Quokka

ACADEMIC PAPER

Resilient and Scalable Cloned App Detection

using Forced Execution and Compression Trees

Resilient and Scalable Cloned App Detection using Forced Execution and Compression Trees

Mohamed Elsabagh
Kryptowire
melsabagh@kryptowire.com

Ryan Johnson
Kryptowire & George Mason University
rjohnson@kryptowire.com

Angelos Stavrou
Kryptowire & George Mason University
astavrou@kryptowire.com

***Abstract*—Android markets have grown both in size and diversity, offering apps that are localized or curated for specific use cases. It is not uncommon for users to be unaware of the exact app version or name they should be installing. This has given rise to the threat of app cloning where adversaries copy the package of an app, minimally modify its code, and redistribute the clone on the market to gain a monetary advantage or to distribute malicious payloads. Existing clone detection methods use static signatures that can be evaded using control- and data-flow obfuscation. Moreover, many approaches do not scale with the number of apps, code size, and complexity, leading to prohibitive detection time requirements. In this paper, we introduce *Dexsim*, a dynamic analysis based system to accurately index apps and identify bytecode similarities. We propose a novel bytecode indexing and matching algorithm that employs concepts from forced execution and LZ78 compression trees, and scales linearly with the number and size of apps. Our experiments on 28k cloned benign and malicious apps showed that *Dexsim* is both scalable and resilient to obfuscation, ferreting out clones within 8 ms pair-wise on average with at least 90% accuracy.**

***Index Terms*—Android Vetting, Clone Detection, Forced Execution, Compression Trees, Malware Classification**

I. Introduction

The market share of Android devices has expanded dramatically over the past few years: as of 2016, Android is the dominant mobile platform with a share of more than 86% of the world market [1]. Android applications (apps, for short) are distributed via an official market, Google Play, and various third-party markets where users can download and install apps. Android developers typically profit from apps either via paid apps or by monetizing free apps using advertisements, subscriptions, and premium features. Unfortunately, Android apps are distributed in an open and easily modifiable format that enables adversaries to clone and redistribute them with little effort. This makes Android apps lucrative targets for plagiarism and piracy, also known as app repackaging or cloning.

There have been considerable efforts to prevent or detect app cloning [2]–[14]. However, the majority of the techniques did not consider the impact of obfuscation on their detection capability. This is especially important since advanced code obfuscation techniques for Android apps are widely accessible to hinder reverse engineering attempts. Recent studies have shown that applying obfuscation techniques enabled clones to evade detection [15]–[17] primarily due to the fact that static analysis approaches cannot resolve runtime code constructs or unroll obfuscated code. Several clone detection techniques assume 3rd party libraries were filtered out from the app code before the analysis which is unrealistic in practice [15]. Other techniques assume GUI structure and hierarchies cannot be obfuscated, an assumption that has been invalidated in recent work [14]. Moreover, advanced detection methods such as graph isomorphism based approaches [18]–[20] do not scale with the number or size of apps, incurring prohibitive detection time that prevents practical deployment.

In this paper, we propose a novel system for exposing bytecode similarities in an efficient, accurate, and resilient way. Our approach uses dynamic analysis and forced path execution to characterize code execution even in the presence of heavy obfuscation. The proposed approach does not require whitelisting of libraries nor is it affected by changes in resource files and GUI structure. *Dexsim* is highly parallelizable, making it suitable for large-scale analysis that is linear in the number of apps. It operates by directly finding the most similar apps in its database to a submitted app. To achieve this, it extracts code patterns from the bytecode trace of an app by dynamically forcing execution through all components in the app. It then efficiently indexes the apps and their traces into a database of depth-bounded LZ78 [21] prediction trees (one tree of probability assignments per app).

Dexsim quickly determines if an incoming app is a clone and finds the nearest apps in its database to the incoming app using a novel similarity measure based on the traces of the incoming app and the statistical properties of the prediction trees in its database. Our approach is designed based on the following key insights: *a*) A cloned app produces similar dynamic bytecode traces to its source (original) app, even if the clone's static control-flow graph (CFG) was obfuscated, in order for the app's functionality to be reserved. *b*) Traces from a clone shall have a significant number of bytecode subsequences that can be highly compressed using the same probability assignments in the compression tree of the traces from the source app. §III discusses the technical detail of the proposed system.

We evaluated a prototype of *Dexsim* against advanced control and data flow obfuscation (see §IV) and investigated its ability to identify malware families. *Dexsim* achieved high accuracy: at least 96% with single obfuscation; and at least 90% with multiple obfuscations applied in serial on the same app. It correctly identified malware families seen in the wild with at least 97% accuracy. It also incurred less than 8 ms detection time per app-pairs, allowing for large-scale deployment as

978-1-5386-5790-4/18/$31.00

detection time scales linearly in the number of indexed apps.

II. Background and Threat Model

Android apps are distributed in the Android Application Package (APK) format. Each APK file contains bytecode compiled into a Dalvik Executable (DEX) file, along with resource files (e.g., images, sounds) and a manifest file (AndroidManifest.xml) that lists the app package name, version number, used permissions, and app components. The DEX file is the input to the on-device Android runtime (either Dalvik or ART) and it contains compiled Dalvik bytecode for all classes used by the app. Native C/C++ components are packaged separately from the DEX executable.

Developers must sign an app with a private key before publishing it. However, due to the portable and open format of APK files, several tools exist to manipulate Android apps postmortem, i.e., after the app has been compiled and distributed. This makes it very easy for malicious actors to plagiarize (clone) or inject malicious behavior into benign apps, by unpacking an app, modifying its meta information or bytecode, and then repackaging it into a new APK and redistributing it to the market, completely without access to the app's original source code.

Threat Model: The aim of this work is to detect apps that share a nontrivial amount of code (app clones). We assume an adversarial setup where an app's meta data, such as developer and package names, are unavailable or cannot be trusted. We detect clones based on bytecode alone without using GUI resources or source code. We assume side information, such as debug symbols and method names, are stripped from the bytecode. We assume the apps can be obfuscated at both the control-flow level (e.g., obfuscated conditional statements) and data level (e.g., obfuscated identifiers and literals). We do not utilize data constructs (e.g., string literals) in detection since they are easily modifiable by adversaries.

We directly work with Android apps compiled and assembled into their final DEX executable (classes.dex) in an APK file. DEX is the file format of Android executable bytecode that gets loaded and executed by the ART (previously Dalvik) runtime virtual machine on Android devices. Our system uses its own DEX forced execution engine to force execution of apps bytecode and runs standalone. It currently ignores native code shared objects (.so files) which we leave for future work.[1]

III. Clone Detection System

Figure 1 shows the workflow of our system. It operates in four phases: 1) common preprocessing, 2) trace collection, 3) indexing, and 4) detection. In the following, we discuss each phase in depth and analyze the detection time complexity.

[1]Native code is native platform-specific assembly compiled into a shared object (.so file) and invoked from bytecode via the Java Native Interface (JNI). It was estimated that only about 8% – 12% of Android apps use native code, mainly for fast access to I/O and graphics extensions [6], [22].

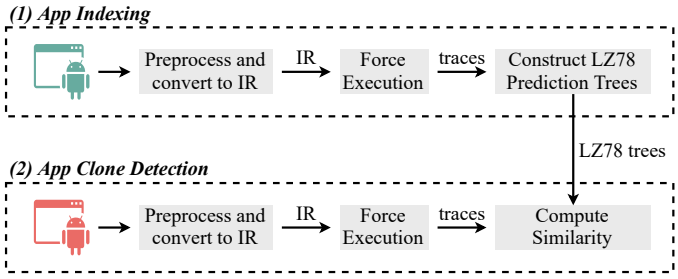

Fig. 1. An overview of *Dexsim* which works in two phases: First, an indexing phase to build a database of LZ78 trees of bytecode traces. Second, a detection phase that finds the nearest trees in the database to an incoming app.

A. Preprocessing

Dexsim starts by unpacking the incoming app and extracting its DEX and manifest file. It then extracts all components declared in the manifest file and disassembles the DEX file into Smali [23] which is a human-readable intermediate representation (IR) of Dalvik bytecode. This makes the bytecode easier to process by later stages of the analysis. It then statically analyzes the bytecode and extracts all possible entry points, such as activities, services, and broadcast receivers, in case the manifest file was corrupt or incomplete. We define an entry point as an app component that can result in full or partial execution of the app via direct (user action) or indirect (inter-process communication) invocation.

B. Forced Execution & Trace Collection

For each app entry point, *Dexsim* executes all control-flow paths rooted at the entry point and records traces of the executed bytecode down all control flow paths. It then encodes the traces as streams of bytes, where each byte corresponds to one DEX opcode (a value in $[0...255]$). We developed a forced path execution [24], [25] emulation engine that force executes each app's component by starting at the component class constructor and initiation methods (i.e., onCreate, onStart, or onReceive). We also force execution down component life cycle, UI listeners, and callback methods. The execution of different paths through a component is modeled using a graph that encodes all possible control transfers that can be taken within the component. The nodes in the graph represent conditional statements that alter the control flow. The engine iteratively forces the execution of all prime paths in the graph: if execution traverses a path that partially overlaps with another path that has already been traversed then the reference to the current node will be moved along already established nodes in until the paths diverge. We limited (in)direct recursion and loops to a single iteration. Figure 2 shows an example of a dynamic CFG and bytecode trace generated by the engine and execution paths that static analysis solutions would not be able to follow, e.g., the actual target of the virtual invocation marked by "*" in the figure can only be resolved dynamically.

We emulated the execution of the majority of the Android API. We built a portion of the Android API from the Android Open Source Project (AOSP) code, including all Android

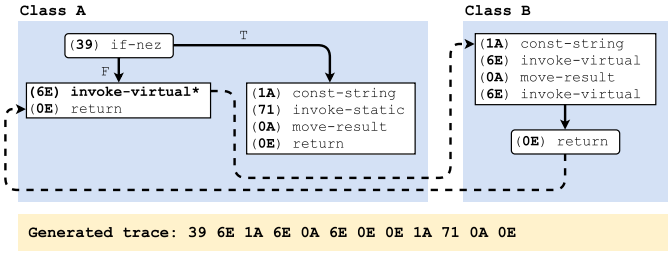

Fig. 2. Sample dynamic CFG and the corresponding trace. Numbers between brackets are the DEX opcode values. Solid lines are direct (static) control flow. Dashed lines are indirect (dynamic) control flow resolved by the engine.

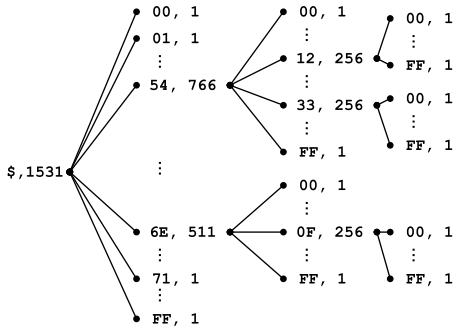

Fig. 3. LZ78 annotated compression tree produced from the trace: `71 54 6E 54 33 54 12 63 0F`. Σ is the DEX opcodes 0x00 to 0xFF ($|\Sigma| = 256$). Each node in the tree contains both an opcode and a count of all child nodes. Each internal node has all $|\Sigma|$ children. For computing $P_D(\texttt{54 71 6E 0F})$, the following path is traversed: \$ $\rightarrow$ 54 $\rightarrow$ 71 $\rightarrow$ \$ $\rightarrow$ 6E $\rightarrow$ 0F, giving a probability of $\frac{766}{1531}\frac{1}{766}\frac{511}{1531}\frac{256}{511} = \frac{256}{1531*1531}$.

callbacks for Activity, Service, Content Provider, and Broadcast Receiver components, all Android callbacks identified in the EdgeMiner [26] dataset, and callbacks introduced in recent Android versions up till version 7.1.

To guarantee unbiased execution, we split the allotted analysis time among components in proportion to each component's bytecode length (the number of bytecode statements in the component). In our preliminary experiments, we empirically found that a time limit of 30 s of forced execution offers a reasonable balance between accuracy and detection time for practical purposes. We plan on systematically studying this trade-off in a future work.

C. App Bytecode Indexing

The output from the forced execution phase is a set of traces $\mathcal{T} = \{T_i\}$, where each T_i is a forced path execution bytecode trace of component i in the app through all executed paths. *Dexsim* splits $\mathcal{T}$ into smaller non-overlapping chunks using a running window of a fixed size w. It then computes the LZ78 [27] *prediction tree* of $\mathcal{T}$ and stores the app identifier (the app unique package name) and the LZ78 dictionary in a database for later retrieval. The window size w influences the LZ78 dictionary size as well as the detection accuracy as w controls the maximum length of a pattern that the prediction tree can learn. In the following, we briefly discuss how the LZ78 prediction trees are constructed and their main usage in prediction and detection.

Of particular interest to us is the prediction component of the LZ78 algorithm, which utilizes the core LZ78 parsing algorithm for making predictions [21]. Parsing in LZ78 works as follows. Given a string q constructed from some alphabet Σ (a finite set of symbols) LZ78 scans q into adjacent non-overlapping substrings. The substrings are stored in an efficient compression tree structure D, often known as the "dictionary."[2] Each time a substring s' is parsed, the algorithm inserts s' to D if s' is not already in D and it extends any substring s in D, i.e., $s' = s||\sigma$ where $s' \notin D$, $s \in D$ and $\sigma \in \Sigma$. This process continues until q is consumed.

LZ78 prediction [21] builds on this by annotating D with empirical probabilities from the counts of child substrings, producing a prediction tree. A child node $s' \in D$ is a parsed substring that extends another parent substring $s \in D$. Leaf nodes correspond to substrings with no extensions. For an alphabet Σ of size n, each node $s \in D$ has n children and n counters, one for each symbol $\sigma \in \Sigma$. The counter of a leaf node is always set to 1 for the empty string. Figure 3 gives an example of a trace compression tree. To compute the prediction probability $P(\sigma|s)$, i.e., a symbol $\sigma \in \Sigma$ extends a string s, we start from the root of D and traverse the tree according to s. When s is fully consumed, $P(\sigma|s)$ is computed as the counter value corresponding to σ divided by the sum of all counters of the current node. The prediction probability of s is given by:

$$P_D(s) = \prod_{i=1}^{|s|} P(s_i|s_1 s_2 \dots s_{i-1}), \tag{1}$$

where s_i is symbol number i in s, and $|s|$ is the length of s.

If a leaf node is reached before s was fully consumed, *Dexsim* continues the traversal the root node of D. This allows the sequences in D to "roll over" by distributing the computation of the prediction probability over all sequences in D, effectively countering control-flow alterations that may modify the order of statements collected by forced execution.

D. Bytecode Similarity Detection

For an incoming app in the detection phase, *Dexsim* processes the app, forces execution through it, and encodes the traces into $\mathcal{T}$. Then, for each LZ78 dictionary D in the database, it computes a similarity score between $\mathcal{T}$ and D as follows. For some trace t_i in $\mathcal{T}$, the prediction probability of t_i by D, i.e., $P_D(t_i)$, represents the likelihood of reproducing t_i from D. That is to say, the log-loss $-\lg P_D(t_i)$ quantifies the information loss endured in the reproduced trace from D compared to the original t_i. The quantity $-\lg P_D(t_i)$ also measures the compression rate of the trace t_i when using the probability assignments in D. The smaller the log-loss, the better the compression. By generalizing this, we can compute a set $\mathcal{H}$ of normalized log-loss values for the subtraces in $\mathcal{T}$:

$$\mathcal{H}(\mathcal{T}, D) = \left\{ \frac{-\lg P_D(t_i)}{|t_i|}; i = 1 \dots |\mathcal{T}|, t_i \in \mathcal{T} \right\}. \tag{2}$$

[2]Dictionaries always contain the empty string \$.

The *dissimilarity* score between $\mathcal{T}$ and D is the average over $\mathcal{H}$, i.e., the average normalized log-loss given by:

$$d(\mathcal{T}, D) = \mathbb{E}\left[\mathcal{H}(\mathcal{T}, D)\right], \tag{3}$$

where $\mathbb{E}[.]$ is the expectation function.

The incoming app is considered a clone if the dissimilarity score is less than some threshold θ, determined on a per-app basis as follows: For each indexed app, re-traverse the LZ78 dictionary of the app using the app's *own* traces that were used in building the dictionary, record the mean and standard deviation of the normalized log loss over the app's traces set, and choose θ to two standard deviations above the mean. More formally, for each app a in the database, we compute:

$$\mathcal{H}_a = \mathcal{H}(\mathcal{T}_a, D_a), \tag{4}$$

$$\theta_a = \mathbb{E}[\mathcal{H}_a] + 2\sigma[\mathcal{H}_a], \tag{5}$$

where $\mathcal{T}_a$ is the app's traces set, D_a is the app's dictionary, and $\sigma[.]$ is the standard deviation function.

Alternatively, *Dexsim* can return the nearest k apps to the incoming app by returning the apps corresponding to the least k dissimilarity scores.

For convenience, a normalized relative similarity score $s(b, a) \in (0, 1]$ between an incoming app b and some app a in the database can also be computed as:

$$s(b, a) = 1 - \frac{\left|d(\mathcal{T}_a, D_a) - d(\mathcal{T}_b, D_a)\right|}{\max(d(\mathcal{T}_a, D_a), d(\mathcal{T}_b, D_a))}, \tag{6}$$

where $\mathcal{T}_a$ and $\mathcal{T}_b$ are the traces sets for apps a and b, respectively, and D_a is the dictionary of app a. Note that $d(\mathcal{T}_a, D_a)$ is precomputed during the indexing stage. The quantity $s(b, a)$ also corresponds to the relative compressibility of the traces of b and a, using the probability assignments in D_a. The higher the value of $s(b, a) \in (0, 1]$ the higher the likelihood that the bytecode sequences in b and a follow the distributions in D_a.

Detection Time Complexity: The forced execution stage on an incoming app a is time-bounded and can take at most $O(cs_a)$ where s_a is the number of statements in the app. Each dictionary is traversed in time $O(|\mathcal{T}_a|)$. Hence, for a database of n apps, the complexity of computing the similarity score between a and the n apps is $O(|\mathcal{T}_a|n)$. Since $|\mathcal{T}_a| \leq s$, where s is the total number of statements in all the n apps, the full app detection time reduces to $O(cs)$. Hence, the total detection complexity is $O(n)$ in the number of apps n in the database.

IV. Evaluation Results

We evaluated *Dexsim* using the DEX clone detection benchmarking framework presented in [16]. The framework utilized SandMark [28], the state-of-the-art obfuscation utility. SandMark heavily alters an app's bytecode by inserting, removing, and transposing bytecode constructs, and obfuscating the app's control and data flows. (This is a stronger setup than typical naive repackaging techniques that only modify a few lines of code in an app.) We also experimented with repackaged Android malware variants captured in the wild as concrete samples of malicious app clones.

We experimented with the following datasets:

1) BEN: a dataset of 3k unique benign apps.
2) BEN-O: 18k unique clones for the apps in BEN using a single obfuscator per clone.
3) CTR: a dataset of the highest 20 benign apps in terms of obfuscation rate.
4) CTR-O: 350 uniquely repackaged app clones for apps in CTR using a single obfuscator per app.
5) CTR-OS: 510 clones using two and three obfuscators in serial on the apps in CTR.
6) MAL: a dataset of 7k malware samples and repackaged variants seen in the wild.

BEN consisted of the top 3000 apps by number of downloads from Google Play in early 2017. We constructed BEN-O by obfuscating all the apps in BEN as follows: For each app in BEN, we produced several clones by applying single obfuscation algorithms from SandMark, i.e., one clone per each algorithm. This produced an average of 6 unique different obfuscated clones per source app (see Table I for detail).[3,4]

We constructed the CTR and CTR-O datasets by selecting the top 20 apps and their clones based on the number of successful obfuscations. Further, we constructed the CTR-OS dataset by obfuscating each app in CTR with the strongest obfuscation algorithms applied in serial (up to 3 algorithms). This gave a total of 450 obfuscated apps in the CTR-OS. We use CTR, CTR-O and CTR-OS to provide insights on different obfuscation algorithms and on the accuracy of *Dexsim*.

The malware dataset (MAL) consisted of 7,430 Android malware samples provided by the VirusShare project [30].[5] We collected the malware family labels by querying the VirusTotal [31] service with the malware hashes from the VirusShare dataset. We excluded samples that had generic family labels (e.g., "Spyware", "Downloader", "PUA", etc.), no family labels, native code samples, J2ME samples, and samples that failed parsing by Jar2dex.

We implemented *Dexsim* in Java and conducted all experiments on an Intel(R) Xeon(R) X5550 16-cores 2.67GHz machine with 72GB of RAM running Ubuntu 16.04.1.

A. Obfuscated Clones Identification Results

We used *Dexsim* to index all source apps in BEN then ran it in detection mode on all obfuscated clones in BEN-O. For each incoming clone in BEN-O, we configured *Dexsim* to output both the detector decision and the list of source apps in BEN sorted in ascending order by their dissimilarity score to the incoming clone, i.e., a nearest neighbors list. An incoming clone app is deemed matched if *Dexsim* detected it as clone and returned its source app among the nearest k neighbors for some value of k. Ideally, we would want to detect that a clone is indeed a clone, and to return its corresponding source app in BEN as the nearest neighbor, i.e., $k = 1$. We report

[3]Unless otherwise stated, we refer to the input app to an obfuscator as the "source" app, and to the output repackaged obfuscated app as the "clone" app.

[4]SandMark and Jar2dex failed to process some apps, which was also observed by prior studies [16], [29].

[5]The malware dataset can be downloaded from: https://archive.org/download/virusshare_malware_collection_000/VirusShare_Android_20140324.zip.

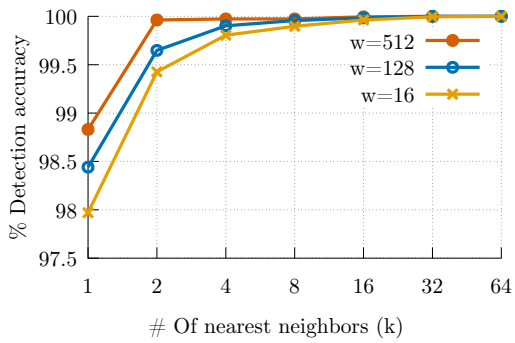

Fig. 4. Detection accuracy of all clones in the BEN-O dataset against their sources in the BEN dataset. w is the window size.

the detection accuracy as a function of the returned k nearest neighbors in BEN of each clone in BEN-O. Specifically, we compute the detection accuracy as: $100 * (1 - \frac{\text{\#unmatched clones}}{\text{\#all clones}})$.

Figure 4 shows the results. For the nearest neighbor case, *Dexsim* achieved at least 97.9% detection accuracy for a trace window size of 16 bytecodes. This increases to 98.8% for a window size of 512 bytecodes. By looking at the nearest 2 neighbors, the accuracy further increases to 99.8 for a window size of 16, and nearly 100% for a window size of 512.

Table I shows the average similarity score per clone type and obfuscation technique. The higher the scores, the better the resiliency against obfuscation. The scores were computed between each clone app and its source app and averaged over all clones of the same type. The results show that the proposed approach is highly resilient to layout and data obfuscation, maintaining at least 0.93 similarity between clones and sources.

Control-flow obfuscations had a relatively greater impact. While for 10 out of the 14 control-flow obfuscators, *Dexsim* achieved greater than 0.90 similarity, the following 4 obfuscators resulted in scores less than 0.90: *Interleave Methods*, *Method Merger*, *Opaque Branch Insertion*, and *Transparent Branch Insertion*. This is due to the heavily modified basic blocks resulting from these transformations. Nevertheless, the lowest score still conserves a high similarity at 0.87.

B. Serializing Multiple Obfuscation Algorithms

It is possible that attackers, in practice, apply multiple obfuscation algorithms to the same clone in order to evade or complicate detection. Hence, we also quantify the effectiveness of *Dexsim* against clones obfuscated via serializing multiple obfuscators. For this experiment, we focused on the strongest obfuscation algorithms based on the scores from §IV-A: *Interleave Methods*, *Method Merge*, *Random Dead Code*, *Opaque Branch Insertion*, *Branch Inverter,* and *Transparent Branch Insertion*. For these six algorithms, we produced clones of the CTR dataset by serializing all possible permutations of length up to three. We limited the number of serialized obfuscators to three since serializing four or more obfuscators failed for all the apps in CTR.[6] We believe this is due to conflicts between the

[6]It took 52 min per app on average to apply three obfuscators in serial.

TABLE I
AVERAGE SIMILARITY SCORE OF CLONES IN THE BEN-O DATASET TO THEIR SOURCES IN BEN. L, C, AND D REFER TO LAYOUT, CONTROL-FLOW, AND DATA-FLOW OBFUSCATION, RESPECTIVELY.

Obfuscator	Type	#Clones	Score
Constant Pool Reorderer	L	936	0.99
Interleave Methods	C	806	0.89
Block Marker	C	936	0.95
Dynamic Inliner	C	293	0.93
Method Merger	C	936	0.87
Class Splitter	C	469	0.94
Static Method Bodies	C	933	0.93
Simple Opaque Predicates	C	788	0.97
Random Dead Code	C	936	0.91
Inliner	C	324	0.95
Reorder Instructions	C	181	0.98
Insert Opaque Predicates	C	398	0.93
Opaque Branch Insertion	C	294	0.89
Branch Inverter	C	928	0.90
Trans. Branch Insertion	C	731	0.88
Array Splitter	D	375	0.96
Integer Array Splitter	D	778	0.97
Overload Names	D	590	0.98
False Refactor	D	936	0.94
Rename Registers	D	936	0.99
Publicize Fields	D	936	0.98
Field Assignment	D	936	0.93
Variable Reassigner	D	296	0.94
Duplicate Registers	D	936	0.98
Boolean Splitter	D	930	0.95
Merge Local Integers	D	934	0.99

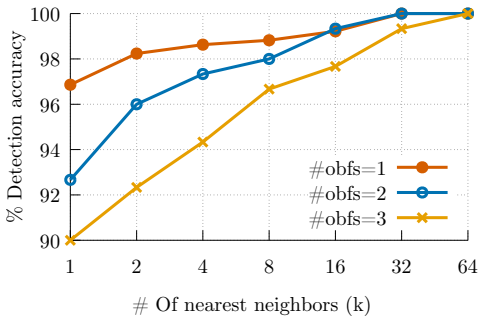

Fig. 5. Detection accuracy of all clones in the CTR-OS dataset against their source counterparts in the CTR dataset.

obfuscators as observed by prior studies [16], [32]. Overall, we produced 450 unique clones constituting the CTR-OS dataset: 150 clones with two obfuscators in serial and 300 clones with three obfuscators in serial.

Figure 5 shows the detection accuracy using one, two, and three obfuscators in serial. For $k = 1$, i.e., *the* nearest neighbor to a clone is its source app, *Dexsim* achieved 96.8%, 92.5% and 90.1% for one, two and three obfuscators in serial, respectively. For $k = 8$, this increases to at least 96% accuracy with three obfuscators in serial. Overall, *Dexsim* showed strong resiliency against serialized obfuscators.

Of all combinations, the lowest average similarity score achieved was 0.84 using the sequence *[Method Merger* $\Rightarrow$ *Opaque Branch Insertion* $\Rightarrow$ *Branch Inverter]*. Nevertheless, the decay in detection accuracy was insignificant, as the detection

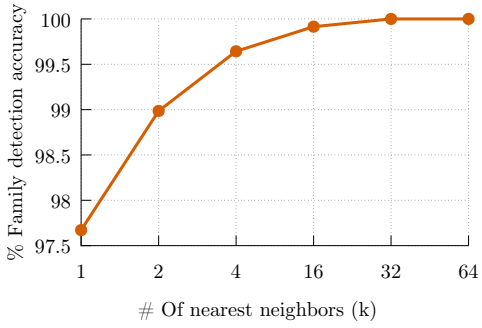

Fig. 6. Family detection accuracy of all malware samples in the MAL dataset.

accuracy was at least 90% for an exact mach, and above 92% for two or more nearest neighbors. In general, *Dexsim* conserved much higher similarity compared to prior work. For instance, AndroGuard achieved only 0.26 to 0.33 similarity when applying three (weaker) obfuscators in serial [16].

C. Malware Variants Identification Results

In this experiment, we indexed two malware samples selected at random from each malware family (35 unique families in total, excluding variants) in the MAL dataset. Then, we ran *Dexsim* in detection mode on the remaining malware samples (test samples). We configured *Dexsim* to output the nearest neighbors list of each test sample and its family label, and computed the detection accuracy as a function of the returned k nearest families as: $100 * (1 - \frac{\text{\#unmatched samples}}{\text{\#all samples}})$, where an unmatched sample is a test malware sample family that was not detected by *Dexsim* within the nearest k neighbors.[7]

Figure 6 shows the results. *Dexsim* achieved at least 97.5% family identification accuracy. For the case where $k = 1$, it flagged 163 malware samples with the wrong label. We manually inspected these samples and found that they were closely related variants of the family labels reported by *Dexsim* (e.g., AndrKongFo-A and AndKongFo-D; AndrKmin-A, AndrKmin-C, and AndrKmin-H). We also observed that for these same variants there was no consensuses among the antivirus products from VirusTotal on the family label.

D. False Positives

We used standard 5-fold cross validation to compute a preliminary estimate of the False Positive Rate (FPR) of *Dexsim*. We split the BEN dataset into five equally-sized portions (600 apps each), used each four-out-of-five portions for indexing and the one left out portion for testing. Some false positives are expected since apps tend to share code, such as SDKs, ad libraries, in-app payment libraries, and apps from the same developer. Overall, the FPR ranged from 0.50% to 2.17%, averaging only 1.20% which is reasonable for vetting purposes and lower than prior solutions (Andarwin 3.72% [6]; ViewDroid 4.70% [32]; DroidSIFT 5.15% [29]).

[7]A better strategy is to decide the family label based on label frequencies of the nearest k neighbors (e.g., majority voting). We opted against this to give a transparent evaluation of the detection power of *Dexsim*.

E. Overhead Results

The total detection time for apps in BEN-O (18,472 incoming apps; 3k indexed apps from BEN) summed up to 2.41 hrs using the maximum tree size ($w = 512$) or less than 6 ms per each incoming-indexed app pairs. The average forced execution time per app was 13 s, and it took 0.71 s on average to construct and save each compression tree. This adds up to about 14 s indexing time per app (11 hrs total for the 3k apps in BEN). The trace window size (w) had a negligible impact on the processing time which is expected since the overall trace length is invariant of the trace window size. Note that the detection time scales linearly with the number of indexed apps, and both indexing and detection times can be directly reduced by distributing the work on multiple servers (a reduction factor of k using k executors). For instance, by extrapolating to 2,000,000 indexed apps, we can estimate the detection time to be about 40 min per app using a typical commercial-grade 80-core server, or only 8 min per app using five servers.

V. Discussion

A. Padding Attacks

Attackers could attempt to evade *Dexsim* by padding the bytecode at the app level or at the component level. For instance, attackers could iteratively merge apps into a single app, till that resulting app evades detection. Likewise, attackers could pad all components in an app with dummy instructions. The idea is to insert enough padding till the resulting app does not compress favorably using any of the probability assignments in *Dexsim*'s database and evading detection. Padding attacks require substantial effort on the part of the attacker to maintain proper file structure and runtime behavior, and we are not aware of any tools to automatically pad or merge Java or Android apps. Nevertheless, it is straightforward to extend *Dexsim* to thwart padding attacks by relaxing the asymmetry of the relative similarity computation performed by *Dexsim* at the expense of an extra $O(n)$ detection time. In Equation (6), $s(b, a) \neq s(a, b)$ for any two apps a and b as $s(b, a)$ computes the *relative similarity* of b to a by computing the compressibility of the traces of b using the dictionary generated from the traces of a. For example, if $b = x\|a$, where x is an attacker-chosen padding and $\|$ is the concatenation operator, then $s(x\|a, a) < s(a, a)$ since it is in the attackers advantage to choose x that is likely to poorly compress using the probability assignments in the dictionary of a. On the other hand, $s(a, x\|a) \approx s(a, a)$ since the dictionary of $x\|a$ also contains the patterns in the traces of a. Hence, a straightforward countermeasure is to instead compute *mutual similarity*, i.e., $s'(b, a) = \min(s(b, a), s(a, b))$, and incur an additional $O(n)$ for detection.

B. HTML Apps and Resource Files

Dexsim cannot execute code embedded in web views (e.g., embedded JavaScript) since the code behind web views is not compiled into the bytecode of the app. This makes it impossible to differentiate between different apps (not clones) that are perhaps implemented entirely in a web view (e.g., HTML apps) and other apps that only serve a single widget (e.g., the majority

TABLE II
COMPARISON OF PRIOR WORK AND *Dexsim*.

Solution Based On	Resilience† T2	T3	L	V	R	O	Detection Time‡	Detection in $n^\star$	Accuracy
String [34]	○	○	○	●	◐	○	$O(s)$	$O(n)$	Low
Hash [2], [4]	○	○	○	◐	◐	○	$O(s)$	$O(n)$	Low
Token [35]	◐	○	○	●	●	○	$O(s)$	$O(n)$	Medium
AST [36], [37]	●	○	○	●	●	○	$O(s)$	$O(n)$	High
PDG [18]–[20]	●	●	◐	●	●	○	$O(v^2m^2)$	$O(n^2)$	High
GUI [11], [32]	●	●	●	○	◐	◐	$O(v^2m^2)$	$O(n^2)$	Medium
Resources [12]–[14]	●	●	●	◐	○	◐	$O(\bar{r}n)$	$O(nlgn)$	Medium
PDG+Hash [6]	●	◐	◐	◐	●	○	$O(\bar{m}n)$	$O(nlgn)$	Medium
CFG [5], [9]	●	●	○	●	●	○	$O(cs)$	$O(n)$	High
Dexsim	●	●	●	●	●	●	$O(cs)$	$O(n)$	High

† L stands for resilience to added or removed libraries. V stands for resilience to view modifications. O stands for data and control flow obfuscation resilience. ● indicates good resilience, ◐ indicates partial resilience, and ○ indicates poor resilience.

‡ Detection time of an incoming app against a database of n indexed apps. c is a small number ($c \ll n$). m is the total number of methods in all apps. $\bar{m}$ is the average number of methods per app. $\bar{r}$ is the average number of resource files per app. v is the number of PDG nodes. s is the total number of statements in all apps.

* Detection time for an incoming app in the number of indexed apps n.

of Android book apps). This also is a known limitation of GUI-based detection solutions as the static GUI in JavaScript and HTML apps is minimal, composed of only a web view and few controls [15], [32]. *Dexsim* does not inspect individual resource files that may be packaged with apps. Resource-similarity solutions (e.g., [12], [13]) are complementary to this work.

VI. RELATED WORK

App clones have been generally categorized into four types [3], [33]: T1) The cloned code has identical code fragments, except for non-code variations such as changes in whitespaces, comments and annotations; T2) In addition to T1, the cloned code contains changes in identifiers, literals and types; T3) In addition to T2, the cloned code has added or removed code fragments; and, T4) In addition to T3, the cloned code is re-implemented through different syntactic variants. T1 clones are considered legacy as they minimally impact an app's bytecode. Control- an data-flow obfuscation fall under T4 clones. In fact, it is arguable that obfuscation is the only realistic T4 cloning technique since attackers typically lack the incentive and resources to fully reverse engineer and re-implement cloned apps at a large scale. Several studies have suggested that T2, T3, and T4 are prevalent on the Android market, posing a serious threat to a healthy ecosystem [2], [3], [5], [10], [11], [15]. Table II summarizes the comparison between *Dexsim* and prior clone detection techniques. Prior techniques were: 1) non-resilient to traditional cloning, obfuscation, libraries, and GUI changes; or 2) non-scalable; or 3) inaccurate.

Google has a service called Google Bouncer for vetting apps submitted to the Google Play market that analyzes each submitted app for approximately five minutes. A study by RiskIQ [38] found that malicious apps in the Google Play store increased 388% between 2011 and 2013, yet the number of apps removed by Bouncer *dropped* from 60% to 23% in the same period. The capacity of Bouncer for detecting cloned apps has not been studied and recent studies found that clones are prevalent across Android markets [5], [10], [15].

Complementary to this work are solutions that detect clones based on GUI similarities. ViewDroid [32] computed the similarity of GUIs across apps using graph isomorphism on the GUI layout. Similarly, [11] detected clones using GUI similarity by encoding the GUI structure as a point in 3D similar to the work of [5] to avoid nonscalability of graph isomorphism. These solutions are based on the hypothesis that an app and its clone are likely to have similar GUI structure. We argue that this assumption is unrealistic against non-naive attackers. The GUI of Android apps is implemented in plain XML and can be easily restructured using equivalent android layout primitives, obfuscated, concealed behind OpenGL surfaces, or dynamically loaded/replaced at runtime, bypassing GUI-based analysis as shown in [14], [39].

Solutions were proposed to detect clones using static and dynamic resource analysis [12]–[14], [39]. The idea is that two apps are likely clones if they share more than some threshold of highly similar resource files (e.g., images and GUI strings). The approaches can work statically by comparing resource files from app packages at rest or dynamically by executing apps on a device or an emulator and capturing resources at runtime (e.g., taking screenshots of rendered views). Similar to GUI-based solutions, static resource-based solutions can be evaded by obfuscating or replacing resource file. Dynamic resource-based solutions, while more resilient to static resource and GUI changes, do not scale as they require actual execution of apps, manual intervention, and can be fingerprinted and bypassed by delaying suspicious behaviors [14], [40].

VII. CONCLUSION AND FUTURE WORK

We introduced *Dexsim*, a scalable and accurate Android bytecode clone detection system highly resilient to control and data obfuscation. Our preliminary results highly suggest the effectiveness and resilience of our approach. In our experiments, a prototype implementation of our approach detected obfuscated clones with at least 98% accuracy. It also achieved more than 97% accuracy for malware family classification. *Dexsim* detected clones within 17 seconds on average, making it suitable for large-scale market vetting.

In a future work, we plan on systemically measuring the capacity of *Dexsim* by injecting foreign code at every possible offset in a sizable app and quantifying the detection accuracy as a function of the injected code length and the injection distance. We also plan to quantify the relationship between forced execution time, path coverage, and detection capacity, which may enable practical assurance guarantees via multi-pass detection by adjusting the execution time based on the desired level of assurance for some apps without impacting the overall average detection time of the system.

ACKNOWLEDGEMENTS

We thank the reviewers for their comments; and Dan Fleck for commenting on an earlier version of this manuscript.

REFERENCES

[1] S. Baker, M. Chau, F. Jeronimo, and K. Kaur, "Smartphone OS market share, 2016 Q3," http://www.idc.com/promo/smartphone-market-share/os, 2016, [Online; accessed 04-Dec-2017].

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

[3] C. Ren, K. Chen, and P. Liu, "Droidmarking: resilient software watermarking for impeding android application repackaging," in *the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 635–646.

[4] 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, Malware, and Vuln. Assessment*. Springer, 2012, pp. 62–81.

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

[6] J. Crussell, C. Gibler, and H. Chen, "Andarwin: Scalable detection of semantically similar android applications," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 182–199.

[7] Y. Shao, X. Luo, C. Qian, P. Zhu, and L. Zhang, "Towards a scalable resource-driven approach for detecting repackaged android applications," in *the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 56–65.

[8] G. Suarez-Tangil, S. K. Dash, M. Ahmadi, J. Kinder, G. Giacinto, and L. Cavallaro, "Droidsieve: Fast and accurate classification of obfuscated android malware," in *5th ACM Conf. Data and Application Security*, vol. 7148. IEEE, 2016, pp. 43–50.

[9] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," *Blackhat*, 2011.

[10] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *European Symp. on Research in Computer Security*. Springer, 2012, pp. 37–54.

[11] 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 *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 659–674. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/chen-kai

[12] O. Gadyatskaya, A.-L. Lezza, and Y. Zhauniarovich, "Evaluation of resource-based app repackaging detection in android," in *Nordic Conference on Secure IT Systems*. Springer, 2016, pp. 135–151.

[13] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina, and E. Moser, "Fsquadra: fast detection of repackaged applications," in *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2014, pp. 130–145.

[14] L. Malisa, K. Kostiainen, M. Och, and S. Capkun, "Mobile application impersonation detection using dynamic user interface extraction," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 217–237.

[15] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting android reuse studies in the context of code obfuscation and library usages," in *the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 242–251.

[16] H. Huang, S. Zhu, P. Liu, and D. Wu, "A framework for evaluating mobile app repackaging detection algorithms," in *Trust and Trustworthy Computing*. Springer, 2013, pp. 169–186.

[17] M. Protsenko and T. Muller, "Pandora applies non-deterministic obfuscation randomly to android," in *Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on*. IEEE, 2013, pp. 59–67.

[18] C. Liu, C. Chen, J. Han, and P. S. Yu, "GPLAG: detection of software plagiarism by program dependence graph analysis," in *12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 872–881.

[19] J. Li and M. D. Ernst, "Cbcd: Cloned buggy code detector," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 310–320.

[20] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Software Engineering. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 321–330.

[21] J. Rissanen, "A universal data compression system," *IEEE Transactions on information theory*, vol. 29, no. 5, pp. 656–664, 1983.

[22] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *Network and Distributed System Security*. The Internet Society, 2016.

[23] B. Gruver, "Smali," https://github.com/JesusFreke/smali, 2009, [Online; accessed 04-Dec-2017].

[24] R. Johnson and A. Stavrou, "Forced-path execution for android applications on x86 platforms," in *7th IEEE Software Security and Reliability*. IEEE, 2013, pp. 188–197.

[25] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX conference on Security Symposium*. USENIX Association, 2014, pp. 829–844.

[26] Y. Cao, Y. Fratantonio, M. Egele, A. Bianchi, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically detecting implicit control flow transitions through the android framework," in *Network and Distributed System Security*. The Internet Society, 2015.

[27] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[28] C. Collberg, G. Myles, and A. Huntwork, "SandMark - a tool for software protection research," *IEEE security & privacy*, vol. 99, no. 4, pp. 40–49, 2003.

[29] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1105–1116.

[30] *VirusShare.com*, https://virusshare.com [Online; accessed 04-Dec-2017].

[31] *VirusTotal*, https://www.virustotal.com [Online; accessed 04-Dec-2017].

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

[33] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[34] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995., 2nd Working Conference on*. IEEE, 1995, pp. 86–95.

[35] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[36] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Int. Conf. on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[37] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Instant code clone search," in *international symposium on Foundations of software engineering*. ACM, 2010, pp. 167–176.

[38] RiskIQ, "RiskIQ reports malicious mobile apps in Google Play have spiked nearly 400 percent," www.riskiq.com/press-release/riskiq-reports-malicious-mobile-apps-google-play-have-spiked-nearly-400/, 2014, [Online; accessed 04-Dec-2017].

[39] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the app is that? deception and countermeasures in the android user interface," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 931–948.

[40] J. Oberheide and C. Miller, "Dissecting the android bouncer," *SummerCon2012, New York*, 2012.