MoonlightBox: Mining Android API Histories for Uncovering Release-time Inconsistencies

Li Li*, Tegawendé F. Bissyandé†, Jacques Klein†
* Faculty of Information Technology, Monash University, Australia
†Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
li.li@monash.edu, {tegawende.bissyande, jacques.klein}@uni.lu

***Abstract*—In most of the approaches aiming at investigating Android apps, the release time of apps is not appropriately taken into account. Through three empirical studies, we demonstrate that the app release time is key for guaranteeing performance. Indeed, not considering time may result in serious threats to the validity of proposed approaches. Unfortunately, even approaches considering time could present some threats to validity when release times are erroneous. Symptoms of such erroneous release times appear in the form of inconsistencies with the APIs leveraged by the app.**

We present a tool called MoonlightBox for uncovering time inconsistencies by inferring the lower bound assembly time of a given app based on the used API lifetime information: any assembly time below this lower bound is considered as manipulated. We further perform several experiments and confirm that 1) over 7% of Android apps are subject to time inconsistency, 2) malicious apps are more likely to be targeted by time inconsistency, compared to benign apps, 3) time inconsistencies are favoured by some specific app lineages. We eventually revisit the three motivating empirical studies, leveraging MoonlightBox to compute a more realistic timeline of apps. The experimental results confirm that time indeed matters. The accuracy of release time is even crucial to achieve precise results.

I. Introduction

In just a decade, Android has grown to be the leading operating system used in a variety of mobile devices ranging from smartphones to home appliances such as TV sets. In this period, the development of apps for the platform has increased steadily: as of April 2017, the official app market, Google Play, was distributing over 2.8 million apps [1]. Several alternative markets also make available a considerable number of apps [2]. Given the impact of Android apps in consumers' daily activities and the openness of its framework, Android has quickly become a hot topic in the software and security research communities. A simple search of the "Android" keyword on Google Scholar lists over 300,000 article entries going back to 2008 when Android was first introduced.

Among the many research directions explored in the literature, some of them explicitly take into account the release time of Android apps. Generally, in software development, the release time corresponds to the time a package is distributed to users. In the Android ecosystem, such time could be matched with the market upload time. Unfortunately, current research datasets, including the AndroZoo continuous stream of apps, do not provide such metadata information. In this context, app assembly time appears to be a valid indicator of release time, since the delay between the assembly and upload time should be negligible since any additional change performed on the code or on app resources will necessarily require a re-assembly. Considering the APK file alone, app assembly time is approximated by checking the *last-modified time* of the main DEX file building from the app source code. This time is then usually considered as the release time and is used in the literature to drive approaches or design experimental validation scenarios. For example, Arash et al. [3] compare app assembly time against the release time of a patched library to find vulnerable apps: they consider a given app to be vulnerable if it contains a vulnerable library and it was not updated after the patched library was released. Allix et al. [4] demonstrate that most state-of-the-art machine learning (ML) based malware detection approaches introduce some bias in their classification when they simply pick a random set of known malware to train the malware detector. They argue that those approaches are not realistic (i.e., not useful in practice for detecting zero-day malware), as they may train on apps collected "in the future" to test apps "from the past". They also empirically show that by correctly taking time into consideration (e.g., training apps "from the past" to test apps "in the future"), ML-based classification results are significantly impacted (in a negative way), comparing to the cases where time is not considered. Overall, all of these examples show that time information is important for Android-related research approaches.

Unfortunately, the assembly time of Android apps is readily manipulable, either through intentional manipulation or due to mistakes in environment settings (e.g., when the development computer system date is reset), resulting in wrong time information for Android apps. In this work, we refer to this manipulation behaviour as **time inconsistency**. Because of time inconsistency, the evaluations of all the aforementioned approaches, which have considered time information as one of their essential parts, will present threats to validity.

In this paper, we propose to mitigate these threats by presenting a research tool called MoonlightBox[1], which attempts to identify the most recent Android APIs that are used by learning from evolution histories of the Android framework base. Given an Android app, MoonlightBox computes its lowest possible assembly time, and uses it to flag apps that are subject to time inconsistency: **apps, which according to**

[1]A precious artefact referenced in the fantasy-comedy film entitled "A Chinese Odyssey". MoonlightBox can open up a time portal, when moonlight shines on it, for its users to travel in time.

their release time, would be calling APIs that do not yet exist since such APIs introduction dates are posterior to the release time.

Although there is no definitive data on the processes through which apps come to present time inconsistency symptoms, we can propose a theory on potential reasons why manipulation can be intentional. Legitimate app developers and malware writers can manipulate app assembly times with different motivations. For example, "opportunist" developers may simply regularly update an app assembly time (without actually changing anything) to pretend that it is actively maintained. After modifying the app code (i.e., DEX file), attackers would like to change the introduction time of the DEX file so as to pretend that the DEX file is unchanged. Similarly, app repackagers may opt to change the assembly time of the repackaged version of an app to make it appear anterior to its original counterpart, ensuring that most detectors will fail to flag the repackaged app (and may instead mis-flag the original as repackaged). In any case, discussions on Q&A sites such as Stack Overflow [5] clearly show that there is a trend in the developer community to investigate the different means available to manipulate the last-modified time of a given zip file such as an Android app package files.

Overall, we make the following contributions:

- Through three empirical studies, including two replication studies on existing approaches (i.e., repackage analysis, ML-based malware detection) and a new research investigation (i.e., app lineage analysis), we demonstrate that time does matter for various Android-related research, i.e., time is an important parameter that should be taken into account.
- We empirically demonstrate that time inconsistency actually exists in Android app markets.
- We present a tool-supported approach called MoonlightBox, a static code analyzer, for mitigating potential threats to the validity of Android research that requires time information to form their approaches.
- We thoroughly investigate the effectiveness of MoonlightBox and its impact on state-of-the-art approaches. We demonstrate that time is not only important for various Android research directions but also sensitive for conducting practical analyses.

II. Time Matters for Android Research

Our objective in this work is to call on an action for Android-related researchers to seriously take app assembly time into consideration, in order to avoid potential research biases and mitigate potential threats to validity of their analyses. To this end, we provide in this section several motivating examples demonstrating the importance of time information for various Android research.

A. ML-based Malware Detection

An increasingly large body of the literature on Android malware detection is leveraging machine learning (ML) techniques to discriminate malicious from benign apps [6] [7] [8] [9]. Unfortunately, state-of-the-art approaches seldom take precautions to avoid the situation where some apps in the training set are historically posterior to apps in the testing set (i.e., learning from the future to test the past). This is known in the field of machine learning as a *data leakage* threat to validity [10]. Recently, a study by Allix et al. [4] has revealed that simply selecting a random set of known malware to train a ML-based malware detector, as it is done to validate most approaches, will yield significantly biased results. Indeed, this random selection may lead to a situation where some items in the training set are actually "from the future" (in terms of creation time) to items in the test set.

To further assess to what extent does app release time matter in ML-based malware detection, we re-visit the ML-based classification conducted in the state-of-the-art MUDFLOW approach by Vitalii et al. [11]. We select this approach because it is the first in the literature that leverages comprehensive (and semantically sound) features to characterize behaviour based on data flow. The authors have further made publicly available all the evaluation artefacts including app samples and their corresponding features so that we do not need to perform the expensive feature extraction step. MUDFLOW automatically identifies malware based on sensitive data flows in Android apps. It implements a one-class classifier on a set of benign apps and applies this classifier on a test set to confirm that an app is benign, otherwise, it is malicious. In the MUDFLOW repository, the dataset contains features for around 2,500 benign apps and 15,000 malware apps. Since the number of samples in the benign set is small and does not allow to experiment with a significant timeline, we propose to perform the one-class classification on the malware class, learning to identify "malicious" data flow. Thus, we consider training on malware samples to detect whether a sample is malicious or not (see discussion in Section VI).

Fig. 1 illustrates the experimental setup of our ML-based malware detection with two experimental testing procedures: Experiment 1 randomly splits the malware set into three subsets without accounting for the release time of test apps; on the other hand, Experiment 2 regroups apps in three sets according to release time. Note that although in the dataset from MUDFLOW repository the majority of apps are old (anterior to 2014), the insights of our investigations should hold for recent samples. We then conduct three runs for each experimental procedures switching the training and testing sets between $S1, S2$ and $S3$: $S1/S2$ indicates that training is using $S1$ while testing is performed on $S2$.

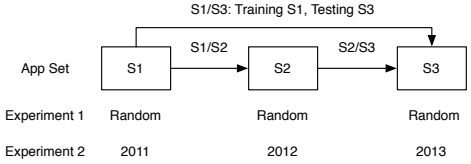

Fig. 1: Experimental Setup for ML-based malware detection.

For each experimental run, we randomly select 500 samples in the training set and 500 samples from the test sets. We repeated this process 100 times to compute a reliable average performance score. Fig. 2 depicts the distribution of classification results (i.e., the number of apps correctly predicted as malware) of our experiments. The median number of correctly predicted apps for $S1/S2$, $S1/S3$, and $S2/S3$ are 206, 242.5, 225.5 (for Experiment 1) and 128.5, 128, 165.5 (for Experiment 2), respectively. We use the Mann-Whitney-Wilcoxon (MWW) statistical test to confirm that the differences between these two experimental results (e.g., between $E1 : S1/S2$ and $E2 : S1/S2$) are indeed statistically significant: the resulting *p-value* confirms that the differences are significant at a significance level[2] of 0.01. We note that, in Experiment 2, when the time is properly taken into account as it should be in the scenario of zero-day[3] malware detection, the performance of the classifiers is actually limited.

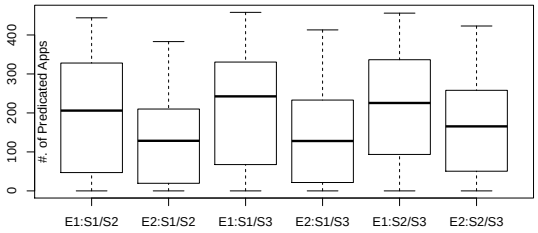

Fig. 2: ML-based malware detection results (E1, E2 stands for Experiment 1 and Experiment 2, respectively. S_i/S_j means that the classifier is trained on S_i and then used to test S_j).

B. Repackaging Pair Construction

Repackaging is a serious threat to the Android ecosystem as it enables plagiarists to redirect profits from legitimate app developers, spreads malware on users' devices, and increases the workload of app maintainers [12], [13]. In less than 10 years, the research around this specific issue has produced a large number of works according to recent literature surveys [14] [15]. The majority of state-of-the-art approaches identify repackaged apps based on similarity computation. Fig. 3 illustrates a typical process of conducting pairwise comparison for a given pair of Android apps. One essential step in this process consists in discriminating the original app from the repackaged one.

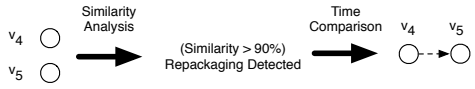

Fig. 3: Example of Pairwise Comparison-based Repackaged App Detection.

The *RepackageRepo* project [16] maintains a repackaging dataset, containing in total 15,296 pairs, available to the Android research community. Each app pair ($\text{App}_{original} \rightarrow \text{App}_{repackaged}$) includes a benign ($\text{App}_{original}$) and a malicious ($\text{App}_{repackaged}$) app. To the best of our knowledge, and according to the description provided by the maintainers, the pairs have been associated without accounting for the implications of their release time. Instead, they leverage the maliciousness status of apps in a pair to determine which one is the repackaged one or the original one. We re-visit this dataset to check the logic of association in each pair: in a pair, a repackaged app must be posterior to its original. By checking the assembly time of both apps for each of the 15,296 pairs, we found that 7,364 pairs (i.e., 48% of app pairs in the *RepackageRepo*) are suspicious pairs since the malicious $\text{App}_{repackaged}$ is released anteriorly to $\text{App}_{original}$.

C. App Lineage Construction

An app lineage is represented as a time-ordered series of some versions of the same app. Since markets only make available the latest version of an app [17], it is often difficult to collect over time the complete lineage of the apps: the rapid updates in Android apps [18] makes it virtually impossible to catch up on some versions after a short break in market watch. Thus, app lineages in research datasets can miss some versions as illustrated in Fig. 4 where the lineage of a given app has missed some version instances.

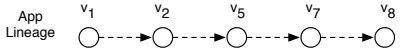

Fig. 4: An App Lineage Example.

.

Nevertheless, even incomplete, app lineages constitute relevant artefacts for understanding the app development process as well as for facilitating advanced incremental analyses based on the *diff* between consecutive versions. These analyses can focus on API update trends, bug/vulnerability fix patterns [19], malicious code injections, etc.

Constructing app lineages is, in theory, straightforward: the *versionCode* attribute in the app's *manifest* file should be configured by developers to reflect the stage of the app evolution. The bigger the versionCode, the most recent the version is. Since every app is supposed to keep a unique app package name [20], we can explore the 5 million apps in Androzoo [2] repository to construct all available app lineages based on the *versionCode* attribute. By only considering lineages that contain at least three apps, we are able to collect 376,344 app lineages. Fig. 5 plots the distribution of those lineages, where the majority of lineages include less than 10 app versions.

Among the collected 376,344 lineages, we have identified 31,464 (8.4%) cases where at least one app in the lineage provides a release time that is inconsistent with the order constructed based on the *versionCode*: i.e., a given app version v_i presents an assembly time that is posterior to the subsequent app version v_{i+1}. This situation raises a validity-related question of the lineages that are obtained without accounting for release time, which in turn may impact various lineage-based analyses.

[2]Given a significance level $\alpha = 0.01$, if *p-value* $< \alpha$, there is one chance in a hundred that the difference between the compared two datasets is due to a coincidence.

[3]zero-day malware are malicious samples that were previously unknown.

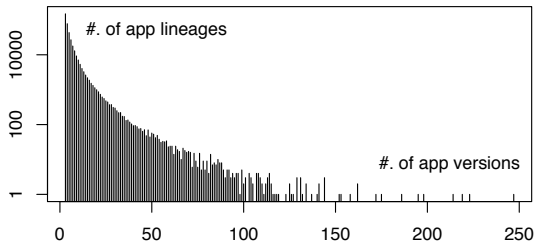

Fig. 5: Distribution lineages following their size.

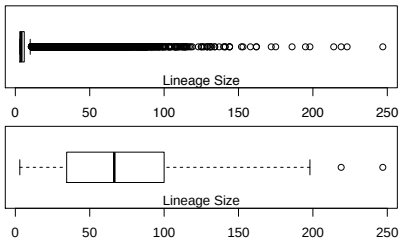

Fig. 6: Distribution of the size of app lineages that are constructed from AndroZoo (above) and are questionable (bottom) because of conflicts between app version code and assembly time.

Fig. 6 further illustrates the distribution of the size of app lineages (i.e., the number of app versions in a lineage): while in general, the median size of a lineage is very small, the median size of lineages that contain versions with suspicious release times is significantly higher than the median size when all constructed lineages are considered. Inconsistencies mostly occur in apps where developers appear to regularly release new versions.

Overall, our empirical investigation of three Android app data-intensive research directions reveals that considering release time can be critical. Failure to take time into consideration may lead to biased approaches and create some threats to validity in performance assessments.

D. Time Inconsistency

Previously described motivating examples to convey the essential message that failing to consider time release of apps can negatively impact the validity of studies and assessments of Android research approaches. We also warn that, even when release time is taken into account, the reported performance may be artificially biased since the app assembly time can be easily manipulated. We refer to the symptom of such manipulation, *whether on purpose or by mistake* (e.g., system clock misconfiguration), as a **time inconsistency**.

Time inconsistencies can be symptomatic of an actual intention to hide the real assembly time so as to deceive app analysts in their time-dependent analysis processes: e.g., given two similar apps, analysts could be tricked into flagging the legitimate version as a repackaged version, when this version displays an earlier release time, leading to invalid conclusions on code changes, etc. It is thus paramount to combat time inconsistency in order to mitigate the potential threats to validity in several Android research directions. Nevertheless, before proposing a solution to address this issue, we investigate a research question (RQ-0) on the extent to which time inconsistency is noteworthy in the Android ecosystem.

RQ-0: *To what extent are real-world Android apps impacted by time inconsistency?*

Towards answering this research question, we leverage the AndroZoo dataset metadata released by the repository maintainers to collect the assembly times of the real-world apps crawled from over 10 markets, including the official Play Store. Figure 7 illustrates the distribution of the assembly time of these apps. We observe that there are 91,328 Android apps created before 2008 (i.e., supposedly before Android was introduced), while 221 apps are assembled after 2017 (i.e., apparently in the future[4]). This finding presents a strong evidence that time inconsistency indeed exists in the Android ecosystem: at least 1.69% ($\frac{91{,}328+221}{5{,}413{,}351}$) of real-world apps do not present a correct assembly time.

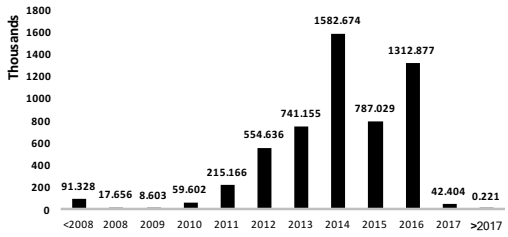

Fig. 7: Assembly times in AndroZoo's app samples.

Given those extreme cases, we suspect that apps may also present an assembly time that is reasonable (e.g., within the range 2012-2016), but which is possibly incorrect. To highlight such cases, we propose, in the next section, a static analysis-based approach called MoonlightBox, which computes a lower bound on the assembly time of a given Android app.

RQ-0 Answer

Time inconsistency does exist in the Android ecosystem. By simply considering assembly times that are unrealistic in practice, because an Android app cannot be built anteriorly to the release of the Android framework or in the future, at least 1.69% of real-world Android apps, collected from common markets and available to the research community, are impacted by time inconsistencies. We expect the actual impact of time inconsistency to be significantly higher, and advocate for an advanced strategy to mitigate the impact of such attacks.

III. MoonlightBox

We propose to address time inconsistencies for mitigating the threats to validity that they bring into Android research directions which heavily depend on app assembly time.

[4]The dataset we used are collected from AndroZoo at 2017

Fig. 8 depicts the working process of MoonlightBox, our approach to deal with time inconsistency. This process mainly unfolds in three steps:

1) API Time Mapping (ATM) step, where we harvest the introduction time of Android APIs and form a mapping that takes as input an API and outputs the introduction time of that API.
2) API Usage Modelling (AUM) step, where we harvest all the Android APIs that are leveraged by a given app.
3) APK Time Prediction (ATP) step, where we compute the lower bound in assembly time of a given app by matching information collected from the previous two steps.

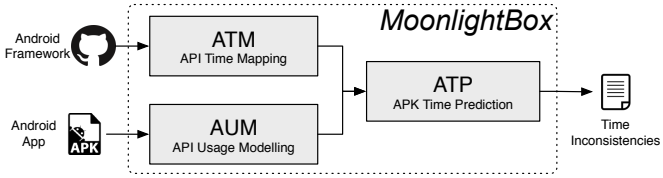

Fig. 8: The working process of MoonlightBox.

A. API Time Mapping (ATM)

In this step, our aim is to build a reliable and reusable mapping to enable our approach, and future research works, to query the introduction time of a given Android API. To this end, we rely on the historical information of the Android framework code base in GitHub [21]. This repository is suitable as it has recorded all the history commits (with API level tags) of the code changes including the introduction of new APIs. Because there are many redundant tags as well as irrelevant tags (i.e., they do not necessarily come with new APIs), we use a selection strategy to only consider the most relevant tags:

- If several tags are committed at the same time, only the one with the smallest version number is considered.
- If two subsequent tags are committed without adding new APIs or deleting existing APIs, only the former one (with smaller version number) is considered.

Overall, our approach has considered 79 tags out of over 200 tags that were initially available.

Algorithm 1 briefly summarizes the working process of the ATM step. In the beginning, it attempts to load, in order to directly update, a previously constructed API time map (line 2). If such map does not yet exist, MoonlightBox initializes an empty map, then traverses the code of all the new tags (i.e., all the new Android framework versions) found in the code base repository. For each tag, MoonlightBox extracts all defined Android API signatures (line 7) and, records (line 10) or updates (line 13) them into the API time map. Eventually, the map contains all APIs (identified based on their unique full signature) associated with their introduction time.

As the algorithm shows, the ATM process for building the API time mapping is in general rather straightforward. Nevertheless, polymorphism feature of the Java language, which

Algorithm 1 API Time Mapping Construction.

```
1:  procedure BUILDAPITIMEMAPPING(newTags)
2:      apiTimeMap = load()
3:      if apiTimeMap == null then
4:          apiTimeMap ← {}
5:      end if
6:      for each t ∈ newTags do
7:          apis ← associatedAPIs(t)
8:          for each api ∈ apis do
9:              if api ∉ apiTimeMap.keys then
10:                 apiTimeMap.put(api, t.time)
11:             else
12:                 if apiTimeMap.get(api) > t.time then
13:                     apiTimeMap.put(api, t.time)
14:                 end if
15:             end if
16:         end for
17:     end for
18:     removeOverridedAPIs(apiTimeMap)
19:     return apiTimeMap
20: end procedure
```

Android has inherited, creates situations where API introduction time is fluctuating depending on the position in the class hierarchy. Consider Listing 1 for example: API *writeToParcel()* was introduced in 2009 and has been overridden by class *MediaDescription* in 2014. Naively, the API time mapping should record that the API defined in *MediaDescription* is available only starting in 2014. Yet an app developed in 2013 could have used this API in a *MediaDescription* object. Thus, a naive approach could report a false positive time inconsistency by considering that the lower bound for the API is 2014. To make our approach more conservative, we overcome the threats of such cases by 1) first identifying all polymorphism-affected APIs in the Android codebase and 2) remapping all such APIs with the corresponding earliest introduction time. Overall, we have empirically found 2,227 polymorphism-affected API cases, including 1,849 cases where an API is later extended by other classes (like the one shown in Listing 1) and 378 cases where an API is introduced to one of its ancestor classes (e.g., API *size* of class *Bundle* is introduced in 2009 while the same API is further defined by class *BaseBundle* in 2014, where *BaseBundle* is a superclass of *Bundle*).

```
1  public interface Parcelable {
2    //Introduced at 2009-09-02 22:39:46
3    public void writeToParcel(Parcel dest, int flags);
4  }
5  public class MediaDescription implements Parcelable {
6    //Introduced at 2014-10-28 04:29:57
7    @Override
8    public void writeToParcel(Parcel dest, int flags) {
9      dest.writeString(mMediaId);
10     ... ...
11     dest.writeBundle(mExtras);
12 }}
```

Listing 1: A challenge raised by polymorphism.

We remind the readers that the API time map constructed in this step can be used independently and thus is reusable for other approaches and tools.

B. API Usage Modelling (AUM)

The goal of this step is to identify all API methods that are called in the code of a given Android app. Since, in general, the source code of market apps is not publicly available, MoonlightBox focuses purely on app bytecode. A common challenge in Android app analysis lies also in the increasing use of dynamic code loading by app developers: analyzers must account for potential API usages in extra code outside the main *classes.dex* code. Consequently, the API usage modelling step should also locate any to-be dynamically loaded code shipped[5] with the APK, to extract relevant API usages. We build on the heuristics proposed by Li et al. [22] to locate the to-be dynamically loaded code: Given an app a, we unzip it and visit all its embedded files, if it is a DEX file through its magic number (035)[6]. If a given visited file is an archive file (e.g., zip), we recursively look into it and check if it contains DEX files. All the identified DEX files, along with the main *classes.dex* file, are then considered for harvesting APIs.

Eventually, we rely on the Soot [23] analysis framework to parse the whole located code, visiting all the statements to identify calls into framework[7] APIs. For each call, this module attempts to match the whole signature, including its declared class, return type, parameter list. Additionally, it also attempts to identify the usage of such APIs that involve in Varargs and generic types.

C. APK Time Prediction (ATP)

Given an Android app a, the constructed API time map (denoted $apiTimeMap$), and the inferred API usage model (denoted $\mathcal{A}$), MoonlightBox implements a third step, ATP, for API Time Prediction, which computes the lower bound t of a's assembly time. t is computed based on the following formula:

$$t = max(apiTimeMap.get(api_i), api_i \in \mathcal{A})$$

Let us consider a given app app_x using two API methods m_1 (introduced in the SDK in January 2013) and m_2 (introduced in the SDK in March 2015). Since app_x cannot use API methods that have not been introduced yet, we are sure that app_x would not have been assembled before March 2015. As a result, the lowest possible assembly time of app_x is the latest introduction time of all API methods used by app_x (i.e., March 2015 in this example).

Note that since t is a lower bound computed by MoonlightBox, the actual assembly time could differ by a few months. Indeed, as shown by McDonnell et al. [24], there is generally several months delay before developers start to adopt newly introduced APIs. On the other hand, once we find, for a given app, that the lower bound t is bigger than the actual assembly time, we can confirm that this app is affected by a time inconsistency.

[5]Some additional code could be downloaded at runtime from a remote server.

[6]Magic number defined in ISO-8859-1 is used to decide the type of a file.

[7]Thanks to the ATM step, we have the list of framework API methods.

IV. Evaluation

Our evaluation addresses the following research questions:

- RQ-1: To what extent are real-world Android apps impacted by time inconsistencies based on the lower time bound obtained by MoonlightBox?
- RQ-2: Are time inconsistencies prevalent in similar proportions between malicious and benign apps?
- RQ-3: Are time inconsistencies recurrent in specific app lineages (i.e., apps from the same developers) or do they occur equally in all lineages?

A. RQ-1: Impact Rate

To answer RQ-1, we randomly select 10,000 benign apps crawled from Google Play store and apply MoonlightBox to check whether their assembly time is consistent with the inferred lower bound. Beforehand, following the criteria tested in Section II-D, we immediately flag 163 apps as presenting incorrect assembly time (i.e., before Android creation or in the future).

By using MoonlightBox, we have further identified 558 apps that are involved in time inconsistencies: those apps are presented as using Android APIs that were not yet available in the SDK at the time of their release. Overall, we have identified more than 7% of the apps ($163+558 = 721$ or 7.21%) subject to time inconsistency.

As an example, the assembly time of the app *com.air.launcher* indicates that the app has been released around 2011-11-09. However, by looking into its accessed APIs, MoonlightBox reports the app code actually accesses API method *getDescription()* of class *android.os.storage.StorageVolume*, which was only introduced on 2016-07-14 (1,708 days later): this suggests that a time inconsistency was likely performed.

Table I summarizes the top-10 accessed APIs for the 721 apps (163+558) whose introduction times contributed to define the lower bound. We further found that the API methods enumerated in this table are, each in the associated release of the framework, among the most adopted by app developers. Among other APIs that are released at the same time, those APIs present the ones that are most favoured by Android developers. We note that these APIs are either related to accessing sensitive data (i.e., permission-specific) or about user interface (i.e., layout), which suggests that, despite developers attempting to manipulate the assembly time, they keep their apps up-to-date with regards to system capabilities in terms of sensitive data access and user interface improvement.

We further consider the 721 identified apps involving time inconsistency and investigate whether some specific time slots are favoured by attackers. We find that the majority (over 75%) of assembly times are distinct from each other, suggesting a random selection of inconsistency time. As reported on Table II, we only find nine assembly time cases that are each associated with more than 1 app. In the noteworthy case of assembly time *1979-11-30 00:00:00* associated with 157 apps, it actually relates to an issue raised by app developers about default configurations [25].

TABLE I: Top 10 Accessed APIs that Contribute to the Identification of Lower Bound Time of Android Apps.

API	Count
android.app.Activity: void requestPermissions(String[],int)	2536
android.app.Activity: boolean shouldShowRequestPermissionRationale(String)	2396
android.widget.PopupWindow: void setWindowLayoutType(int)	2381
android.widget.PopupWindow: void setOverlapAnchor(boolean)	2363
android.widget.CompoundButton: Drawable getButtonDrawable()	2327
android.graphics.drawable.Drawable: boolean setLayoutDirection(int)	2294
android.graphics.drawable.Drawable: int getLayoutDirection()	2244
android.content.Context: int getColor(int)	2200
android.app.AppOpsManager: int noteProxyOp(String,String)	2178
android.app.AppOpsManager: String permissionToOp(String)	2178

TABLE II: Nine Assembly (Inconsistent) Time that appear in at least Two Infected Apps.

Time	Count	Time	Count
1979-11-30 00:00:00	157	2013-12-16 17:01:16	7
2011-11-22 03:02:34	6	2008-02-29 10:33:46	5
2014-09-14 17:21:42	3	2008-02-29 03:33:46	2
2011-11-09 19:50:26	2	2012-03-03 22:09:04	2
2013-12-16 09:01:16	2		

We take the opportunity of this study to investigate delays with which new APIs are adopted by app developers. We compute the delay based on apps that are not flagged as infected by time inconsistency, following the formula:

$$delay = dexTime - lowerBoundTime$$

Fig 9 illustrates the delays distribution. Over half of the considered apps have a delay of over 317 days (close to one year). Some outlier delays can reach up to 1,000 days, suggesting potential time inconsistency that is not explored in this study: those are indeed related to a potential upper bound usage of APIs (i.e., when API methods cannot possibly be used any more in an app).

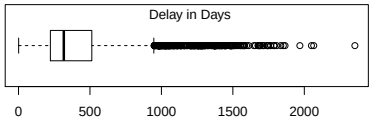

Fig. 9: Distribution on the Delays (in days) of Leveraging New Introduced APIs.

RQ-1 Answer

Answer to **RQ-1:** Over 7% of randomly selected real-world benign apps are infected by time inconsistencies, which is alarming, since, translated in terms of absolute number on GooglePlay, about 210,000 apps (7% of 3 million) have an inconsistent releasing time. Additionally, this study allowed to show that APIs that access sensitive data or leverage advanced user interface capabilities are more rapidly adopted by Android developers.

B. RQ-2: Malicious Vs. Benign

To answer RQ-2, we randomly select 10,000 malicious apps from AndroZoo [2] and investigate the proportion of time inconsistency infected apps, in comparison with the previous impact study on benign apps. After applying MoonlightBox on those malicious apps, based on the lower bound times inferred by MoonlightBox, we identify 1,396 apps that present suspicious assembly times. This number is significantly higher than the number of infected benign apps. As it is commonly known in the Android security community [26], the majority of malicious apps are built by repackaging benign apps. In order to hide such repackaging traits which could be detected by checking their similarity with previous (in terms of release time) apps, attackers may artificially manipulate the app assembly time.

TABLE III: The 75 app lineages associated to "com.sinosoft".

Lineage	Versions	Infected	Lineage	Versions	Infected
com.sinosoft.chahao901	3	3	com.sinosoft.chahao	26	26
com.sinosoft.sinafinance	24	24	com.sinosoft.bubujingxin	6	6
com.sinosoft.jddxylk	25	25	com.sinosoft.sxwqn	11	11
com.sinosoft.starpic	21	21	com.sinosoft.sjsxwqn	25	25
com.sinosoft.baidukanshu	25	25	com.sinosoft.star	19	19
com.sinosoft.egun	4	4	com.sinosoft.raokoulin	23	23
com.sinosoft.naoliceshi	4	4	com.sinosoft.zhougong	26	26
com.sinosoft.renpinjisuanqi	5	5	com.sinosoft.chayoubian	20	20
com.sinosoft.shoudiantong910	7	7	com.sinosoft.xiaohuajizhongying	4	4
com.sinosoft.wsty	5	5	com.sinosoft.ybgjm	5	5
com.sinosoft.chakuaidi	19	19	com.sinosoft.xieshen	7	7
com.sinosoft.meituitianxia908	3	3	com.sinosoft.njjzw	4	4
com.sinosoft.yushi	26	26	com.sinosoft.wdwmny	4	4
com.sinosoft.sinahouse	21	21	com.sinosoft.zhen	4	4
com.sinosoft.liuying	5	5	com.sinosoft.lieche	25	25
com.sinosoft.compass	22	22	com.sinosoft.lieren	8	8
com.sinosoft.fengliusanguo	5	5	com.sinosoft.mimi	21	21
com.sinosoft.mwgs	7	7	com.sinosoft.xuezu	5	5
com.sinosoft.nba	22	22	com.sinosoft.llk	16	16
com.sinosoft.wdlpsmd	5	5	com.sinosoft.momeinv	4	4
com.sinosoft.chatianqi	24	24	com.sinosoft.emnqyd	8	8
com.sinosoft.xiaosangansidui	3	3	com.sinosoft.water	20	20
com.sinosoft.huanshou1	6	6	com.sinosoft.hxzcdxmq	6	6
com.sinosoft.chaip	17	17	com.sinosoft.guichuideng	5	5
com.sinosoft.saolei910	7	7	com.sinosoft.wangyou	7	7
com.sinosoft.xingganchemo907	4	4	com.sinosoft.crazyybp	25	25
com.sinosoft.xingzuoyunshi	18	18	com.sinosoft.qipaomeini908	4	4
com.sinosoft.naojinjizhuanwan	21	21	com.sinosoft.duanxinwzw	18	18
com.sinosoft.chashouji	23	23	com.sinosoft.xiaohuatiankong	9	9
com.sinosoft.bxwwylk	21	21	com.sinosoft.chagongjiao	27	27
com.sinosoft.renxing	5	5	com.sinosoft.chashenfenzheng	22	22
com.sinosoft.huli	6	6	com.sinosoft.shenrimima	19	19
com.sinosoft.baoxiaowangwen	16	16	com.sinosoft.jpjs	7	7
com.sinosoft.miyu	4	4	com.sinosoft.news	22	22
com.sinosoft.shenxiaoyunshi	17	17	com.sinosoft.zhuxian	7	7
com.sinosoft.baigu	5	5	com.sinosoft.ntsj	5	5
com.sinosoft.lishi	25	25	com.sinosoft.qingchenghuangfei	5	**4**
com.sinosoft.xiaohua	42	**41**			

Fig. 10 illustrates the distribution of delays for adopting new APIs in malware and benign apps. We observe that, overall, the delays have fewer variations across benign apps than across malicious apps, and malware presents longer delays than benign apps. The statistical significance of the difference of median delay values suggests that malware apps less often use up-to-date APIs.

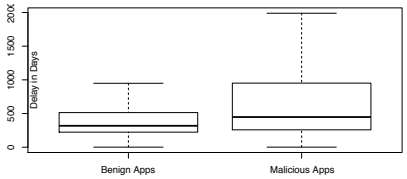

Fig. 10: Delays between benign and malicious apps.

RQ-2 Answer

Answer to **RQ-2:** Malicious apps are more likely to be affected by time inconsistency, compared to benign apps. They also have longer delays in terms of adopting new released Android APIs.

C. RQ-3: Specific App Lineages

RQ-3 investigates whether time inconsistencies are favoured in some specific app lineages. To this end, we randomly select 5,000 app lineages from the lineage set we have built previously (cf. Section II-C). Application of MoonlightBox reveals that 842 (out of the 5,000) lineages contain at least one app version that is affected by time inconsistency. Over half of those lineages have at least two app versions that are affected. This suggests that once an app version is affected by time inconsistency, there is more than 50% probability that there are another one of its peers in a lineage that is affected. We also found 153 app lineages where, in each, all of their app versions are affected by time inconsistency. As an example, 75 app lineages[8] are associated with a single company using "com.sinosoft" as the package name. Table III enumerates those app lineages, along with the number of app versions inside and the number of apps being infected by time inconsistencies. Among the 75 app lineages, 73 of them have all of their apps been infected by time inconsistency. Overall, there are in total 1,001 app versions developed by "com.sinosoft" with 999 apps being affected by time inconsistency. Towards understanding why those apps are attacked[9], our further investigation reveals that 971 (92%) apps are actually malicious, thanks to VirusTotal's reports. This finding confirms to our previous finding showing that malicious apps are more likely to be affected by time inconsistency, which also suggests a potential implication of our approach: Given an app lineage, if most of their app versions are infected by time inconsistency, those apps in the lineage will likely be malicious apps.

RQ-3 Answer

Answer to **RQ-3:** Time inconsistencies are favored by some specific app lineages, where their developers are likely to manipulate the assembly time of their developed apps.

V. Implications

In Section II, we have introduced three motivating examples showing that time information is important for various Android research directions. We now revisit these studies by replacing app assembly time with the lower bound time computed by MoonlightBox.

A. Performing Realistic ML-based Classifications

ML-based malware detection approaches that aim at detecting zero-day malware should be historically coherent, i.e., the training set should only contain apps preceding the ones of the testing set. Indeed, realistic settings would assume that it is impossible to learn from the future. Since the app assembly time can be manipulated, experiments based on this time could still be subject to threats to validity. To further demonstrate this claim, we replicate the second experiment described in Section II-A whose initial results are presented in Fig. 2. In this new experiment, referred to as experiment 3, we keep the same settings as experiment 2 except that now we align the timelines for $S1$, $S2$ and $S3$ based on the lower bound time computed by MoonlightBox instead of the app assembly time as previously done in experiment 2.

Table IV provides the classification results in terms of detection accuracy for the three experimental settings. Overall, we note that the performance drops when we get the timelines to be closer to the reality of app release time. The differences of performance can be explained by the fact that training samples in experiment 2 may not be "accurate", because of time inconsistencies, in the sense that some apps in the training set may still be misplaced since they are from the future and thus create a data leakage problem. Nevertheless, assembly time remains a close approximation of release time since the result of experiment 2 is only slightly better than experiment 3 but much worse than experiment 1. The reason why the result of experiment 3 is worse than experiment 2 is that inconsistency in assembly time introduces biases in training classifiers: training data contains future knowledge, leading to artificial performance (just like a blind random sampling of experiment 1).

Our experimental setting highlights the potential impact of dealing with time-inconsistent datasets. Using MoonlightBox to correct time alignment creates realistic settings that require malware detection researchers to investigate new features to improve performance in the wild. This experiment results further suggest that Android malware by themselves evolve fast. It is hard to predict new types of malware by only learning features from known malware. This phenomenon probably explains why there are already hundreds of ML-based malware detectors presented in research but barely any of them is applied in practice.

TABLE IV: ML-based malware detection results (accuracy in median).

Experiments	S1/S2	S2/S3	S1/S3
Experiment 1 (random)	41.2%	48.4%	45.1%
Experiment 2 (assembly time)	25.7%	25.6%	33.1%
Experiment 3 (API-based lower bound time)	5.4%	20%	25.3%

B. Validating Repackaging Datasets

In Section II-B, we have empirically shown that the *RepackageRepo* project contains benchmark data on repackaging pairs that may actually be of unreliable quality (half of the collected pairs involve an app tagged as repackaged but which was actually released before the original app according to assembly time). Since this benchmark will be relied upon in the community, we propose to properly assess it based on MoonlightBox's lower bound time.

Table V summarizes the verification results. Previously (based on app assembly time), we have shown that 7,364 pairs could be questioned (because the repackaged app appears to

[8]Note that 75 is the number we have obtained from the AndroZoo project. In reality, the company could have developed more apps.

[9]We actually planned to achieve this purpose by contacting the app developer. Unfortunately, we cannot do that because we could not find the developer's contact information, which could be justified by the fact that most of those apps are actually malware.

be more recent than the original app), accounting for 48% of the total pairs. However, based on the time information returned by MoonlightBox, it appears that 15,218, i.e., most of the repackaged apps, are not released before their original counterpart. Concretely 7,328 (99.5%) out of the 7,364 "questionable" pairs are proven to be false alarms in the previous investigation (i.e., those apps are simply suffering from time inconsistencies). Similarly, we find that some repackaged pairs which appeared to be correctly aligned in the timeline were spotted by MoonlightBox. We enumerated only 42 of such pairs which we report as false negatives. These results show that time inconsistencies can cause both false positive and false negative results for building repackaging app pairs.

TABLE V: Release time investigation for all the pairs available in the *RepackageRepo* project.

Pairs	App Assembly Time	Lower Bound Time
Normal	7,932 (51.9%)	15,218 (99.5%)
Questionable	7,364 (48.1%)	78 (0.5%)
Total	15,296 (100%)	15,296 (100%)

To increase confidence in the benchmark in *RepackageRepo*, we have informed its authors and provided them with information to share with potential users.

C. Building Reliable App Lineages

We now revisit, with MoonlightBox, the study presented in Section II-C on the re-construction of app lineages. Since (1) we expected to face scalability issues[10] in applying MoonlightBox on all the 5 million apps available in the AndroZoo project and (2) because our focus is more on the time validity of two successive versions of an app (the assembly time of v_i should precede the assembly time of v_{i+1}) rather than on the time validity of a whole lineage, we opt to randomly select 30,000 app pairs (app_{v_x}, app_{v_y}) where app_{v_x} and app_{v_y} are two successive versions of the same app.

First, by investigating the assembly time of each pair, we found exactly 29,000 pairs which seem "normal", i.e., the assembly time of app_{v_x} precedes the one of app_{v_y}. The remaining 1,000 pairs are however "questionable", i.e., the assembly time of app_{v_x} is posterior to the one of app_{v_y}.

Then, we compute with MoonlightBox the lower bound time of each app in the pairs, and we compare the "time validity" of each pair with the one obtained by using the assembly time. Table VI summarizes the experimental results. 481 pairs among the initial 29,000 "normal" pairs are now flagged as questionable: these are thus false negatives of the approach with assembly time. In contrast, 966 pairs included in the initial 1,000 questionable pairs are now flagged as normal: these are thus false positives of the assembly time-based approach.

[10]In practice, it is challenging to collect all the apps provided by the Androzoo maintainers, because of network bandwidth and storage constraints.

TABLE VI: Release time investigation for all 30,000 randomly selected pairs of subsequent app versions from 7,445 app lineages.

Pairs	App Assembly Time	Lower Bound Time
Normal	29,000 (96.7%)	29,485 (98.3%)
Questionable	1000 (3.3%)	515 (1.7%)
Total	30,000 (100%)	30,000 (100%)

Overall, these revisiting studies empirically show that time indeed matters for various Android-related research. However, the accuracy of the obtained time information is even crucial for the final results.

VI. Threats to Validity and Discussion

The first threat to the validity of our approach lies in the selection of the datasets. The apps collected from Androzoo may not be representative. This threat is mitigated by the fact that Androzoo is the largest and most up-to-date research dataset in the Android community. Furthermore, we have randomly sampled those apps, the large majority being available in Google Play.

App assembly time may not be perfectly accurate to represent the app releasing time. Ideally, we should obtain the release time (along with other relevant metadata) from app markets. Unfortunately, in practice, it is only possible (with substantial engineering work and large, dedicated computing infrastructure) to collect such information for apps that are currently present in markets. Nevertheless, it is virtually impossible to retrieve such metadata for previous app versions, mainly because these metadata are overwritten with data of updated app versions. To the best of our knowledge, no significant dataset, including AndroZoo, is providing app versions with release metadata.

MoonlightBox is strictly focused on inferring the lower bound release time of an Android app. There may exist cases where time inconsistency is performed by replacing the actual date with a reasonably more recent date. Detecting such inconsistencies would involve inferring the upper bound release time of an app. Nevertheless, given that app developers generally use recent APIs to keep up with the latest OS features, manipulation distances in such cases are short, thus leading to limited impact. MoonlightBox thus focuses on inconsistencies going beyond the lower bound. We hope our work will stimulate new interests in our community for more contributions to addressing the problem. We also plan to investigate this aspect with an heuristic-based approach in future work.

Based on the lower bound release time computed by MoonlightBox, we have experimentally demonstrated the existence of time inconsistencies in the Android ecosystem. Although we have seen evidence in developer Q&A sites that some developers are interested in learning how to perform time manipulation, we do not have further concrete evidence on the practical benefits of these attacks beyond the motivating examples on biasing research approaches. It could also be that most cases of time inconsistencies are actually developer mistakes. Nevertheless, our findings clearly indicate that malware

(including repackaged apps) are more likely to be affected by time inconsistency, suggesting a correlation with malicious intent.

Finally, for the motivating example on ML-based classification, we have opted to change the scenario of train/test used in the original MUDFLOW [27] paper given the limited set of benign apps provided in the paper repository which could not allow to properly highlight the timeline issue. We did not attempt to build a reliable dataset but simply reused the one provided by MUDFLOW, which could be the reason why our ML-based approach achieves low accuracy. In any case, the objective of this paper was not to focus on the performance of the MUDFLOW approach, but rather the performance gap caused by a manipulation of the timeline in dataset construction.

VII. Related Work

To the best of our knowledge, we are the first to empirically investigate, characterize and combat time inconsistencies in the Android ecosystem. Even if some works do consider time information (e.g., the last-modified time of the main DEX file building from the app source code) in their approach, they never question the validity of their approaches impacted by the time considered.

There are several studies proposed in the literature to devise reliable experimental protocols [28] [29] [30]. As argued by Blackburn et al. [28], an unsound empirical finding may misdirect a whole field, encourage the pursuit of unworthy ideas. Pieterse et al. [30] argue that researchers should ensure a rigorous design of their experiments and apply software performance measurements techniques to guarantee reliable results, so as to support the repeatability of their experiments, comparability of their reported results, and verifiability of their claims. The findings of those studies help to ensure that research directions and results are in line with practices. Our work follows the same objectives, aiming to highlight the importance of building a reliable experimental setting and thus to make the results more useful for real-world problems and to help individuals conduct better science and encourage a cultural shift in our community to identify and promulgate sound claims.

Similar to our work, other works such as the one done by McDonnell et al. [24] and the ones completed by Li et al. [31], [32], [33] have also attempted to map API calls to their exact releases so as to resolve various API-related issues, e.g., study the stability and adoption of Android APIs [24]. Allix et al. [4] have experimentally discussed the importance of considering history information (i.e., time) for ML-based malware detection of Android apps. As shown in their study, most state-of-the-art assessment scenarios in the literature of ML-based malware detection simply pick a random set of known malware to train a malware classifier. This setup yields significantly biased results [34] [35] [36] when the objective is to detect zero-day malware, where by definition, malware is not known yet. To be realistic, in such scenarios the training set should only contain apps that precede the ones of the testing set. Even if Allix et al. have demonstrated that time matters for ML-based malware detection, their experimental evaluation suffers from the threat to validity since their training/test sets are built based on the app assembly time (DEX file time), which, as shown in this work, is subject to time inconsistencies. MoonlightBox can be leveraged to complement their work by ensuring a more accurate experimental setting leading to more reliable experimental results.

Many related works focusing on Android app repackaging detection such as [37], [38] and [39] have distinguished repackaged apps from original ones. However, those works do not take time information into consideration, resulting in potential threats to the validity of their approaches. As an example, Li et al. [38] have collected a benchmark of repackaged Android app pairs without taking time information into account. The original and repackaged apps are actually distinguished based on their malicious status. As confirmed by our replication study described in Section V, this kind of benchmark is questionable since it can contain app pair for which the repackaged app has been created before the birth of its original counterpart. This problem would impact empirical findings leveraging such benchmark but also other works that are performed on similar benchmarks (e.g., FSquaDRA2 [40]).

Some works such as [41] [42] [43] explicitly mention that their purpose is to detect repackaging Android app pairs but not to identify which is the original one and which is the cloned one. In order to further leverage the results reported by their approaches, one has to come up with a reliable approach to distinguish between the original and repackaged ones. MoonlightBox can actually be leveraged to partially support this purpose. The advantage of applying MoonlightBox in this setting is that the results (e.g., which one is repackaged?) yielded by MoonlightBox will hardly be wrong, although it is not capable of handling all the possible cases, (e.g., some repackaged apps only make small changes to their original counterparts, remaining same lower bound time between the compared two apps).

VIII. Conclusion

In this paper, we first experimentally demonstrated that omitting to consider release time (approximated by app assembly time) may lead to biased approaches and introduce threats to validity in performance assessments. However, we also showed that assembly time is often affected by a new symptom, namely time inconsistency. To overcome this symptom, we proposed a prototype tool name MoonlightBox which could be used by researchers and practitioners to mitigate potential threats to validity in the performance of their assessments. We investigated the extent to which MoonlightBox can help break down insights on time inconsistency, using a large number of real Android apps.

IX. Acknowledgments

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects CHARACTERIZE C17/IS/11693861 and Recommend C15/IS/10449467.

REFERENCES

[1] App Brain. Number of available android applications. http://www.appbrain.com/stats/number-of-android-apps. Accessed: 2018-01-10.

[2] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community. *arXiv preprint arXiv:1709.05281*, 2017.

[3] Arash Alavi, Alan Quach, Hang Zhang, Bryan Marsh, Farhan Ul Haq, Zhiyun Qian, Long Lu, and Rajiv Gupta. Where is the weakest link? a study on security discrepancies between android apps and their website counterparts. In *International Conference on Passive and Active Network Measurement*, pages 100–112. Springer, 2017.

[4] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Are your training datasets yet relevant? In *International Symposium on Engineering Secure Software and Systems*, pages 51–67. Springer, 2015.

[5] Change the last-modified time of a zip file elements. http://stackoverflow.com/questions/ 23499213/change-the-last-modified-time-of-a-zip-file-elements. Accessed: 2018-01-10.

[6] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *Intelligence and security informatics conference (eisic), 2012 european*, pages 141–147. IEEE, 2012.

[7] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 377–388. ACM, 2016.

[8] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, and Yang Liu. A multi-view context-aware approach to android malware detection and malicious code localization. *arXiv preprint arXiv:1704.01759*, 2017.

[9] Xinli Yang, David Lo, Li Li, Xin Xia, Tegawendé F Bissyandé, and Jacques Klein. Characterizing malicious android apps by mining topic-specific data flow signatures. *Information and Software Technology*, 2017.

[10] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):15, 2012.

[11] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *International Conference on Software Engineering (ICSE)*, 2015.

[12] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. On locating malicious code in piggybacked android apps. *Journal of Computer Science and Technology*, 2017.

[13] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Simidroid: Identifying and explaining similarities in android apps. In *The 16th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom 2017)*, 2017.

[14] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, et al. A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 2016.

[15] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 2017.

[16] Repackagerepo: A repository of repackaged android apps. https://github.com/serval-snt-uni-lu/RepackageRepo. Accessed: 2018-01-10.

[17] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play? a large-scale empirical study. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[18] Stuart McIlroy, Nasir Ali, and Ahmed E Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empirical Software Engineering*, 21(3):1346–1370, 2016.

[19] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. On vulnerability evolution in android apps. In *The 40th International Conference on Software Engineering, Poster Track (ICSE 2018)*, 2018.

[20] Set the application id. https://developer.android.com/studio/build/application-id.html. Accessed: 2018-01-10.

[21] platform frameworks base. https://github.com/android/platform_frameworks_base. Accessed: 2018-01-10.

[22] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA)*, 2016.

[23] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *CETUS*, 2011.

[24] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *29th IEEE International Conference on Software Maintenance (ICSM)*, pages 70–79. IEEE, 2013.

[25] Last modified timestamp on all files in apk default to fri, nov 30 1979 00:00:00. https://issuetracker.google.com/issues/37116029. Accessed: 2018-01-10.

[26] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[27] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. 2014.

[28] Stephen M Blackburn, Amer Diwan, Matthias Hauswirth, Peter F Sweeney, José Nelson Amaral, Tim Brecht, Lubomir Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, et al. The truth, the whole truth, and nothing but the truth: a pragmatic guide to assessing empirical evaluations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(4):15, 2016.

[29] Sid-Ahmed-Ali Touati, Julien Worms, and Sébastien Briais. The speedup-test: a statistical methodology for programme speedup analysis and computation. *Concurrency and computation: practice and experience*, 25(10):1410–1426, 2013.

[30] Vreda Pieterse, Vreda Pieterse, and David Flater. *The ghost in the machine: don't let it haunt your software performance measurements*. US Department of Commerce, National Institute of Standards and Technology, 2014.

[31] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *The 32nd International Conference on Software Maintenance and Evolution (ICSME 2016)*, 2016.

[32] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[33] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, 2018.

[34] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[35] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 559–570. ACM, 2013.

[36] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[37] Ke Tian, Danfeng (Daphne) Yao, Barbara G. Ryder, and Gang Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *MoST@S&P (W)*, 2016.

[38] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security (TIFS)*, 2017.

[39] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 431–444. ACM, 2013.

[40] Olga Gadyatskaya, Andra-Lidia Lezza, and Yury Zhauniarovich. Evaluation of Resource-based App Repackaging Detection in Android. In *Proceedings of the 21st Nordic Conference on Secure IT Systems*, NordSec 2016, pages 135–151, 2016.

[41] Fang Lyu, Yapin Lin, Junfeng Yang, and Junhai Zhou. Suidroid: An efficient hardening-resilient approach to android app clone detection. In *Trustcom/BigDataSE/I SPA, 2016 IEEE*, pages 511–518. IEEE, 2016.

[42] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *International Conference on Software Engineering (ICSE)*, 2014.

[43] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2015.