based gained attention as well. Chen et al. [7] and Alam et al. [2] both used CFGs properties for Android code clone and variant detection. The former study proposed to use the CFG-based fingerprint called *centroids*, which are similar to mass centers based on control flow characteristics and basic block size. In 2016, the same authors applied the centroid approach to detect malicious Android bytecode libraries at large scale [8]. Alam et al. dealt with both bytecode and native code variant detection. Their approach relies on an CFG annotated with intermediate language

statements (ACFG), on which subgraph isomorphism analysis is performed.

When comparing the many methodologies usually the measures efficiency and accuracy are consulted to position them in the research landscape. Hence, we will put our approach into context with the other methodologies.

Efficiency. Our implementation took a few hundred milliseconds for very small libraries of about 100 functions to several minutes for extremely large libraries of 50,000 functions and more while roughly displaying linear growth. On the efficiency scale our approach performed worse that hashing-based approaches, e.g., Juxtapp which was shown to extract signatures and perform pairwise comparisons of 95,000 apps on an EC2 cluster with 200 threads in less than 4 hours [19]. This is a significantly faster scanning than measured in this study for the centroids on native libraries, but at the price of comparably low accuracy and robustness. Runtimes on the same level as our implementation can be found in ACG-based and annotated CFG-based approaches, for example, DroidSIFT [40] with 175.8 seconds per app on average on an Intel Xeon CPU, and DroidNative [2] with 59.81 s/MB for disassembly, annotated CFG generation and classification.

Our implementation is faster than the PDG-based tool DNADroid and beats Andarwin [12] in terms of presimilarity analysis. Andarwin reported 109 seconds on average per thread to create PDG-based semantic vectors which is slower that our fingerprint generation. However, Andarwin applies locality sensitive hashing during similarity computations and thereby shows better efficiency values after the fingerprint computation.

Accuracy. The centroid approach was parametrized for a low FPR. Section 3.3 details that the CDD/LSD chosen had still allowed the detection of library variants. Including small sized libraries of under 400 functions the FPR grew to 10.8%. Excluding these libraries, a centroid difference degree/library similarity threshold pair of 0.01/0.7 showed no single false positive among 1500 unrelated pairs and detected same versions of a library with 54.6% accuracy. In this regard, a detection rate of 89% on the investigated library version set with an FPR of 10.8% was determined for the CDD/LSD threshold pair 0.01/0.7. Notably also versions with different ABIs had fingerprint similarity scores which allow discrimination from unrelated pairs. The randomly sampled clusters of the centroids DB with an average intra-cluster LSD of 0.89 and inter-cluster LSD of 0.08 showed that centroids-based native library variant scanning has overall high discriminative power.

Compared to other approaches our implementation performs in the range of DroidLegacy [2] with a detection rate of 92.73% with 20.83% FPR for malware variant detection. The authors of DroidSIFT presented 93% recall and 5.15% FPR for ACG-based bytecode signatures, but aside from substantial ACG-inherent shortcomings those appealing numbers stem from a classifier specifically trained in malware family detection [40]. The PDG tool Andarwin was reported to cluster bytecode clones with a 3.72% FPR, and its evaluation includes no FNRs [12]. Chen et al. in contrast observed highly attractive FPRs for centroids of bytecode, namely no single false positive [7, 8]. The malware detection experiments via centroids DB clusters with known malware demonstrated that centroids-based scanning can be utilized for native library malware detection at large scale. In contrast to application level VirusTotal only recognized a few clusters as malicious on native library level, what again suggests a need for new effective techniques of scanning for malicious native libraries.

7 DISCUSSION

While the comparison with other similarity measures shows a good accuracy and high efficiency, the current implementation still has limitations that motivate future areas of research for the centroid approach.

7.1 Limitations

One limitation of our study is its exclusion of x86, MIPS and 64-bit ARM Android native libraries as the IDA Pro plug-in does not support the respective ABIs yet. The signature database build was done by crawling for ELF files and carving their sections, being effective but it does not identify with absolute certainty each native library. Furthermore, encrypted ELF files that are decrypted at runtime are not detected by our approach. ELF files using heavy obfuscation, such as packers or virtualization based obfuscation require deobfuscation prior to the analysis. While some tools for deobfuscation exist [24, 32, 34], the impact on the centroid accuracy after deobfuscation is not clear and needs to be assessed.

The clustering was performed in native library pregroups to confine the experiments' runtime to sane levels, thus the clusters might miss samples at opposite pregroup border regions and core functionality library variants. With respect to the present study's accuracy evaluation false negative rates were derived on the basis of native library versions, which are not totally representative for malicious native library variants or native library clones.

7.2 Future Work

The results of this work suggest that machine code variant detection via centroids is an accurate method. A promising aspect is the automatic identification of heavily reused standard libraries in reverse engineering. The IDA Pro disassembler currently applies so-called FLIRT signatures [20]. Those signatures store the first 32 bytes of each function in prefix trees in signature files and function names are kept in the tree leafs. Centroids are an interesting alternative for such an approach as they have a memory footprint of only four floating point values per function, and their accurateness avoids multistage collision handling efforts.

We focussed on malware in the evaluation, but another nearby application of similarity metrics is plagiarism detection. IP theft through application clones is still an unsolved problem. Respective large-scale and potentially cross-market studies might reveal clones [8], rebranding and copyright infringement. The centroids technique is also applicable to fields like automatic library version identification and vulnerability pattern detection.

Looking at the overall fingerprinting approach of our study, several possible alternative parametrization models are not investigated yet. Regarding the native library similarity degree definition, for instance the Jaccard-Index would be applicable to a full library variant scanning which relates the cardinality of the intersection

of two centroid sets to that of their union. Also alternative definitions of centroid difference degrees are possible, including different normalizations or using the minimum coordinate pair distance.

8 CONCLUSION

We present an improved version of the centroid-based similarity measure and the first study of native library use among Android malware in third-party app stores. The improvements operate on an algorithmic level which we leverage to create an efficient and accurate analysis system. A dataset of over 508,745 apps from 18 different third-party APK stores serves the purpose of evaluation. We established accuracy parameter combinations that allow for a clone and variant detection with low FPR of 10% for all libraries and an FPR of 1% for libraries bigger than 400 functions, while retaining a precision of 89% in both cases. A malware centered analysis of our dataset uncovered infection ratios of up to 17.05% among the third-party app stores and outperformed VirusTotal, especially if only the malicious native library was analyzed. Looking at the efficiency of our approach it performed better than PDG-based and Android bytecode clone detection.

REFERENCES

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, Principles, Techniques. *Addison Wesley* 7, 8 (1986), 9.

[2] Shahid Alam, Issa Traore, and Ibrahim Sogukpinar. 2015. Annotated control flow graph for metamorphic malware detection. *Comput. J.* 58, 10 (2015), 2608–2621.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*.

[4] Pavel Berkhin. 2006. A survey of clustering data mining techniques. In *Grouping multidimensional data*. Springer, 25–71.

[5] Ashish Bhatia. 2015. AndroMalShare. http://sanddroid.xjtu.edu.cn:8080/, accessed on 16. March 2018.

[6] Marcel Busch, Mykolai Protsenko, and Tilo Müller. 2017. A Cloud-Based Compilation and Hardening Platform for Android Apps. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ACM, 37.

[7] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 175–186.

[8] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 357–376.

[9] Christian Collberg. 2015. The Tigress C diversifier/obfuscator. *Retrieved August* 14 (2015), 2015.

[10] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1-10 (2001), 1–8.

[11] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the clones: Detecting cloned applications on android markets. In *European Symposium on Research in Computer Security*. Springer, 37–54.

[12] Jonathan Crussell, Clint Gibler, and Hao Chen. 2013. Andarwin: Scalable detection of semantically similar android applications. In *European Symposium on Research in Computer Security*. Springer, 182–199.

[13] Andrew Rice Daniel Thomas, Alastair Beresford and Daniel Wagner. 2018. AVO: Collection of all android vulnerabilities. https://androidvulnerabilities.org/all, accessed on 16. March 2018.

[14] Luke Deshotels, Vivek Notani, and Arun Lakhotia. 2014. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014*. ACM, 3.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[16] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, Manoj Singh Gaur, and Ammar Bharmal. 2013. AndroSimilar: robust statistical feature signature for Android malware detection. In *Proceedings of the 6th International Conference on Security of Information and Networks*. ACM, 152–159.

[17] Google LLC. 2018. Getting started with NDK. https://developer.android.com/ndk/guides/index.html, accessed on 15. March 2018.

[18] Andy Greenberg. 2011. Phone rootkit carrierIQ may have violated wiretap law in millions of cases. https://www.forbes.com/sites/andygreenberg/2011/11/30/phone-rootkit-carrier-iq-may-have-violated-wiretap-law-in-millions-of-cases/, accessed on 16. March 2018.

[19] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. 2012. Juxtapp: A scalable system for detecting code reuse among android applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 62–81.

[20] HexRays Inc. 2018. FLIRT Signatures in depth. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml, accessed on 30. March 2018.

[21] Xuxian Jiang. 2011. Security alert: New sophisticated android malware droidkungfu found in alternative chinese app markets. *URL http://www. csc. ncsu. edu/faculty/jiang/DroidKungFu. html* (2011).

[22] Caleb Fenton Jon Sawyer, Tim Strazzere. 2015. Offensive and Defensive Android Reverse Engineering. https://github.com/rednaga/training/tree/master/DEFCON23, accessed on 15. March 2018.

[23] Anatoli Kalysch. 2017. Third Party APK Store Crawlers. https://github.com/anatolikalysch/APKCrawler, accessed on 03. April 2018.

[24] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. 2017. VMAttack: Deobfuscating Virtualization-Based Packed Binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*. ACM, 2.

[25] Hyun Jae Kang, Jae-wook Jang, Aziz Mohaisen, and Huy Kang Kim. 2014. Androtracker: Creator information based android malware classification system. In *Information Security Applications-15th International Workshop, WISA*, Vol. 8909.

[26] Kaspersky Lab. 2018. Mobile malware evolution 2017. https://securelist.com/mobile-malware-review-2017/84139/, accessed on 10. March 2018.

[27] Thomas Lengauer and Robert Endre Tarjan. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1, 1 (1979), 121–141.

[28] Dominik Maier, Tilo Müller, and Mykola Protsenko. 2014. Divide-and-conquer: Why android malware cannot be stopped. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*. IEEE, 30–39.

[29] McAfee Labs. 2017. Android Banking Trojan MoqHao Spreading via SMS Phishing in South Korea. https://securingtomorrow.mcafee.com/mcafee-labs/android-banking-trojan-moqhao-spreading-via-sms-phishing-south-korea/, accessed on 15. March 2018.

[30] Raymond T. Ng and Jiawei Han. 2002. CLARANS: A method for clustering objects for spatial data mining. *IEEE transactions on knowledge and data engineering* 14, 5 (2002), 1003–1016.

[31] Mila Parkour. 2015. Contagio Mobile. http://contagiominidump.blogspot.com/, accessed on 19. March 2018.

[32] Rolf Rolles. 2009. Unpacking Virtualization Obfuscators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies (WOOT'09)*. USENIX Association, Berkeley, CA, USA.

[33] Search Security. 2015. How did a malicious app slip past Google Play app store security? http://searchsecurity.techtarget.com/answer/How-did-a-malicious-app-slip-past-Google-Play-app-store-security, accessed on 15. March 2018.

[34] Tim Strazzere. 2015. android-unpacker. https://github.com/strazzere/android-unpacker, accessed on 13. March 2018.

[35] Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. 2014. Detecting code reuse in android applications using component-based control flow graph. In *IFIP International Information Security Conference*. Springer, 142–155.

[36] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[37] Trend Micro. 2014. Malware in Apps' Clothing: A Look at Repackaged Apps. https://www.trendmicro.com/vinfo/us/security/news/mobile-safety/malware-in-apps-clothing-a-look-at-repackaged-apps, accessed on 15. March 2018.

[38] VirusShare.com. 2018. VirusShare Malware Repository. https://virusshare.com, accessed on 01. April 2018.

[39] Ilsun You and Kangbin Yim. 2010. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. IEEE, 297–300.

[40] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1105–1116.

[41] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 185–196.

[42] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 317–326.